FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Application of Safety Verification Techniques on ROS Software

Tiago Neto

Master in Electrical and Computers Engineering

Supervisor: Armando Sousa PhD

Co-Supervisor: Rafael Arrais

July 24, 2019

Abstract

The increasing demand for custom-made products, readily available and cheap is causing a paradigm shift in the industry. This change is so flagrant that is being called the fourth industrial revolution or Industry 4.0. The cornerstones of the Industry 4.0 are Cyber-physical systems along with decentralized decision making and flexible production, achieved by the sharing of the work environment between collaborative robots and human operators.

However, this new perspective of the work environment raises safety issues, since the outdated barriers between human and machine are no longer present. In this new vision, the barriers are replaced by software alternatives that ensure the safety of the human operator. Therefore, it must be ensured that this software is safe and dependable.

Since robotic software is complex, it is not straightforward to ensure its safety. For that reason, some alternatives have been proposed. Static verification techniques are among these alternatives, providing insightful information and being able to detect potential issues since the early stages of development and without the need to run the system.

This work used the HAROS tool to perform static verification techniques to a mobile manipulator case study to access the quality and safety of its software. Furthermore, some of the issues found were tackled and the overall quality of the software and its safety were improved. This allowed to confirm the importance of static analysis and validate the used tool. The experience gained with this analysis allowed to produce a good practice guide for future use. Moreover, it was possible to suggest enhancements for the static analysis tool.

Resumo

A crescente procura por produtos personalizados, prontamente disponíveis e a baixo custo foi o motor de uma recente mudança de paradigma na indústria. Esta mudança é tão marcada que passou a ser designada por quarta revolução industrial (ou *Industry 4.0*). Os pilares desta são os *Cyber-physical systems*, em conjunto com a tomada de decisão descentralizada e a produção flexível, alcançada pela valiosa associação entre robôs colaborativos e operadores humanos no ambiente laboral.

No entanto, esta nova perspetiva leva ao surgimento de novas questões de segurança, uma vez que as barreiras antes existentes entre homem e máquina não estão mais presentes. Nesta nova visão, as barreiras são substituídas por software que garante a segurança do operador humano. Assim sendo, a segurança e confiabilidade deste elemento deve ser assegurada de alguma forma.

Como o software robótico é altamente complexo, não é fácil garantir a sua segurança. Portanto, algumas alternativas foram propostas ao longo do tempo. As técnicas de verificação estática contam-se entre essas alternativas, uma vez que fornecem informações detalhadas e são capazes de detetar problemas desde as etapas iniciais de desenvolvimento e sem a necessidade de executar o sistema.

Neste trabalho, essas técnicas foram utilizadas num caso de estudo robótico de forma a assegurar a qualidade e segurança do seu software. Além disso, alguns dos problemas encontrados foram resolvidos e a qualidade geral do software, assim como a sua segurança, foram aperfeiçoadas. Confirmando assim a importância da análise estática de software e a validade da ferramenta usada. A experiência adquirida com esta análise permitiu produzir um guia de boas práticas para uso futuro. Além disso, foi possível sugerir melhorias para a ferramenta de análise estática.

Acknowledgements

Firstly, to both my supervisors, Professor Armando Jorge Miranda de Sousa and Rafael Lírio Arrais, for their unmeasurable help and guidance throughout this journey.

To Pedro Melo and Henrique Domingos from INESC TEC for their friendship, help, patience, and advice.

To my parents for all their support, comfort, and advice throughout my life. This also applies to my grandparents and all of my family.

To all my friends from my home-town and from FEUP, especially the ones I shared room i105, for their friendship, support, and even for distracting me from work.

At last, but not least, I want to thank my girlfriend, for her patience, advice, support and love, that helped light up my path.

This work is partially financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583).

Tiago Filipe Miranda Neto

"Remember to look up at the stars and not down at your feet.
Try to make sense of what you see and wonder about what makes the universe exist.
Be curious.
And however difficult life may seem, there is always something you can do and succeed at.
It matters that you don't just give up."

Stephen Hawking

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AS	Authentication Server
AGV	Automated Guided Vehicle
API	Application Programming Interface
APM	Advanced Plant Model
CRIIS	Centre of Robotics in Industry and Intelligent Systems
CC	Cyclomatic Complexity
CPSs	Cyber Physical Systems
DNS	Domain Name System
DoS	Denial of Service
FASTEN	Flexible and Autonomous Manufacturing Systems for Custom-Designed Products
GPOS	General Purpose Operating System
HAL	Hardware Abstraction Layer
HAROS	High Assurance ROS
HASLab	High-Assurance Software Laboratory
IEEE	Institute of Electrical and Electronics Engineers
IDEs	Integrated Development Environments
LIDAR	Light Detection And Ranging
LOC	Number of Lines of Code
PCL	Point Cloud Library
PM	Production Manager
ROS	Robot Operating System
RTOS	Real-time Operating system
SCXML	State Chart Extensible Markup Language
TM	Task Manager
ToF	Time of Flight

Chapter 1

Introduction

Each day, the amount of robots in industry increases as they replace humans in repetitive and non-ergonomic tasks. However, there is a increasing demand for custom-made products that must be cheap and quickly available. This demand is forcing a revolution in industry, the so called Industry 4.0. This industry bring with it a reality composed of concepts such as the Cyber-Physical Systems (CPSs) and the collaborative robots. In both, robots collaborate and share the work environment with humans. This collaboration raises safety concerns about the robotic software to avoid risking human lives.

1.1 Context and Motivation

The shifting of paradigm caused by the industry 4.0 brought several new concepts, and it will revolutionize the production systems in the upcoming years. Driven by the consumers' demand of customized products with short development periods, the industry needs to be able to adapt. Therefore, to cope with these demands, the industry must become more flexible and reconfigurable with decentralized decision making [6].

A key concept in the industry 4.0 are the Cyber-Physical Systems. These deeply integrate physical and digital systems to allow a flow of information between them, leading to the decentralization of the information and an improvement in collaboration [6]. To provide the required flexibility, a collaboration between industrial robots and humans on the work environment is fundamental.

This evolution of the work environment of robotic systems is also pushing an evolution on their safety systems. Security techniques associated with classical robotic systems relied on physical barriers and other hardware systems to ensure the safety of the human operator. However, the new paradigm of robotics led to the need of more modern approaches. This implies the safety concerns of modern robotic systems depend on software systems. This paradigm shift makes the software verification a cornerstone for the safety and robustness of modern robotic systems. It is also expected that these verifications will play an important role in the certification and commercialization of modern collaborative robotic systems.

However, developing robotic software is complex and requires many expertise. This created the need for a thin, robust, general-purpose robotic software, as result, the Robot Operating System (ROS) was built [7]. ROS is a framework for the development of robotic software, with a collection of tools, libraries and conventions developed by the collaboration of several groups. The main objective of this software is to simplify the complex task of developing robotic software with robust behaviour for a wide range of robotic platforms.

The motivation for this work comes from the need to develop, adapt and verify techniques of safety verification of industrial robotics software to ensure its safety.

1.2 Problem Definition

The advent of autonomous, mobile and collaborative robots raised the security and safety concerns to a higher level. Also, the change of paradigm in modern robotics from the safety provided by physical barriers to safety ensured by software systems created the need to ensure that the software is robust and can guarantee the safety of human collaborators and the working environment.

The development of robust robots is a difficult and laborious task, as an harmonious integration of complex subsystems is required. Nowadays, ROS - a framework that provides libraries and abstractions to speed up the development of robotic systems - is very widespread among developers and industry. It is therefore expected that this framework will be widely used in the future. ROS also provides adaptability and configuration to robotic systems. Although these are some of the reasons for its success, they are also a problem that complicates the creation of tools to fully and easily certify the safety of ROS applications.

To promote the reliability of the software produced, ROS code relies on the use of rigorous assessment tools, which in turn will assure the final product is certified and able to meet market requirements. These requirements comprise both the functioning of each system and the global behaviour of it. The SAFER project [1] , in which this work is integrated brings together the expertise of computer scientists, with a background on software system design and analysis, and experienced robot engineers, to develop such techniques in the context of ROS robots. The role of static analysis is fundamental in the verification of performance, safety and adaptability in ROS-specific and general-purpose source code.

The role of static analysis is fundamental in the verification of performance, safety and adaptability in ROS-specific and general-purpose source code. Therefore, the aim of this work, performed in collaboration with Centre of Robotics in Industry and Intelligent Systems (CRIIS) and High-Assurance Software Laboratory (HASLab) from INESC TEC, is to validate, develop and implement verification techniques. These techniques are intended to adapt and correct nonconforming or unsafe code present in the robotic systems of a case study developed by INESC TEC.

[1] https://www.inesctec.pt/pt/projetos/safer

1.3 Objectives

The goal for this work is to promote better code quality and safety of industrial robot software, more precisely for the Flexible and Autonomous Manufacturing Systems for Custom-Designed Products (FASTEN) [2] case study, provided by INESC TEC. To achieve that, the case study must first be studied, understood and described. With that knowledge acquired, static verification tools will be used, namely the High Assurance ROS (HAROS) framework [8], which was developed by HASLab. Under the SAFER project, these tools will be used to not only analyse the source code but also to analyse its architecture and extract valuable information crucial to improve its software quality and its safety.

1.4 Contributions

This dissertation work supports the importance of static analysis in robotics and ROS-based software. Moreover, it validated HAROS tool in the context of modern robotics, namely mobile manipulators, at INESC TEC - CRIIS. With the experience gained using HAROS, it was possible to create a good practice guide. This guide will be adopted by the INESC TEC - CRIIS developers in future projects. Furthermore, some suggestions to improve HAROS and the results achieved with its usage are included.

1.5 Structure

To conclude this introduction, a brief insight into the structure of this dissertation is now provided. The structure is as it follows.

- **Chapter 2, *"Fundamentals"*** - In this chapter, the fundamental concepts needed to understand the work developed in the following chapters are provided. The presented concepts provide insight into this field of study.
- **Chapter 3, *"Background and Related Work"*** - In this chapter, a general overview of problems associated with ROS is presented. Additionally, informations concerning approaches to verify and improve the safety of ROS software is provided, along with a review of related work with the safety verification tool that will be used.
- **Chapter 4, *"Case Study"*** - In this chapter, is presented the case study that is the base for this work, concerning the project, the robot and its architecture.
- **Chapter 5, *"Analysis"*** - In this chapter, the methodology adopted for this work is presented, along with the results of its application on the chosen case study.
- **Chapter 6, *"Conclusion and Future Work"*** - Finally, in this chapter the main conclusions about this work are drawn. A perspective about the future of this work is also shared.

[2] http://www.fastenmanufacturing.eu/

Chapter 2

Fundamentals

This chapter will comprise a review of the fundamental concepts and tools of this work. The points addressed in this chapter are base notions of Real-Time Systems, Security, Dependability, and ROS. This chapter will focus on safety, vulnerability to attacks and real-time compliance in ROS. ROS 2 will also be briefly reviewed, regarding the same focus points as reviewed for ROS. However, ROS 2 will not be the focus of this work, since ROS is by far the most used framework.

2.1 Base concepts

2.1.1 Real-Time Systems

Real-time computing systems rely on three essential characteristics revolving around time. First, to ensure the correctness of this kind of systems, they must work promptly. This means that its correctness does not only rely on the logic value of the performed operations but it also considers if they were performed on time. Hence, these systems' tasks and messages must be scheduled in a way that ensures its completion, reception or sending promptly. Secondly, these systems must be reliable since, in case of failure, they might compromise human physical integrity. Lastly, real-time systems rely on an active component, the machine where the system itself runs [9].

2.1.2 Security and Safety

Security and safety are two very similar words. However, despite this similarity, they have different meanings.

A system can be considered safe if it does not have catastrophic consequences on its users and the environment. In other words, if it is not a threat to human physical integrity. This definition is different from the definition of security.

Security in a system is defined by three attributes: availability, confidentiality, and integrity. This means a system is secure when no unauthorized information is revealed. Confidentiality is thus assured since there are no unauthorized alterations to the system and the system is only available to perform authorized action [1].

2.1.3 Dependability

Dependability has two definitions: the first defines it as the ability of a system to provide a service that can be trusted and, on the other hand, the alternative definition is the ability of a system to avoid frequent and severe service failures. Ultimately, dependability is a concept which comprehends five attributes: availability, reliability, safety, integrity and maintainability. Therefore, a dependable system is ready to provide correct service in a continuous manner (availability and reliability), without compromising human physical integrity or its environment and without unauthorized alterations to itself (safety and integrity) while it is also able to be modified and repaired (maintainability) [1].

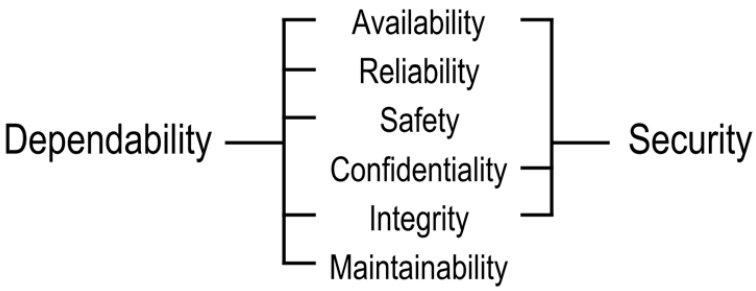

Figure 2.1: Dependability vs Security [1]

2.2 Software Metrics

Institute of Electrical and Electronics Engineers (IEEE) defines software metrics as *"a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality"* [10].

Software metrics are extremely useful for quality assurance. These metrics provide a reproducible and quantitative quality measurement. This evaluation should be performed during the different phases of the software development life-cycle, [11] in order to control and improve the software [12].

These metrics can be classified into 3 different types: (i) procedures metrics, (ii) project metrics, and (iii) product metrics. Procedure metrics comprise the development, mainly concerning the duration of procedures, effectiveness of the used methods, improvement of procedures, and their predictions. These metrics are used by high-level management for the control and management of the development stage [12]. On the other hand, the project metrics are used to control and interpret the project status. Based on them, adjustments can be applied to technical methods and management strategies. These adjustments mitigate risks and optimize the development, consequentially improving the quality of the product [12]. Finally, product metrics are used to control and predict the quality of the product. To perform that evaluation, some of the key metrics are software complexity, reliability and maintainability.

Among the software metrics, the most commonly used are: Cyclomatic Complexity (CC), Halstead Complexity (HC) and Number of lines of code (LOC) [11].

2.2.1 McCabe Complexity

In 1976, driven by the need to modularize a software in such a way that the resulting modules could be tested and maintained, Thomas McCabe developed a mathematical approach technique to face this necessity [13].

The developed technique was based on a graph, the control flow graph. This graph is composed of nodes that represent a block of code with the sequential flow, and arcs that represent a branch of the flow of the program. Each graph has a unique entry and exit nodes. Through these graphs, it can be controlled and measured the number of basic paths of a program [13].

A function of this graph is then used to calculate a cyclomatic number, the cyclomatic complexity. This function is pictured on equation 2.1, where e is the number of edges, n is the number of nodes. This metric provides 2 major contributions, first its value gives the recommended test for the software, and second, when used throughout the life cycle of the software maintains it reliable, testable and manageable [14].

$$V(G) = e - n + 2 \tag{2.1}$$

An example of a block of code and its respective control flow graph can be found in figure 2.2. Applying the equation 2.1 to that flow graph it is obtained a cyclomatic complexity of $V(G) = 9 - 8 + 2 = 3$

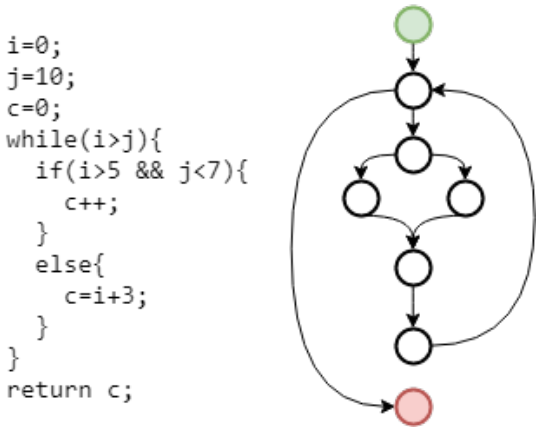

Figure 2.2: Cyclomatic Complexity and Control Flow Graph example

Measuring the cyclomatic complexity using a control flow graph and equation 2.1 can be complex, and if done by hand it is very prone to errors. Conveniently there are many easier ways to obtain that value. A commonly used approach is counting binary decisions (p) and the cyclomatic complexity is given by equation 2.2. In other words, with this method, each *if*, *while*, *for*, each *case* of a *switch*, *&&* and || for example add 1 to the cyclomatic complexity value [14].

$$V(G) = p + 1 \tag{2.2}$$

Now, applying the method of equation 2.2 on the control flow graph and code of figure 2.2 the obtained value is $V(G) = 3 + 1 = 4$. The value of p is obtained with: *while* +1, *if* +1, and *&&* +1, thus $p = 3$.

As a general rule, it is recommended to keep the cyclomatic complexity to a limit of 10. This is justifiable by the fact that very complex modules will be harder to test, modify, maintain, understand and will also be more prone to errors. Limits up to 15 have been used but only when projects have experienced staff and if willing to commit extra effort to test these complex modules.

2.2.2 Halstead Complexity

The Halstead complexity metrics, developed in 1977 by Maurice Halstead, are among the first methods to measure the complexity of code. For this evaluation the code is divided in tokens. Each of this tokens is classified in either operator or operand, being this the base for this evaluation [15]. The unique operators (*n1*), unique operands (*n2*), the total number of operators (*N1*), and the total number of operands (*N2*) are the base values of this metrics, from which the following equations depend on [15].

$$n = n1 + n2 \tag{2.3}$$

$$N = N1 + N2 \tag{2.4}$$

$$V = N \cdot \log_2(n) \tag{2.5}$$

$$D = \frac{n1}{2} \cdot \frac{N2}{n2} \tag{2.6}$$

$$E = V \cdot D \tag{2.7}$$

$$B = \frac{E^{\frac{2}{3}}}{3000} \tag{2.8}$$

From equation 2.3 it is obtained the size of the vocabulary, and from equation 2.4 the program length. The value obtained with equation 2.5 is the program volume and is on the most important value of this metrics. This volume is measured in mathematical bits and its value can be used to determinate if a function or a file are to complex. When a function has a volume over 1000 or a file has a volume over 8000 it is possible to assume that they are too complex. The equations 2.6 and 2.7 represent the difficulty and the effort to implement the code, respectively. Finnaly, the last equation, equation 2.8, which is also very important, provides a estimation of the amount of bugs delivered. This number of bugs should be under 2, although experiences showed that files contain more bugs then the value given by equation 2.8 [15].

2.2.3 Maintainability Index

Proposed in 1991 by Oman and Hagemeister, the Maintainability Index is a very useful metric to evaluate the maintainability of software [16]. The maintainability of the software is extremely relevant since from the cost of developing software 40 to 60 per cent is spent on maintenance [17]. This metric can be used to assess the quality of the code, predict and detect defect prone code, determinate if perfective maintenance should be done, and if the system should be re-engineered [17].

This metric can be calculated using equation 2.9 or with equation 2.10 [18]. The equation 2.9 provides the maintainability index without accounting the influence of comments present in the code, while equation 2.10 have them in account. Equation 2.10 should only be used to calculate the maintainability index if the bulk of the comments present in the software are considered appropriate and correct. If not, equation 2.9 is potentially more adequate [16]. The values on which this equations depend are: average Halstead Volume (*aveV*), average extended cyclomatic complexity (*aveG*), average lines of code (*aveLOC*) and average percent of lines of comments (*perCM*) and are all per module values.

$$MIwoc = 171 - 5.2 \cdot ln(aveV) - 0.23 \cdot aveV(G) - 16.2 \cdot ln(aveLOC) \tag{2.9}$$

$$MIcw = MIwoc + 50 \cdot sin(sqrt(2.4 \cdot perCM)) \tag{2.10}$$

The values yielded by the equations above should be interpreted in the following way: (i) Over 85 - Good maintainability , (ii) 65-85 - Moderate maintainability , (iii) Under 65 - Difficult to maintain. Modules which had become unmaintainable should be rewritten, to avoid risky code [18].

2.3 Robotics and programming Middleware

A middleware can be defined as an abstraction layer that operates between the operating system and the software application. The objective of middleware is to simplify software development. This is achieved by providing software tools that abstract the heterogeneous hardware used in robotics. With these tools, the development costs reduced since it is no longer necessary to develop software for every component of the robot [19]. This hardware abstraction allows reusability of software and also provides scalability and opportunity to upgrade or replace components without the need to redevelop software.

The advantages of middleware lead to the development of several different approaches. Among the most used middlewares we find: the Robot Operating System (ROS), Player, Open Robot Control Software (Orocos), Microsoft Robotics Developer Studio (MRDS), Orca, Open Platform for Robotic Services (OPRoS), Coupled Layer Architecture for Robotic Autonomy (CLARAty), Middleware for Robotics (Miro),Evolution Robotics Software Platform (ERSP), Carmen, and several other [19] [20].

Table 2.1: Comparison between Robotic Middleware

Middleware	OS	Programming Language	Open source	Distributed architecture	HW interfaces and drivers	Robotic algorithms	Simulation	Control / Realtime oriented
ROS	Unix	C++, Phyton, Lisp	Y	Y	Y	Y	+/-	N
Orocos	Linux, OS/X	C++	Y	Y	Y	Y	N	Y
MRDS	Windows	C#	N	Y	Y	Y	Y	N
Player	Linux, Solaris, BSD	C++, Tcl, Java, Python	Y	+/-	Y	Y	Y	N
Orca	Linux, Win, QNX Neutrino	C++	Y	Y	Y	+/-	N	N
OPRoS	Linux, Windows	C++	Y	Y	Y	Y	Y	N
CLARAty	Unix	C++	Y	Y	Y	Y	N	N
Miro	Linux	C++	Y	Y	Y	N	N	N
ERSP	Linux, Windows	?	N	Y	Y	Y	N	N
Carmen	Linux	C++	Y	Y	Y	Y	Y	N

The large amount of middleware solutions supports the idea that modern robotic development is very complex and highly specialized. That difficulty leads the researchers to develop solutions tailored to their specific needs. When comparing these robotics middleware,from table 2.1, it is possible to notice that very few are oriented towards real-time control, which leads to the conclusion that the research effort is being place on high-level problems [20]. It is also noticeable that most of the frameworks have modular architectures to allow the reusability and reduce the integration efforts. Most of this middleware use C++, which can be justified by this language greater speed and better performance [20].

Within the robotics community the middleware that is being more widely used and more attractive to the robotics community is ROS [20]. This preference can be justified by the extraordinary tools provided by this framework and by the vast amount of algorithms for a wide range of applications. On top of that, ROS is also open source [20].

2.4 Robot Operating System

ROS is an open-source framework whose primary goal is to support the reuse of code in robotics research and development. This is accomplished by a wide range of tools, libraries and conventions provided by ROS, that are incorporated in packages. These packages support a large amount of robots, and the amount of robots keeps increasing every year, supporting the widespread use of

this framework. These packages are developed by a vast community and shared through repositories, which facilitates the development of code [21]. The reuse of code is important, since developing a robotic system is complex it requires expertise. However, being open-source gives developers the possibility to use the standard code and if necessary develop or change code for a more specific use.

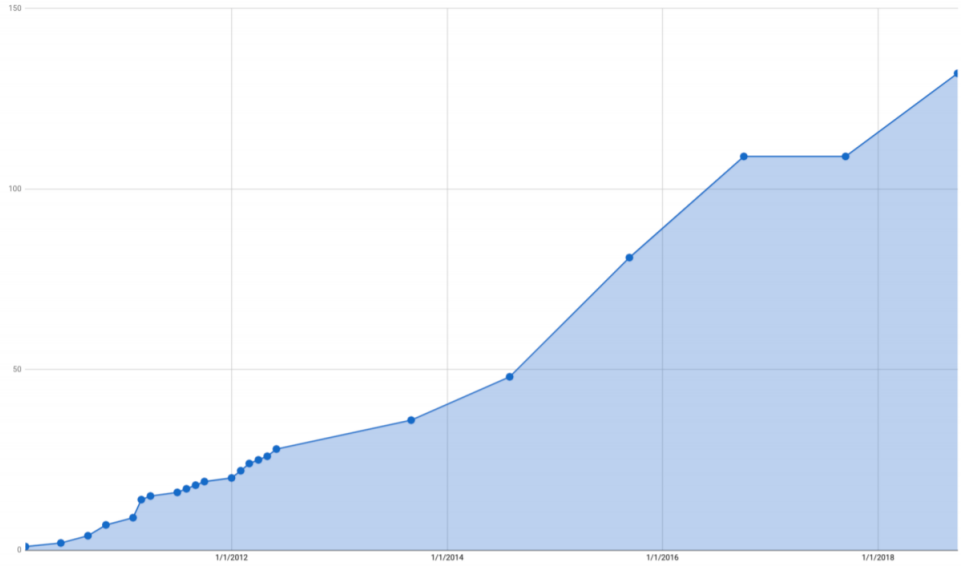

Figure 2.3: Evolution of different robots available in ROS [2]

Even though ROS has many advantages, it still has some problems that need to be addressed to ensure safe and dependable use in safety-critical contexts, such as industry, transportation and health. Two main problems need to be taken into consideration to allow the use of ROS in the contexts described above. First, ROS is not a real-time framework [21] and, secondly, ROS is vulnerable to attacks.

2.4.1 ROS Architecture

ROS was designed to be modular and has a peer-to-peer architecture. This means its processes are distributed and communicate with each other to accomplish a multi-function system [22].

The processes are known as nodes. Each node is responsible for the computation of a part of the system, such as path planning, control of wheel motors and others. All nodes need a Master. The Master acts as a name-service, registering the nodes and their information and enabling the communication between nodes by working like a DNS (Domain Name System) server [22]. Nodes communicate with each other through messages, which can comprise several data types, from the most basic to the most complex structures. To distribute these messages, a publisher-subscriber pattern is used, where the messages are published on topics.

A topic is a message bus. Its name is used to identify the content of the messages to be traded. There, any node can send (publisher node) or receive (subscriber node) messages as long as they

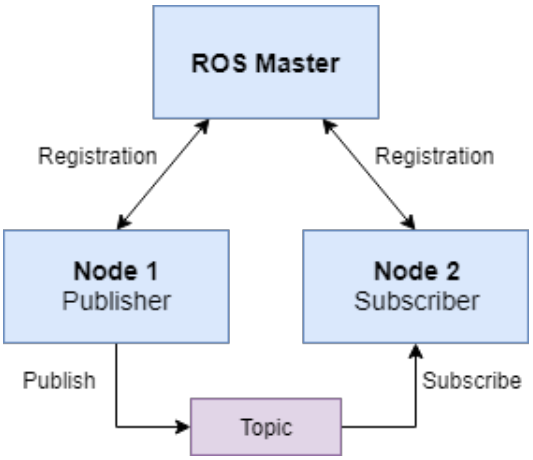

Figure 2.4: ROS Publisher-Subscriber Communication

are connected to the bus and the messages are the right type [22]. This type of communication is usually of 1 publisher to n subscribers.

The communication under topic and the publisher/subscriber model is very flexible. However, due to being only one-way communication, it is not appropriate for request/response communication. This type of interaction is provided by services, which are provided by nodes under a certain name. These services are characterised by a pair of messages, one for the request and the other for the response.

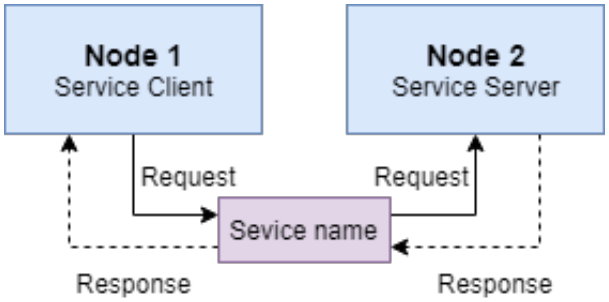

Figure 2.5: ROS Services Communication

Another very useful Client-Server communication are ROS Actions. These are similar to ROS Services but add some extra features. They are better suited for cases where services take a long time to execute. In these cases, the client might want/need to cancel the request and might also want/need to receive periodic information about the execution of the process. ROS Actions communication uses a special set of messages between client and server. This communication scheme and the messages that can be traded between ROS Action Server and Client can be seen in Figure 2.6. Messages can be Goal or Cancel, from client to server, and Status, Feedback or Result, from server to client. The goal is a message sent from client to server which contains relevant parameters for the action to be performed. From client to server another kind of message that can be sent, is the cancel message. This message allows cancelling one, several or all goals that were previously requested. From server to the client there is the result message, which is sent when the

goal is completed and additional information about the completed goal. Before the completion of the goal, other messages are being sent from server to client. Those are the feedback and status messages. Status messages are updates about the status of every goal on the system, they are sent periodically at a fixed rate. Feedback messages are updates on the evolution of the goal. Those messages contain additional information useful for the action clients [23] [24].

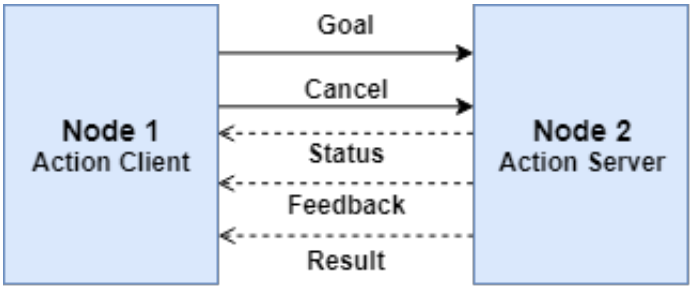

Figure 2.6: ROS Actions Communication

TCP/IP in its standard form is the most commonly used communication protocol between ROS nodes.

2.4.2 ROS and Real-time

The standard version of ROS is not a real-time framework [21]. This is due to the fact that it is based on the Ubuntu distribution of Linux, which does not fulfil the real-time requirements of the industrial control technology [25].

Even though ROS is a real-time framework, some solutions can be integrated to provide real-time capabilities.

2.4.3 ROS Security

The evolution of the robotic and industrial systems powered by Industry 4.0, and its main ideas, created the idea of Cyber-Physical Systems (CPSs). The main goals of Industry 4.0 are to increase productivity, production quality and workers health, with the CPS being the cornerstone of the improvements [26]. The Cyber-Physical System englobes all physical systems that can exchange data, via software, with other physical systems and human operators. CPS can then be defined as a network that integrates physical systems, like actuators and sensors, and computational data, allowing the decentralization and cooperation between systems under a Cloud environment [26].

Despite the advantages provided by Industry 4.0, there are also some issues, namely in the security field, that must be addressed to not jeopardize human physical integrity or the systems itself. This and other issues are also delaying the mass use of ROS in industry. There are already some detected problems and some solutions for those problems, but they must be solved in a manner that does not compromise other parts of ROS.

Due to the increasing levels of connectivity with exterior networks, brought by Industry 4.0, ROS-based applications became more vulnerable to cyber attacks. [27]

The main security issues in ROS applications are: Unauthorized Publishing (Injections), Unauthorized Data Access and Denial of Service attacks (DoS) on specific ROS nodes. [28]

2.4.3.1 Unauthorized Publishing (Injections)

In ROS-based applications, nodes do not need authorization to publish on a topic, which is a problem. The lack of authorization allows the injection of false data or commands into an application.

2.4.3.2 Unauthorized Data Access

The ROS structure allows all nodes to subscribe to any topic within the application. After subscription, they will receive all the data that is being published on that topic. The data is published on the topic could contain confidential or business-critical information which can be easily used since it is not encrypted.

2.4.3.3 Denial of Service attacks (DoS) on specific ROS nodes

DoS attacks can be performed on ROS by publishing large amounts of fake data. This will leave subscribers with a high processing load, preventing them from performing useful processing.

2.4.4 ROS Code Quality

Code quality has a complex definition. Quality comprises two different aspects: external and internal code quality. External quality is dependent on what the program can achieve and how it performs. Its usability is also an important attribute, particularly when it comes to user perspective and experience. On the other hand, internal quality is dependent on the source code itself. To have good internal quality, the code should be clean, structured, and organized. It is also important that it is easy to understand, maintain and upgrade [5].

2.4.5 ROS Code Metrics

Among ROS developers, metrics are not consensual. Nonetheless, there are some metrics which are more commonly accepted. These metrics can be divided into one of three distinct classes: function-based, class-based or file-based metrics. General code metrics are presented in Table 2.2.

Table 2.2: Summary of code metrics accepted by ROS community (from [5]).

File-based	Function-based	Class-based
Comment to code ratio	Cyclomatic complexity	Coupling between objects
	Number of executable lines	Number of immediate children
	Number of function calls	Weighted methods per class
	Maximum nesting of control	Deepest level of inheritance
	Estimated static path count	Number of methods available in class

2.5 ROS 2

With the increase of popularity and the standardization of ROS usage in many robotic systems, the security risks associated with its use become more relevant. Since ROS was not designed to deal with potential hostile attackers, the need for a similar system but with security concerns in its design appeared.

Robot Operating System 2 comes out as an answer to security and other concerns, such as real-time concerns, that plain ROS could not satisfy.

Chapter 3

Background and Related Work

In this chapter, a review of the literature about ROS will be made. This review will focus on safety, vulnerability to attacks and real-time compliance.

3.1 ROS and Real-time

As stated in Chapter 2, ROS is not a real-time framework, despite that there are ways to qualify ROS with real-time capabilities.

In order to provide real-time capabilities and guarantees for systems running ROS, two different approaches can be applied [3]. One of the proposed approaches is the inclusion of embedded real-time systems to ROS, while the other is to export ROS packages to a Real-Time Operating System (RTOS), and a third, a hybrid one [3].

One option following the inclusion of tools to provide real-time capabilities to ROS is called ROSCH (Real-Time Scheduling Framework for ROS). This framework rely on three main functionalities to ensure real-time performance. One of the main functionalities is a Synchronization system, the second is a fixed-priority based Directed Acyclic Graph Scheduling framework and third a fail-safe functionality [29].

On ROSCH framework the synchronization system is used to guarantee that on a node which gathers multiple topics, there are no significant difference between the timestamps of data from different sensors(Publishers), for which that node is subscribing to. This is done by adjusting the publishing period of the publishers, making them publish simultaneously. This function might not be necessary if the scheduling framework is used, but even in that case it is still useful for keeping the timestamps gap between a certain value when a node fails a deadline [29].

The hybrid approach is based and a dual-core processor, in one core runs a RTOS, namely Nuttx, and on the other core runs a General Purpose Operating System (GPOS), namely Linux. This hybrid system is called RGMP and has two main functions, initialize the system and create a communication bridge between the RTOS and GPOS. On this system there are two kinds of ROS nodes, the real-time nodes and the non-real-time ones, this nodes can communicate between

each other under the TCP/IP protocol, using the system virtual network to communicate between real-time and non-real-time nodes [3].

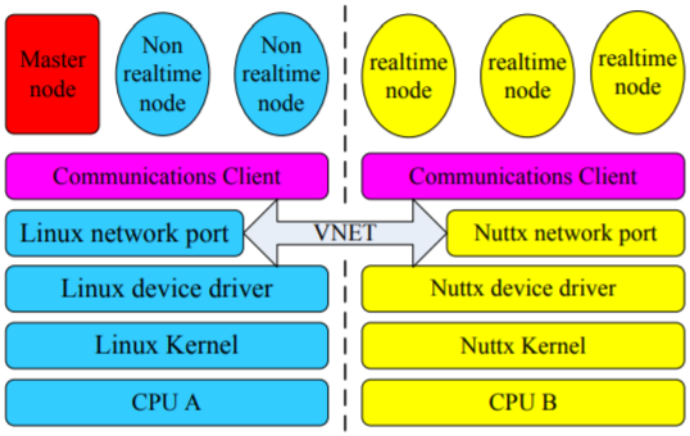

Figure 3.1: Architecture of RGMP (From [3])

3.2 ROS Safety Verification

Safety is a very important characteristic of collaborative, autonomous and mobile robots. ROS provides developers with a large set of customizable robot services and libraries. Due to this diversity, there is no solution to completely analyse and verify ROS programs in a formal way and certify their safety to guarantee correct behaviour of the robots (i.e. to guarantee that robots do not perform unsafe or unauthorized actions). ROS, however, does not impose strict development rules to ensure its safety. It only suggests two mechanisms to ensure the safety of its software [8]. The first is a style guide based on the Google C++ Style Guide, the ROS C++ Style Guide, while the second is a set of metrics quality metrics thresholds [8].

The mechanisms advised by ROS are not sufficient and not restrictive in terms of safety. In the automotive and the aerospace industry, safety is taken much more seriously. These areas have much more strict guidelines, such as MISRA C++ and JSF AV C++, from which ROS could benefit [8].

Given this lack of standardized strict and robust safety mechanism for ROS, some approaches have been proposed to tackle this problem, such as runtime monitors, static analysis techniques, model checkers and proof assistants [30].

3.2.1 Runtime monitors

Runtime monitoring is a technique that can be used throughout the development cycle of software, from initial testing to actual system deployment. This technique uses monitors to follow system properties that the user is interested in, from global behaviour of the system to behaviour of objects, during a program execution. During runtime, the traced properties are verified and appropriate actions are triggered, accordingly to validation or violation of the property being traced [31].

A example of the use of this technique on ROS is ROSRV (ROS Runtime Verification). This solution implements a intermediate node, named *RVMaster*, between ROS Master and the other nodes. This framework is focused on two main points: monitoring safety properties and enforce security policies. The security policies are similar to the solutions previously reviewed, defining which nodes can subscribe or publish to each topic and the commands that each node can execute. The safety properties monitoring traces some property of interest, by creating nodes that can publish and subscribe, i.e. acting as a men-in-the-middle. This monitors are event-based, acting or stooping actions accordingly to the registered sequence of events.

ROSRV uses Monitoring Oriented Programming, which allows to specify temporal properties of events and to trigger actions on specific sequences of events. This framework is also capable of generating C++ code, transforming the specifications on the monitor nodes [32].

3.2.2 Static Verification

Static analysis techniques are one of many software engineering techniques that can elevate the quality of code. This simple and time-efficient technique allows, since an early phase of development, to extract precious information from a program without running it. Among the collected information internal quality metrics and conformity with coding standards are the most important [8]. When applied to ROS, this technique should yield information about the behaviour of each of its subsystems and the interactions between them [33].

Despite the potential of this technique, applying it on ROS is not so straightforward. As previously mentioned, ROS is very customizable, has a large amount of primitives and is written in several different programming languages. This diversity leads to an extremely complex and unfeasible implementation of a static analysis for an ROS arbitrary system. Nevertheless, for a more restrict set of ROS subsystems it could be achievable [33].

A example of a static analyser for ROS-based code is HAROS. HAROS was developed with two fundamental ideas: (i) is the integration with ROS specific settings, and (ii) it should not be restrictive, allowing the use of a wide range of static analysis techniques. The idea (ii) leads to HAROS allowing the integration and use of third-party analysis tools, as plug-ins [8].

This tool allows the fetching of ROS source code, its analysis and the compilation of a report in an automatic way. Therefore, it can be easily used, even by developers without extensive knowledge of ROS or static analysis techniques. The properties that are analysed can be of two categories: rules or metrics. Rules report violations and metric return a quantitative value. Concluded the analysis step HAROS generates a report and displays the results in a graphical form [8].

3.2.3 Formal Verification

Formal verification is a technique that comprises both security and liveliness, i.e. guarantees that no unwanted behaviour will occur and ensures the software will meet the required performance levels . This technique uses mathematical models, formal models, to represent the system. This

models can then be checked. Model checking is extremely useful since it allows to test all possible combination of events. With that unwanted behaviours, resulting of unthought combinations, can be detected and removed [34].

A popular formal model is called Timed automata and allow to verify and specify real-time systems. This model has already been used with ROS-based applications. To create a formal model was used a publisher-subscriber implementation. The key parameters of the implementation were extracted. Later the parameters were used to create the formal model, which consisted of a network of timed automata. Using *UPPAAL*, a model checker, were tested several combinations of values of parameters. As result real-time properties of the ROS application where verified. The model were also used to search parameters to validate properties of a specific robot [30].

3.3 ROS Security

ROS, as stated in Chapter 2, has some lacks in security, which need to be addressed to increase its robustness.

The main vulnerabilities of ROS come from the lack of authentication and authorization to access the nodes and their communication. A simple improvement, that could allow security systems to identify a ongoing attack, can be made on the configuration of ROS. Changing the ROS master network port will force the attackers to perform a full port scan, which might be detected [28]. This change alone will not prevent malicious attacker from tempering with the systems and gaining access to its data, therefore other security measures, that focus the authentication and authorization vulnerabilities, are needed.

3.3.1 Application-Level Security

To address the lack of authentication and authorization, a first approach can be performed at the application level. To tackle the the lack of authentication, an Authentication Server (AS) is introduced [28] to keep track of which nodes subscribe or publish to a topic.This will only keep unauthorized nodes from subscribing or publishing, which will prevent the injection of false data. To solve the authorization problem the data on topics is encrypted, with the topic-specific encryption key being generated and stored by the AS, and only given to authorized nodes, meaning that the data can not be eavesdropped by unauthorized parts [28].

Despite the security increase, provided by this application-level solution, there are still vulnerabilities. This solution does not prevent DoS attacks since it does not prevent nodes from joining ROS graph and publishing messages, that are meaningless since authentic nodes won't get them, which allows the attacks.

3.3.2 Communication Channel security

To further increase the security, a change in ROS communication is necessary. A possible solution, to secure the communications would be using Transport Layer Security (TLS), Datagram

Transport Layer Security (DTLS) and topic authorization for each node. The TLS does a first handshake in a encrypted manner to ensure data confidentiality. If successful, it allows the following TCP communications to be performed in an encrypted, authorized and authentic way. The data is authenticated by the Message Authentication Codes (MACs) [28].

3.4 HAROS

The majority of proposed solutions to address safety concerns of modern autonomous robotic systems, as well ans their software development practices disregard existent standards and follows non-standard approaches in the development of safe robots. This option for non-standardized approaches limits the credibility of this solutions and restricts its usability [35].

Contrarily to other solutions, the HAROS framework uses standard techniques and integrates them in a ROS specific solution. Another advantage of this tool is the verification of conformance with strict code standards that are widely adopted in safety critical areas, such as the automotive and aerospace industry. This standards are te MISRA C++ [36] and JSF AV C++ [8]. Besides the verification of the source metrics and code standard conformance, it is also capable of extracting process metrics such as the number of commits [8].

This tool already allowed the mining of several ROS repositories and allowed to understand how certain ROS base features are being used [33]. Collecting this information could help the community to clarify features that are ether being less used or even misused. It is also helpful for the developers of static analysis tools to understand which specific features to support in the future [33].

Additionally, this framework provides a feature that no other does. The feature is a model extractor with the capability to obtain the ROS Computation Graph at compilation time [4]. This graph is the representation of the architecture of ROS systems, which after extraction HAROS though is visualization tool provides, such as the example of Figure 3.2.

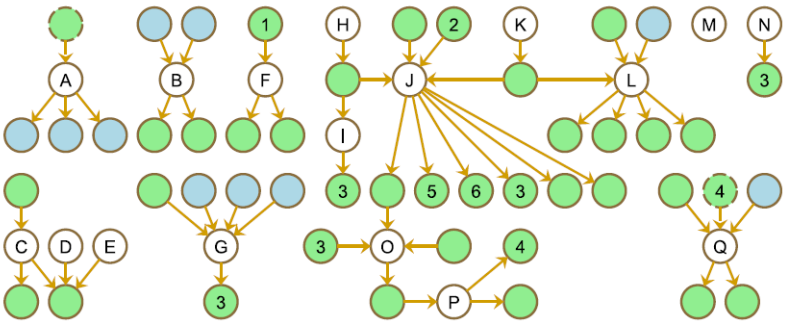

Figure 3.2: Architectural model of AgRob V16 [4]

For a developer, being able to obtain the architecture of the system just by compiling it is extremely relevant. Usually, this information can only be obtained with runtime analysis with tools

such as *rqt_graph*. However, this is very time consuming and does not cover all the possible execution paths. Therefore, the immediate and easy to access to the architecture, provided by HAROS, allows the developers to easily reason about it and its properties. Thus, allowing developers to be able to find issues in the architecture in a faster and easier way [4].

Chapter 4

Case Study Presentation - FASTEN Project

This work will have the FASTEN[1] case study as its main focus. On this chapter, the FASTEN case study will be described. The scope of this is the description of the project itself, more specifically the robot and its characteristics, the software architecture, the coding standards followed during the software development and the functional properties of the system.

4.1 FASTEN Project

The Flexible and Autonomous Manufacturing Systems for Custom-Designed Products (FASTEN) project purpose is to develop, demonstrate, validate, and disseminate a modular and integrated framework able to efficiently produce custom-designed products. FASTEN will also prove the concept of a open and standardized framework able to produce and deliver these custom-designed products in an autonomous, fast and low-cost manner. To achieve that digital integrated service/products manufacturing processes, decentralized decision-making and data interchange tools will be integrated. Other technologies such as sophisticated self-learning, self-optimizing and advanced control software will also be implemented to establish a fully connected and additive manufacturing system.

This project focus on helping two companies, ThyssenKrupp and Embraer, surpass challenges such as the increasing demand diversity, products with shorter life cycles and supplying low volume per order needs. To tackle these challenges flexible solutions adequate to manufacture and deliver custom products need to be used. For this work, the focus will be the Embraer case, which is being tackled by INESC TEC.

For the Embraer case, the robot should assemble kits using a robotic arm and possibly transporting the kits. For the transport, a solution is combining a Automated guided vehicle (AGV) with the robotic arm. The parts that compose the kits come from an Automated Warehouse System within a box.

[1] http://www.fastenmanufacturing.eu/

The FASTEN project was chosen since it is a recent and complex project that represents well the modern robotics. This project is also aligned with INESC TEC and CRIIS goals.

4.2 Robot

In order to accomplish the desired capabilities, the right components for the robot were selected by INESC TEC. One of the fundamental requirements for the components was that it should be possible to integrate them with the ROS framework.

The robot platform, the AGV, has omnidirectional wheels, which allow the robot to move in any direction. This platform was designed and build by INESC TEC. Incorporated on the platform there are two SICK S300 Expert laser scanners. Their principle of work is the Time of Flight (ToF), which is a measurement of time between the emission and reception of the reflected light signal, that can be converted to the distance between the sensor and reflecting point. Therefore they are used to locate the robot in the work environment, by measuring its distance to known reflecting points. These lasers are also used for safety reasons, to prevent collisions with either people and objects. This robot platform and the other hardware components can be seen in Figure 4.1.

Figure 4.1: FASTEN Mobile Manipulator

On top of the robot platform, there is a robotic arm, more specifically a Universal Robots UR10. This robotic arm can operate with a payload of up to 10 Kg, has a reach of 1300mm and 6 degrees of freedom. On the robotic arm, there is a 3D camera, a Photoneo PhoXi 3D Scanner. This camera produces a point cloud, which can be used to create a virtual 3D model of the part. It is responsible for the detection of the parts to be pick by the robotic arm, and it is also responsible

to detect their position and orientation. All of that information is of extreme importance to plan a collision-free trajectory for the robotic arm and its gripper, preventing damage to the equipment and to the parts. The camera and robotic arm used on the robot can be seen in Figure 4.2.

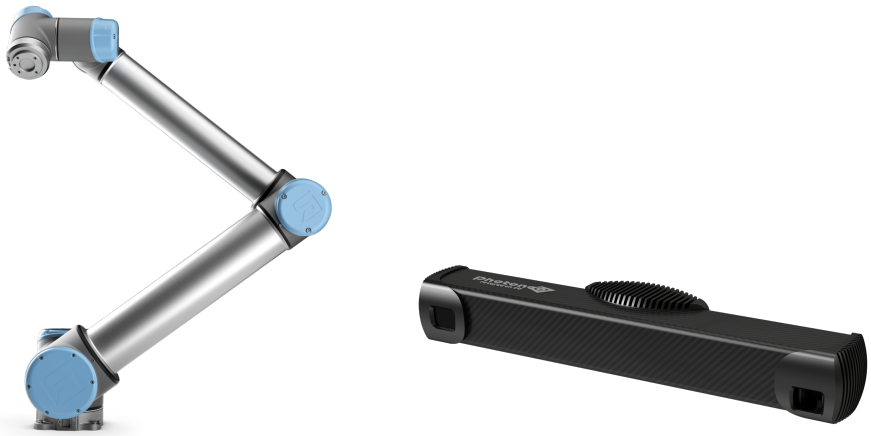

Figure 4.2: UR10 Robotic arm (left) and Photoneo PhoXi 3D Scanner (right)

To accomplish the picking of the previously found part the robotic arm has 2 different grippers to perform that task. One of those grippers is a Robotiq 2-Fingers. This single gripper can pick parts of different shapes and sizes and also allows the adjustment of the gripping force. The other gripper is a Schmalz Area Gripper. This one is highly versatile since it can handle objects regardless of their size, geometry, material and surface, mostly due to its high holding force. The two grippers can be seen in Figure 4.3.

Figure 4.3: Robotiq 2-Fingers Gripper (left) and Schmalz Area Gripper (right)

4.3 System Software Architecture

The software architecture of this robotic system was developed with three main objectives in mind, that lead to three structural ideas. The first objective was to reduce the cost of adapting robot applications by promoting the code re-usability. To achieve the first objective the idea was to use

Skill-based robot programming. A ROS skill can be defined as the set of operations that a robot can execute. The second objective is to promote intuitive and flexible robot programming, achieved by Task-Level Orchestration. The third objective is to support generic interoperability with Manufacturing Management Systems and Industrial Equipment, with that purpose the components of this architecture were developed to allow Vertical and Horizontal Integration.

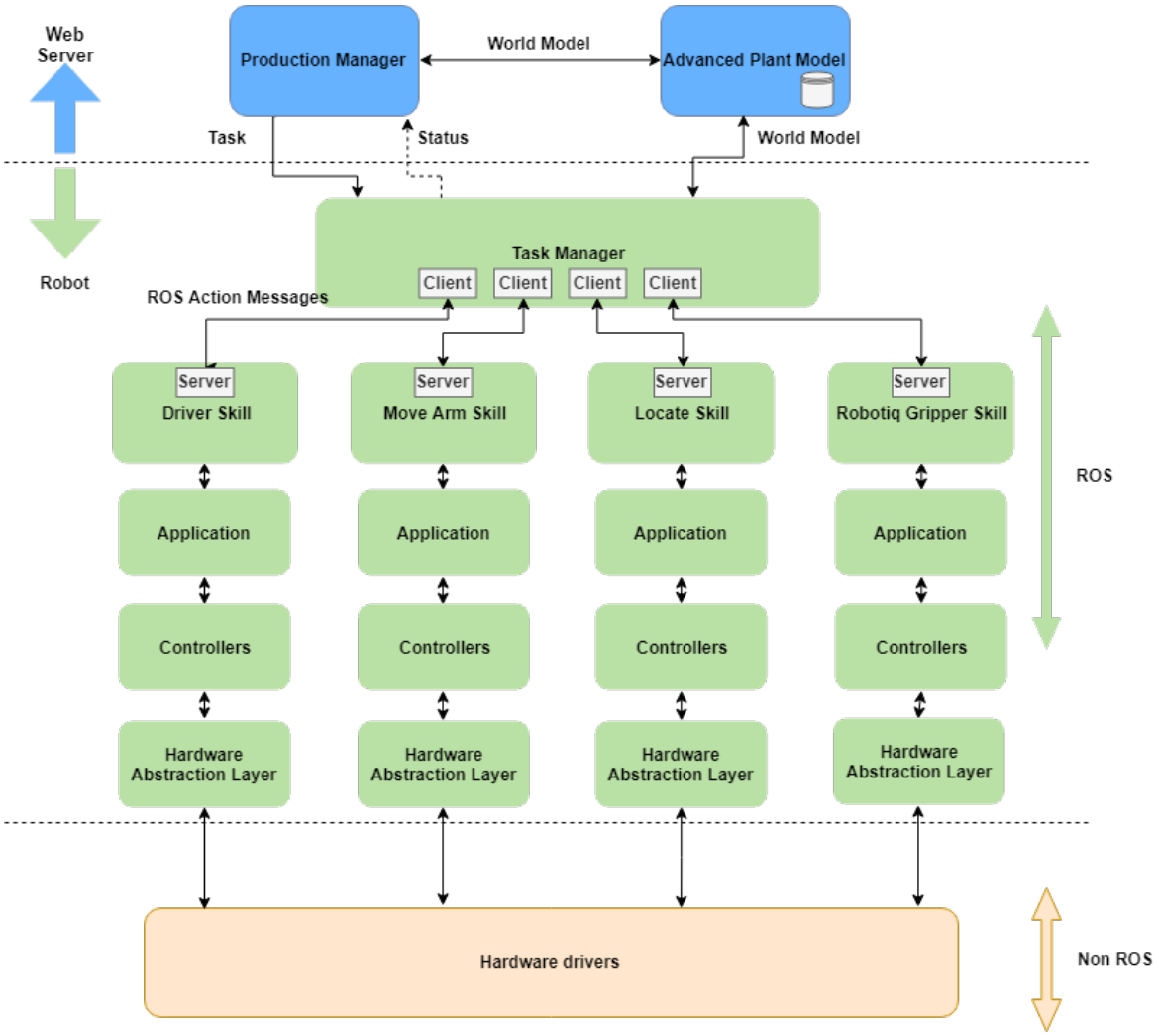

Figure 4.4: High-Level Software Architecture of the FASTEN robot system

This robotic system has a distributed architecture. One part is implemented on a server and the other on the robot itself. On the robot part, there is a division between what is implemented using the ROS framework and the part that is not implemented with ROS. In the server side, there are two components the Production Manager (PM) and the Advanced Plant Model (APM) [37]. On the ROS side of the robot, there are the skills and the Task Manager (TM). Finally, on the non-ROS side of the robot, there are the hardware drivers. This High-Level Software Architecture of this system is presented in Figure 4.4.

The APM keeps a near real-time model of the production environment, which can be seen of Figure 4.5 [37]. This includes location and state of all production resources, parts and materials

that are being used in the current production schedule. All this information becomes very useful to the other software components of the architecture. Also on the web server side, there is the PM. The PM is responsible to manage the production resources of the production environment, control the execution of the production schedules and it is also responsible for monitor the ongoing performance of the different production tasks. To fulfil these responsibilities, the Production Manager uses the data of physical objects and functional elements stored in the APM to allocate resources and execute the tasks.

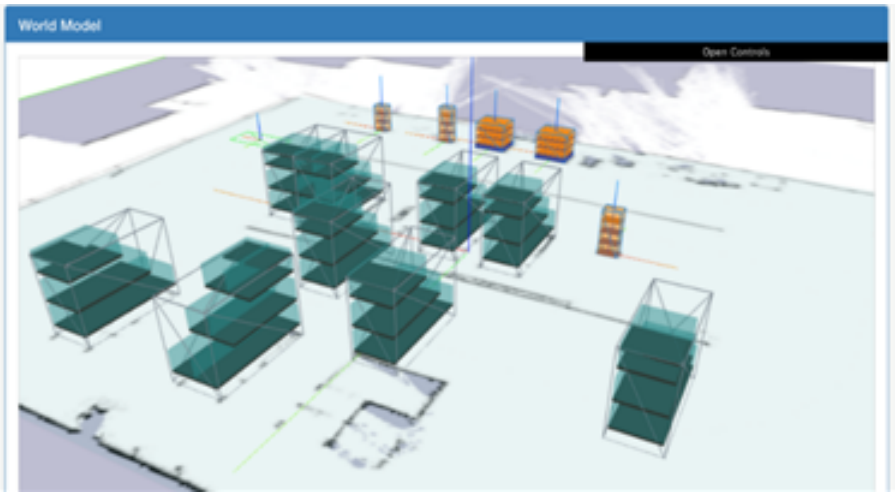

Figure 4.5: World Model kept at APM

On the robot, the most important component is the Task Manager (TM). TM is the main node on the robot. It has two primary functions: provides integration between the robot and other modules on the system, like the APM or the PM, and is responsible for the orchestration of tasks, using the skills of the robot. The tasks to be executed are encoded within State Chart Extensible Markup Language (SCXML) files, which is a state machine notation for control abstraction. SCXML allows interoperability between the TM and the APM supporting dynamic and intuitive task-level robot programming. On the TM there is a ROS Action Client for each skill and on each skill, there is a ROS Action server. This is due to the fact that skills are implemented using ROS Actions. The TM uses skills by defining a goal and sending it to the respective Action server. It can also receive updates from the Action server about the execution of the goal, feedback. The TM also has an option to cancel the execution, if needed. When the execution is completed it receives, from the skill Action server, the result and additional information about the outcome of the performed action.

Still, on the robot and the ROS part, there are the four skills, move arm skill, gripper skill, locate skill and drive skill. The move arm skill is responsible for the movement of the robotic arm. The gripper skill is responsible for the actuation of the gripper. The locate skill is responsible for the recognition and localization of the parts that need to be picked. Finally, the drive skill is responsible for the movement of the robotic platform and to ensure that the movement is collision free. As previously mentioned each of these skills has a ROS Action Server, used to receive a

goal from TM. Each of these skills is organized in three different parts, which are application, controllers and hardware abstraction layer. These three layers allow a goal received from TM to be transmitted to the hardware drivers and then executed.

4.3.1 Architecture Detailed Description

The previously made description purpose was to provide an overview of the architecture, of its concept and its components. This detailed description is only focused on the ROS part of the architecture. It will provide a description of the implemented nodes and topics and their purpose. However, the drive skill will not be detailed, since, at the moment this work is being developed it is not yet fished.

4.3.1.1 Task Manager Detailed Description

As previously stated the Task Manager is responsible for the integration between the ROS side and the web server side, namely the Production Manager and Advanced Plant Model. The implementation of it uses 4 nodes and several topics. *task_manager*,*task_manager_heart_beep* , *task_manager_robot_profile* and *task_manager_robot_map* are the nodes created for this implementation.

task_manager node is the central node of the Task Manager. It is the node that receives the tasks from the PM and then executes those tasks. These tasks are performed by calling the skills servers instantiated on the robot.

Advertised topics:

- */OSPS/TM/AssignTaskListResp* - This topic is used to send to the Production Manager a response, informing it that the TM received the issued list of tasks. It has a message of the type [osps_msgs/TMAssignTaskListResp].
- */OSPS/TM/CancelTaskResp* - This topic is used to send to the PM a message confirming the cancel of a requested task. It has a message of the type [osps_msgs/TMCancelTaskResp].
- */OSPS/TM/ExecuteTaskResp* - This topic is use send to the PM a message confirming that it started the execution of the requested task. It has a message of the type [osps_msgs/TMExecuteTaskResp].
- */OSPS/TM/TaskStatus* - This topic is used to send to the PM a message, providing it feedback about the execution of a task. It has a message of the type [osps_msgs/TMTaskStatus].
- */OSPS/TM/TaskStatusResp* - This topic is used to send feedback about the execution of a task whenever the PM asks for it. It has a message of the type [osps_msgs/TMTaskStatusResp]
- */MoveArmSkill/goal* - This topic is related to a ROS Action client. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionGoal] and is used to send a goal sent to the move arm skill.

- */MoveArmSkill/cancel* - This topic is related to a ROS Action client it has a message of the type [move_arm_skill_msgs/MoveArmSkillActionCancel] and is used to send a message to cancel the goal previously sent.

Subscribed topics:

- */OSPS/PM/AssignTaskListReq* - This topic is used to receive a list of tasks from the PM. It has a message of the type [osps_msgs/PMAssignTaskListReq].
- */OSPS/PM/CancelTaskReq* - This topic is used to receive a order,from the PM, to cancel a task . It has a message of the type [osps_msgs/PMCancelTaskReq].
- */OSPS/PM/ExecuteTaskReq* -This topic is used to receive a order,from the PM, to execute a task from the PM. It has a message of the type [osps_msgs/PMExecuteTaskReq].
- */OSPS/PM/TaskStatusReq* - This topic is used to receive an order, from the PM, to provide feedback about the execution of a given task. It has a message of the type [osps_msgs/PMTaskStatusReq].
- */MoveArmSkill/feedback* - This topic is related to a ROS Action server. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionFeedback] and is used receive periodic informations about the execution of the goal of the move arm skill.
- */MoveArmSkill/result* - This topic is related to a ROS Action server. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionResult] and is used to receive upon completion of the goal a one-time message containing additional information about the move arm skill.
- */MoveArmSkill/status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to receive periodic informations about the execution of goals.

task_manager_heart_beep node is used to periodically send to the APM information about the operational status of the robot. It is a quite simple node, with only two publications and one subscription.

Advertised topics:

- */OSPS/TM/HeartBeep* - This topic is used to send a periodic message to the APM and inform that the Task Manager is running, and issue an update on its current operational status. It has a message of the type [osps_msgs/TMHeartBeep].
- */OSPS/TM/HeartBeepResp* - This topic is used to send a response message to the APM and inform that the Task Manager is running, and issue an update on its current operational status. Unlike the previous topic, this topic is issued only when there has been a previous request (APMHeartBeepReq). It has a message of the type [osps_msgs/TMHeartBeepResp].

Subscribed topics:

- */OSPS/APM/HeartBeepReq* -This topic is used to receive from the APM a request to inform it if the Task Manager is running, and to get information on its current operational status. It has a message of the type [osps_msgs/APMHeartBeepReq].

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

task_manager_robot_profile node is very simple. Its main objective is to communicate with PM to inform about the skills that this robot is capable of executing, and to issue some information on the properties of the robot (such as the robot's name, dimensions, etc).

Advertised topics:

- */OSPS/TM/RobotProfile* - This topic is used to respond to requests from the APM to inform it about the skills of the robot. It has a message of the type [osps_msgs/TMRobotProfile], containing the skills that the robot can execute. Besides, this topic also relays information about the properties of the robot.

Subscribed topics:

- */OSPS/APM/RobotDiscovery* - This topic is used to ask the robotic fleet (or a specific robot) to send information on its properties. It has a message of the type [osps_msgs/APMRobotDiscovery].

task_manager_robot_map node is also very simple, with only one subscription and one publisher. The goal of this node is to communicate with APM providing information to update the world model with the robot's occupation map.

Advertised topics:

- */OSPS/TM/RobotMapResp* - This topic is used to respond to map requests from the APM. It has a message of the type [osps_msgs/TMRobotMapResp] and is used to send the 2D occupation map of the environment.

Subscribed topics:

- */OSPS/APM/RobotMapReq* - This topic is used to receive requests from APM. It has a message of the type [osps_msgs/APMRobotMapReq] and is used to require the 2D occupation map of the environment.

4.3.1.2 Move Arm Skill Detailed Description

As previously stated the move arm skill is responsible for the movement of the robotic arm. The implementation of this skill uses 3 nodes and several topics. *move_arm_skill_server*, *ur_driver* and *arm_action_controller* are the nodes created for the functioning of this skill. *move_arm_skill_server* and *arm_action_controller* nodes, use ROS Actions for communication. This means that these nodes will publish/subscribe to 5 different topics related to ROS Actions. Those topics are the goal, cancel, status, feedback and result topic.

move_arm_skill_server node receives goals from the Task Manager and starts their execution. This node is a ROS Action server, which communicates with the client located on the Task Manager (TM). *move_arm_skill_server* node is not only a ROS Action server but is also a ROS Action client for other Actions, which servers are located on *arm_action_controller* node. Those other actions are MoveJ, MoveL, MoveJJoints and MoveLJoints.

Following this detailed description of the node, it will be described the topics advertised by this node, i.e. topics where this node publishes, and subscribed topics. The type of messages on those topics and its objectives will also be detailed.

Advertised topics:

- */MoveArmSkill/feedback* - This topic is related to a ROS Action server. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionFeedback] and is used to send periodic informations about the execution of the goal.

- */MoveArmSkill/result* - This topic is related to a ROS Action server. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionResult] and is used to send upon completion of the goal a one-time message containing additional information.

- */MoveArmSkill/status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */MoveJ/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and can cancel the execution of the goal.

- */MoveJ/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJActionGoal] and is used to send a goal in Cartesian Space.

- */MoveJJoints/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and can cancel the execution of the goal.

- */MoveJJoints/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJJointsActionGoal] and is used to send a goal in Joint Space.

- */MoveL/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and can cancel the execution of the goal.

- */MoveL/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLActionGoal] and is used to send a goal in Cartesian Space.
- */MoveLJoints/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and can cancel the execution of the goal.
- */MoveLJoints/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLJointsActionGoal] and is used to send a goal in Join Space.

Subscribed topics:

- */MoveArmSkill/goal* - This topic is related to a ROS Action Client. It has a message of the type [move_arm_skill_msgs/MoveArmSkillActionGoal] and is used to receive a goal sent by TM with a goal to be completed by the skill.
- */MoveArmSkill/cancel* - This topic is related to a ROS Action client it has a message of the type [move_arm_skill_msgs/MoveArmSkillActionCancel] and is used to receive a message to cancel the goal previously indicated by the TM.
- */MoveJ/feedback* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveJActionFeedback] and is used to receive periodic informations about the execution of the goal.
- */MoveJ/result* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJActionResult] and is used to receive a one time message upon the completion of the goal with additional information.
- */MoveJ/status* - This topic is related to a ROS Action clien,t it has a message of the type [actionlib_msgs/GoalStatusArray] and is used to receive periodic informations about the execution of goals.
- */MoveJJoints/feedback* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJJointsActionFeedback] and is used to receive periodic informations about the execution of the goal.
- */MoveJJoints/result* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJJointsActionResult] and is used to receive a one time message upon the completion of the goal with additional information.
- */MoveJJoints/status* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJJointsActionStatus] and is used to receive periodic informations about the execution of goals.
- */MoveL/feedback* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLActionFeedback] and is used to receive periodic informations about the execution of the goal.

- */MoveL/result* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLActionResult] and is used to receive a one time message upon the completion of the goal with additional information.

- */MoveL/status* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to receive periodic informations about the execution of goals.

- */MoveLJoints/feedback* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLJointsActionFeedback] and is used to receive periodic informations about the execution of the goal.

- */MoveLJoints/result* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLJointsActionResult] and is used to receive a one time message upon the completion of the goal with additional information.

- */MoveLJoints/status* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to receive periodic informations about the execution of goals.

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

arm_action_controller node has 4 ROS Action Servers MoveJ, MoveL, MoveJJoints and MoveLJoints. The actions are related to two different ways of planning the trajectory of the robotic arm, with Joint Space or Cartesian Space. MoveJ and MoveL are connected with Cartesian space movement planning and currently represent about 10% of the robotic arm movement. MoveJJoints and MoveLJoints, on the other hand, are connected with Joint space movement planning and currently represent 90% of the robotic arm movement.

Advertised topics:

- */MoveJ/feedback* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveJActionFeedback] and is used to send periodic informations about the execution of the goal.

- */MoveJ/result* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveJActionResult] and is used to send a one time message upon the completion of the goal with additional information.

- */MoveJ/status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */MoveJJoints/feedback* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveJJointsActionFeedback] and is used to send periodic informations about the execution of the goal.

- */MoveJJoints/result* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveJJointsActionResult] and is used to send a one time message upon the completion of the goal with additional information.

- */MoveJJoints/status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */MoveL/feedback* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveLActionFeedback] and is used to send periodic informations about the execution of the goal.

- */MoveL/result* - This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveLActionResult] and is used to send a one time message upon the completion of the goal with additional information.

- */MoveL/status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */MoveLJoints/feedback*- This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveLJointsActionFeedback] and is used to send periodic informations about the execution of the goal.

- */MoveLJoints/result*- This topic is related to a ROS Action server. It has a message of the type [arm_action_controller/MoveLJointsActionResult] and is used to send a one time message upon the completion of the goal with additional information.

- */MoveLJoints/status* - This topic is related to a ROS Action server. It has a message of the type[actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */ur_driver/URScript* - This topic is used to send messages of the type [std_msgs/String]. This is intended to send moveL and moveJ commands directly to the robotic arm.

Subscribed topics:

- */MoveJ/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and is used to receive a message to cancel the goal previously indicated by the client.
- */MoveJ/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJActionGoal] and is used to receive a goal in Cartesian Space.
- */MoveJJoints/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and is used to receive a message to cancel the goal previously indicated by the client.
- */MoveJJoints/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveJJointsActionGoal] and is used to send a goal in Joint Space.
- */MoveL/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and is used to receive a message to cancel the goal previously indicated by the client.
- */MoveL/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLActionGoal] and is used to receive a goal in Cartesian Space.
- */MoveLJoints/cancel* - This topic is related to a ROS Action client. It has a message of the type [actionlib_msgs/GoalID] and is used to receive a message to cancel the goal previously indicated by the client.
- */MoveLJoints/goal* - This topic is related to a ROS Action client. It has a message of the type [arm_action_controller/MoveLJointsActionGoal] and is used to send a goal in Joint Space.
- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.
- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.
- */wrench* - This topic is used to receive the force on the robotic arm. It has a message of the type [geometry_msgs/WrenchStamped].
- */joint_states* - This topic is used to receive the state of every joint on the robotic arm. It has a message of the type [sensor_msgs/JointState].

ur_driver is the node that implements the Hardware Abstraction Layer (HAL). This is used to provide the interface between ROS and the robotic arm drivers. It also provides useful information about the position of the robotic arm, essential for its control.

Advertised topics:

- */io_states* - This topic is used to share information about the state of the IO of the robotic arm. It has a message of the type [ur_msgs/IOStates].

- */joint_states* - This topic is used to share the state of every joint on the robotic arm. It has a message of the type [sensor_msgs/JointState].

- */ur_driver/parameter_descriptions* - This topic is used to get the parameters configurations of the node. It has a message of the type [dynamic_reconfigure/ConfigDescription].

- */ur_driver/parameter_updates* -This topic is used to dynamical reconfigure a certain parameter of the node. It has a message of the type [dynamic_reconfigure/Config].

- */wrench* - This topic is used to publish TCP force on the robotic arm. It has a message of the type [geometry_msgs/WrenchStamped].

- */ur_driver/tool_velocity* -

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

Subscribed topics:

- */ur_driver/URScript* - This topic is used to receive messages of the type [std_msgs/String]. This is intended to receive moveL and moveJ commands and directly transmit them to the robotic arm.

4.3.1.3 Robotiq Gripper Skill Detailed Description

As previously stated this skill is only responsible for the actuation of the gripper. That single responsibility makes this the most simple skill of this robot. It has only two nodes *robotiq_gripper_skill_server* and *robotiqCModel*.

robotiq_gripper_skill_server node is a ROS Action server, which communicates with the client located on Task Manager. It receives tasks from TM and ensures their execution. This node also communicates with *robotiqCModel* node, the HAL of this skill.

Advertised topics:

- */RobotiqGripperSkill/Feedback* - This topic is related to a ROS Action server. It has a message of the type [robotiq_gripper_skill_msgs/RobotiqGripperSkillActionFeedback] and is used to send periodic informations about the execution of the goal.

- */RobotiqGripperSkill/Result* - This topic is related to a ROS Action server. It has a message of the type [robotiq_gripper_skill_msgs/RobotiqGripperSkillActionResult] and is used to send upon completion of the goal a one-time message containing additional information.

- */RobotiqGripperSkill/Status* - This topic is related to a ROS Action server. It has a message of the type [actionlib_msgs/GoalStatusArray] and is used to send periodic informations about the execution of goals.

- */CModelRobotOutput* - This topic is used to send actuation commands to *robotiqCModel*, which provides the HAL of this skill. It has a message of the type [robotiq_c_model_control/CModel_robot_

Subscribed topics:

- */RobotiqGripperSkill/Goal* - This topic is related to a ROS Action client. It has a message of the type [robotiq_gripper_skill_msgs/RobotiqGripperSkillActionGoal] and is used to receive a goal sent by TM with a goal to be completed by the skill.

- */RobotiqGripperSkill/Cancel* - This topic is related to a ROS Action client. It has a message of the type [robotiq_gripper_skill_msgs/RobotiqGripperSkillActionCancel] and is used to receive a message to cancel the goal previously indicated by TM.

- */CModelRobotInput* - This topic is used to receive information about the present state of the gripper. It has a message of the type [robotiq_c_model_control/CModel_robot_input].

/robotiqCModel node provides the HAL of the gripper skill. It receives a command from *robotiq_gripper_skill_server* node witch then communicates to the gripper via Modbus.

Advertised topics:

- */CModelRobotInput* - This topic provides this node information about the present state of the gripper. It has a message of the type [robotiq_c_model_control/CModel_robot_input].

Subscribed topics:

- */CModelRobotOutput* - This topic is used to receive actuation commands from *robotiq_gripper_skill_server*. It has a message of the type [robotiq_c_model_control/CModel_robot_output].

4.3.1.4 Locate Skill Detailed Description

The locate skill is one of the most complex of the robot. As previously stated it is responsible for the 3D capturing and the recognition and localization of the parts and their orientation. This skill provides the capability of localization with 3 or 6 degrees of freedom, using the Point Cloud Library (PCL) to achieve that. The PCL is an open source project that provides 2D, 3D and point cloud processing tools.

Since this skill uses resources that are highly demanding on the CPU and memory of the robot some of its features are only used on demand to avoid wasting resources.

object_recognition_skill_server node acts as the wrapper of the skill. It is responsible for receiving the goals from the TM and guarantee their execution. That execution is performed by using the capabilities of the remaining nodes of this skill.

Advertised topics:

- *object_recognition_skill_server/feedback* - This topic is related to a ROS Action Server. It has a message of the type [ObjectRecognitionActionFeedback] and is used to send periodic informations about the execution of the goal.
- *object_recognition_skill_server/result* - This topic is related to a ROS Action Server. It has a message of the type [object_recognition_msgs/ObjectRecognitionActionResult] and is used to send upon completion of the goal a one-time message containing additional information.
- *object_recognition_skill_server/status* - [ObjectRecognitionActionStatus]

Subscribed topics:

- *object_recognition_skill_server/goal* - This topic is related to a ROS Client Server. It has a message of the type [object_recognition_msgs/ObjectRecognitionActionGoal] and is used to receive the goal sent by the TM.
- *object_recognition_skill_server/cancel* - This topic is related to a ROS Client Server. It has a message of the type [ObjectRecognitionActionCancel] and is used to receive a message to cancel the goal previously indicated by the TM.

dynamic_robot_localization/drl_localization_node is responsible for the recognition and localization of the parts that will be picked. It uses PCL tools to perform the localization with 3 or 6 degrees fo freedom.

Advertised topics:

- */dynamic_robot_localization/aligned_pointcloud* - This topic is used to publish a pointcloud coming from the *ambient_pointcloud* topic after applying the registration correction. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/aligned_pointcloud_inliers* - This topic is used to publish registered inliers. If empty, messages will not be published. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/aligned_pointcloud_outliers* - This topic is used to register the pointcloud outliers. If empty, messages will not be published. This is used by OctoMap, occupancy map. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/ambient_keypoints* - This topic is the one where the detected keypoints will be published. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/ambient_pointcloud_filtered* - This topic is used to publish the ambient pointcloud after the application of the filters. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/diagnostics* - This topic is used to provide information about: the number of points, the keypoints in the reference and the ambient cloud before and after filtering. It has a message of the type [dynamic_robot_localization/LocalizationDiagnostics].

- */dynamic_robot_localization/filtered_pointcloud* - This topic is used to publish the ambient pointcloud after the application of the filters and the circular buffer. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/initial_pose_with_covariance* - This topic is only used when the tacking state is reset. The message of this topic is only published after the initial pose has been estimated. It has a message of the type [geometry_msgs/PoseWithCovarianceStamped].
- */dynamic_robot_localization/localization_detailed* - This topic is used to provide information about the current pose, which was computed by the localization system. It has a message of the type [dynamic_robot_localization/LocalizationDetailed].
- */dynamic_robot_localization/localization_initial_pose_estimations* - This topic is used to publish a array with initial pose estimations. When performing tracking, it will have 0 poses. If the tacking is lost, and the initial pose estimation using features succeeds, it will have the accepted poses of the last initial pose estimation. It has a message of the type [geometry_msgs/PoseArray].
- */dynamic_robot_localization/localization_pose* - This topic is used to publish poses, which are useful to interact with other packages or to visualize in *rviz*. It has a message of the type [geometry_msgs/PoseStamped].
- */dynamic_robot_localization/localization_pose_with_covariance* - This topic is used to publish poses with covariance, which are useful to interact with other packages, namely amcl package (a localization package). It has a message of the type [geometry_msgs/PoseWithCovarianceStamped
- */dynamic_robot_localization/localization_times* - This topic is used to provide information about the wall clock time (in milliseconds) of the main localization steps, and also the global time. It has a message of the type [dynamic_robot_localization/LocalizationTimes].
- */dynamic_robot_localization/reference_keypoints* - This topic is used to provide information about the reference keypoints. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/reference_pointcloud* - This topic is used to provide information about the reference cloud currently being used by the localization system. It has a message of the type [sensor_msgs/PointCloud2].
- */dynamic_robot_localization/reference_pointcloud_keypoints* - This topic is used to provide information about the reference pointcloud keypoints. It has a message of the type [sensor_msgs/PointCloud2].
- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

Subscribed topics:

- */map* - This topic is used to receive a down-projected 2D occupancy map from the 3D map. It has a message of the type [nav_msgs/OccupancyGrid].

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

dynamic_robot_localization/laserscan_to_pointcloud_assembler node is capable of assembling a pointcloud from a series of laser scans using spherical linear interpolation.

Advertised topics:

- */dynamic_robot_localization/ambient_pointcloud* - This topic is used to provide a pointcound, which points come from a sensing device. This information is then used to update the localization pose. It has a message of the type [sensor_msgs/PointCloud2].

- */dynamic_robot_localization/laserscan_to_pointcloud_assembler/parameter_descriptions* - This topic is used to get the parameters configurations of the node. It has a message of the type [dynamic_reconfigure/ConfigDescription].

- */dynamic_robot_localization/laserscan_to_pointcloud_assembler/parameter_updates* -This topic is used to dynamical reconfigure a certain parameter of the node. It has a message of the type [dynamic_reconfigure/Config].

Subscribed topics:

- */tilt_scan* - This topic is used to receive a laser scan from a Light Detection And Ranging (LIDAR) sensor mounted on a platform with tilt. It has a message of the type [sensor_msgs/LaserScan].

- */base_scan* - This topic is used to receive a laser scan from a LIDAR sensor mounted on the base of the robot. It has a message of the type [sensor_msgs/LaserScan].

- */dynamic_robot_localization/odom* - This topic is used to receive odometry information from the encoders of the wheels of the robot. It has a message of the type [sensor_msgs/LaserScan].

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very

important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

dynamic_robot_localization/pose_to_tf_publisher_node is used to extract information from poses and publish it as 3D coordinate frames. It is also capable of republishing or remapping a given 3D coordinate frame into another. This republishing or remapping can add a time offset to the frame which can be useful to synchronize ground truth poses.

Advertised topics:

- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.

Subscribed topics:

- */dynamic_robot_localization/localization_pose* - This topic is used to receive information about poses, which are useful to interact with other packages or to visualize in *rviz*. It has a message of the type [geometry_msgs/PoseStamped].
- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.
- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

octomap_server node is used to build and distribute 3D occupancy maps. These maps can be static or build form the received data in the form of a pointcloud. The information for the topics of this node is only published if those topics have subscribers.

Advertised topics:

- */free_cells_vis_array* - This topic is used to provide information about which voxels (value of a 3D grid) are free. It has a message of the type [visualization_msgs/MarkerArray].
- */map* - This topic is used to provide a down-projected 2D occupancy map from the 3D map. It has a message of the type [nav_msgs/OccupancyGrid].
- */occupied_cells_vis_array* - This topic is used to provide information about which voxels (value of a 3D grid) are occupied. It has a message of the type [visualization_msgs/MarkerArray].
- */octomap_binary* - This topic provides the complete maximum-likelihood occupancy map. The information in this topic is binary and indicates if space is free or occupied. It has a message of the type [octomap_msgs/Octomap].

- */octomap_full* - This topic provides the complete maximum-likelihood occupancy map. This topic has information about the complete probabilities and other additional information stored in the tree. It has a message of the type [octomap_msgs/Octomap].
- */reference_pointcloud_update* - This topic is used to publish actualized versions of the point-cloud as new information is being received. It has a message of the type [sensor_msgs/PointCloud2].
- */octomap_server/parameter_descriptions* - This topic is used to get the parameters configurations of the node. It has a message of the type [dynamic_reconfigure/ConfigDescription].
- */octomap_server/parameter_updates* - This topic is used to dynamical reconfigure a certain parameter of the node. It has a message of the type [dynamic_reconfigure/Config].

Subscribed topics:

- */aligned_pointcloud_outliers* - This topic is used to receive information about the registered outliers. It has a message of the [sensor_msgs/PointCloud2].
- */initial_2d_map* - This topic is used to receive the initial 2D occupancy grid map to initialize the octomap representation. It has a message of the type [nav_msgs/OccupancyGrid].
- */tf* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system.
- */tf_static* - This topic is used to keep track and inform about the coordinate frames of every component of the system. It has a message of the type [tf2_msgs/TFMessage] and is very important to the safe movement of every component in the system. It is similar to */tf* with the difference that this topic keeps static transformations.

phoxi_camera node contains drivers for *PhoXi* devices. This means that this node provides the ROS interface for those devices. The information obtained by the camera is available in several topics, each of them with information in different formats. The topics made available by this topic are the following.

Advertised topics:

- */diagnostics* - [diagnostic_msgs/DiagnosticArray]
- */phoxi_camera/camera_info* - This node provides information about the camera. It has message of the type [sensor_msgs/CameraInfo].
- */phoxi_camera/confidence_map* - This node provides a confidence map of the captured image. It has a message of the type [sensor_msgs/Image].
- */phoxi_camera/depth_map* - This node provides a depth map of the captured image. It has a message of the type [sensor_msgs/Image].

- */phoxi_camera/image_raw* - This node provides the original captured image. It has a message of the type [sensor_msgs/Image].
- */phoxi_camera/normal_map* - This node provides a normal map of the captured image. It has a message of the type [sensor_msgs/Image].
- */phoxi_camera/parameter_descriptions* - [dynamic_reconfigure/ConfigDescription]
- */phoxi_camera/parameter_updates* - [dynamic_reconfigure/Config]
- */phoxi_camera/pointcloud* - This node provides a pointcloud of the environment, obtained with a structured light. It has a message of the type [sensor_msgs/Image].
- */phoxi_camera/texture* - This topic is used to publish the grayscale image that was captured by the camera. It has a message of the type [sensor_msgs/Image].

4.4 Coding Standards

Presently at CRIIS, there is not any specific coding standard for C++ being enforced. However, from the observation of the source code, it is possible to identify some style similarity with Google C++ style. Not having an enforced code standard means that different developers could follow different styles, thus creating variations on the source code. This could also compromise the readability of the code, is also not good for maintainability, and can cause dangerous code due to the use of features which are disapproved by code standards. Therefore, for these reasons a coding standard should be enforced.

Chapter 5

Case Study Analysis and Improvements

This chapter comprises the analysis of the source code and the launch files of the FASTEN project. The proposed methodologies and tools used to perform the analysis are also presented on this chapter.

The FASTEN project, which was describe in chapter 4 uses several ROS packages. Some were developed by the ROS community, while others are being developed by CRIIS. In this analysis, 22 ROS packages were analysed and being directly used on the mobile manipulator. From those packages, 14 contained C++ source code, while the remaining 8 emcompassed Python source code or only launch files. The C++ packages contained approximately 200,000 lines of code.

5.1 Methodologies and tools

As stated in chapter 1, this work was developed in collaboration with the SAFER [1] project. For this reason, and due to its unique features that were describe in chapter 3, the HAROS framework was the chosen tool. It was used to perform the static analysis and the model extraction for the architectural analysis.

The methodology adopted for this work was a iterative one. The reason for this choice is that it is very difficult to solve all the issues of a software in one run. It is also not recommended to do it in a single run, since changes can cause new issues that would be hard to trace the origin of, something that would not happen if a more methodical approach was taken.

For the source code, an initial analysis of the code was performed. The issues that were found were listed on tables for each analysed package. In this tables, the issues were grouped by its type and their occurrences registered. To each issue was then attributed a category related to its cause. Each issue was also evaluated to determinate its severity and the effort necessary to solve it. All of these classifications and evaluations were detailed and are available in this chapter.

Post this, and following the proposed methodology, 3 iterations were performed. To assess which issue would be tackled first was created a score, based on the weighted sum of the number of

[1] https://www.inesctec.pt/pt/projetos/safer

occurrences, severity and effort to solve. Before each new iteration, some of the issues with higher score found in the previous analysis were tackled. This process was then repeated iteratively.

For the architectural analysis, the idea is similar, since it also follows a iterative methodology. In the initial analysis, the extracted model is compared to the model previously described in chapter 4 to ensure its similarity. If the architecture is not similar, the causes for that will be traced, and if possible, fixed. Then, this process will be repeated iteratively until a similar model is achieved.

5.2 Initial Analysis

This initial analysis contains the raw data collected the HAROS framework. Here, the issues were divided into 3 categories: Formatting, Code Standards, and Metrics. The first category - Formatting - encloses issues related to indention, whitespaces and placement of braces. The second - Code Standards - encloses issues related to code standards, such as casting or namespace using-directives. Finally, the third - Metrics - encloses issues relate to metrics, such as cyclomatic complexity or the maintainability index. These results are displayed on tables.

Some different issues were grouped into one issue according to similarity, also easing presentation. As an example, issues of placement of the opening curly brace and issue of placement of closing curly brace were grouped as *Opening/Closing Curly Brace*.

Since it was impossible and impractical to solve every issue with one run, it was necessary to determinate which issues would be tackled first. To determinate the priority of issues and define which ones were the first to be tackled, the equation 5.1 was created. This equation attributes a score to each issue within a package. The score is a weighted sum of the number of issues - *Num*, the Severity of the issue - *S*, and the Effort to Solve - *E*. Each of these variables had their respective coefficient. K_1 and K_3 were attributed the coefficient 1. To K_2, that represented the Severity, the coefficient 10 was attributed. A highest coefficient weight was given to Severity, so it could have a more significant impact on the issue's global score.

$$Score = K_1 \cdot Num + K_2 \cdot S + K_3 \cdot E; \tag{5.1}$$

Table 5.1: Move arm skill package analysis

Package - move_arm_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	103	Formatting	1	1	114
Opening/Closing Curly Brace	93	Formatting	1	1	104
Integer Types	22	Code Standard	2	2	44
Function Length	8	Metric	2	3	31
Casting	8	Code Standard	3	1	39
Include	5	Code Standard	2	2	27
Include Order	4	Code Standard	1	1	15
Cyclomatic Complexity	4	Metric	3	3	37
No Copyright Statement	3	Code Standard	1	3	16
Maintainability Index	3	Metric	3	1	34
Indent Access Modifiers	3	Formatting	1	3	16
File Length	1	Metric	3	3	34
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	1	Metric	3	3	34
Blank Lines In Code Blocks	1	Formatting	1	1	12
No Header Guard	1	Code Standard	2	1	22
Non-const Reference Parameters	1	Code Standard	1	2	13
Total issues	**262**		**Total Score**		**626**

Table 5.2: Arm action controller package analysis

Package - arm_action_controller					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	59	Formatting	1	1	70
Line Length	59	Formatting	1	1	70
Include Order	20	Code Standard	1	1	31
Integer Types	12	Code Standard	2	2	34
Non-const Reference Parameters	5	Code Standard	1	2	17
Include	3	Code Standard	2	1	24
No Copyright Statement	3	Code Standard	1	1	14
Cyclomatic Complexity	3	Metric	3	3	36
Function Length	3	Metric	2	3	26
Function Parameters	2	Metric	3	3	35
Maintainability Index	2	Metric	3	3	35
Whitespace Before Comments	2	Formatting	1	1	13
Indent Access Modifiers	2	Formatting	1	1	13
Newline at End of File	2	Formatting	1	1	13
End of Namespace Comment	2	Formatting	1	1	13
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Casting	1	Code Standard	3	1	32
Total issues	**183**		**Total Score**		**566**

Table 5.3: Arm interface package analysis

Package - arm_interface					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	253	Formatting	1	1	264
Opening/Closing Curly Brace	164	Formatting	1	1	175
Include Order	31	Code Standard	1	1	42
Integer Types	21	Code Standard	2	2	43
Include	20	Code Standard	2	1	41
Non-const Reference Parameters	15	Code Standard	1	2	27
No Copyright Statement	13	Code Standard	1	1	24
Indent Access Modifiers	13	Formatting	1	1	24
End of Namespace Comment	13	Formatting	1	1	24
Casting	11	Code Standard	3	1	42
Whitespace Before Comments	10	Formatting	1	1	21
Halstead Bugs	6	Metric	3	3	39
Function Length	6	Metric	2	3	29
Whitespace Before Comment Text	6	Formatting	1	1	17
No Header Guard	6	Code Standard	2	1	27
Halstead Volume	5	Metric	3	3	38
Maintainability Index	5	Metric	3	3	38
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Blank Lines In Code Blocks	3	Code Standard	1	1	14
Do Not Include Twice	1	Code Standard	1	1	12
Single Else-If Else Line	1	Formatting	1	1	12
Order of Evaluation	1	Code Standard	3	1	32
Avoid Thread-Unsafe Functions	1	Code Standard	3	2	33
Avoid Rvalue References	1	Code Standard	3	2	33
Complex Multi-line Comments and Strings	1	Formatting	1	1	12
Total issues	**611**		**Total Score**		**1126**

Table 5.4: Ur modern driver package analysis

Package - ur_modern_driver					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	718	Formatting	1	1	729
Line Length	294	Formatting	1	1	305
Non-const Reference Parameters	194	Code Standard	1	2	206
Indent Access Modifiers	115	Formatting	1	1	126
Integer Types	44	Code Standard	2	2	66
Include	40	Code Standard	2	1	61
No Header Guard	35	Code Standard	2	1	56
Make Constructors Explicit	22	Code Standard	1	1	33
Newline at End of File	19	Formatting	1	1	30
Function Length	17	Metric	2	3	40
Avoid Unapproved Headers	16	Code Standard	1	3	29
Avoid C System Headers	13	Code Standard	2	1	34
Do Not Use C Types	13	Code Standard	2	2	35
Casting	12	Code Standard	3	1	43
Halstead Bugs	11	Metric	3	3	44
Halstead Volume	8	Metric	3	3	41
Avoid Rvalue References	8	Code Standard	3	2	40
Avoid Namespace Using-Directives	6	Code Standard	3	2	38
Maximum Executable Lines of Code	4	Metric	2	3	27
Maintainability Index	4	Metric	3	3	37
Avoid Unapproved Classes and Functions	3	Code Standard	1	3	16
Redundant Empty Statement	3	Formatting	1	1	14
Unused Variables	2	Code Standard	2	1	23
File Length	2	Metric	3	3	35
Header Guard Format	2	Code Standard	1	1	13
No Copyright Statement	2	Code Standard	1	1	13
TODO Comment Format	2	Code Standard	1	1	13
Order of Evaluation	2	Code Standard	3	1	33
Do Not Include Twice	1	Code Standard	1	1	12
Include Directory in Header	1	Code Standard	2	1	22
Header Guard Must Close	1	Code Standard	2	1	22
Whitespace Before Comments	1	Formatting	1	1	12
Whitespace Before Comment Text	1	Formatting	1	1	12
Do Not Use Default Lambda Captures	1	Code Standard	3	2	33
C Standard Library	1	Code Standard	2	1	22
Whitespace Around Unary Operator	1	Formatting	1	1	12
Total issues	**1619**		**Total Score**		**2327**

Table 5.5: Robotiq Ethercat package analysis

Package - robotiq_ethercat					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	40	Formatting	1	1	51
Integer Types	10	Code Standard	2	2	32
Line Length	10	Formatting	1	1	21
Include Order	9	Code Standard	1	1	20
Whitespace at the End of Line	9	Formatting	1	1	20
Whitespace After Comma	5	Formatting	1	1	16
Parenthesis & Whitespace	5	Formatting	1	1	16
Indent Access Modifiers	3	Formatting	1	1	14
Whitespace Before Comments	3	Formatting	1	1	14
Function Length	3	Metric	2	3	26
Avoid C System Headers	2	Code Standard	2	1	23
No Copyright Statement	2	Code Standard	1	1	13
Non-const Reference Parameters	2	Code Standard	1	2	14
Include	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Header Guard Format	1	Formatting	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
End of Namespace Comment	1	Formatting	1	1	12
Make Constructors Explicit	1	Code Standard	1	1	12
Storage Class Before Type	1	Code Standard	1	1	12
No Redundant Variables	1	Code Standard	2	1	22
Smallest Feasible Scope	1	Code Standard	1	2	13
Total issues	**112**		**Total Score**		**441**

Table 5.6: Robotiq c model control package analysis

Package - robotiq_c_model_control					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	14	Formatting	1	1	25
Whitespace at the End of Line	9	Formatting	1	1	20
Integer Types	6	Code Standard	2	2	28
Line Length	5	Formatting	1	1	16
Include Order	4	Code Standard	1	1	15
No Copyright Statement	3	Code Standard	1	1	14
Whitespace Before Comments	3	Formatting	1	1	14
End of Namespace Comment	2	Formatting	1	1	13
Non-const Reference Parameters	2	Code Standard	1	2	14
Indent Access Modifiers	2	Formatting	1	1	13
Include	1	Code Standard	2	1	22
Parenthesis & Whitespace	1	Formatting	1	1	12
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Newline at End of File	1	Formatting	1	1	12
No Namespace Indentation	1	Formatting	1	1	12
Total issues	**56**		**Total Score**		**264**

Table 5.7: Dynamic robot localization package analysis

Package - dynamic_robot_localization					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	8070	Formatting	1	1	8081
Line Length	5228	Formatting	1	1	5239
Opening/Closing Curly Brace	1362	Formatting	1	1	1373
Include Order	637	Code Standard	1	1	648
Non-const Reference Parameters	573	Code Standard	1	2	585
No Copyright Statement	207	Code Standard	1	1	218
Parenthesis & Whitespace	131	Formatting	1	1	142
Include	112	Code Standard	2	1	133
No Header Guard	73	Code Standard	2	1	94
Whitespace at the End of Line	57	Formatting	1	1	68
Cyclomatic Complexity	52	Metric	3	3	85
Whitespace Before Comments	46	Formatting	1	1	57
Function Length	32	Metric	2	3	55
Whitespace Before Comment Text	24	Formatting	1	1	35
Whitespace After Comma	21	Formatting	1	1	32
Blank Lines In Code Blocks	15	Formatting	1	1	26
Indent Access Modifiers	13	Formatting	1	1	24
Make Constructors Explicit	13	Code Standard	1	1	24
Casting	12	Code Standard	3	1	43
Integer Types	8	Code Standard	2	2	30
End of Namespace Comment	6	Formatting	1	1	17
Header Guard Format	6	Code Standard	1	1	17
Unused Variables	5	Code Standard	2	1	26
Halstead Bugs	4	Metric	3	3	37
Do Not Use C Types	4	Code Standard	2	1	25
Function Parameters	4	Metric	2	3	27
Halstead Volume	3	Metric	3	3	36
Header Guard Must Close	3	Code Standard	2	1	24
Complex Multi-line Comments and Strings	3	Formatting	1	1	14
File Length	3	Metric	3	3	36
One Command Per Line	3	Formatting	1	1	14
Avoid Thread-Unsafe Functions	3	Code Standard	3	2	35
No Namespace Indentation	2	Formatting	1	1	13
Order of Evaluation	2	Code Standard	3	1	33
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
TODO Comment Format	1	Formatting	1	1	12
Redundant Empty Statement	1	Formatting	1	1	12
Whitespace Around Assignment	1	Formatting	1	1	12
Total issues	**16742**		**Total Score**		**17438**

Table 5.8: Phoxi camera package analysis

Package - phoxi_camera					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	2398	Formatting	1	1	2409
Opening/Closing Curly Brace	717	Formatting	1	1	728
Line Length	298	Formatting	1	1	309
Whitespace After Comma	112	Formatting	1	1	123
Complex Multi-line Comments and Strings	107	Formatting	1	1	118
Integer Types	93	Code Standard	2	2	115
Parenthesis & Whitespace	75	Formatting	1	1	86
Non-const Reference Parameters	46	Code Standard	1	2	58
Whitespace Before Comment Text	38	Formatting	1	1	49
Casting	35	Code Standard	3	1	66
Include Order	34	Code Standard	1	1	45
Indent Access Modifiers	24	Formatting	1	1	35
Make Constructors Explicit	23	Code Standard	1	1	34
Function Length	20	Metric	2	3	43
Whitespace Before Comments	20	Formatting	1	1	31
Cyclomatic Complexity	17	Metric	3	3	50
Avoid C System Headers	13	Code Standard	2	1	34
Blank Lines In Code Blocks	13	Formatting	1	1	24
Whitespace at the End of Line	13	Formatting	1	1	24
No Copyright Statement	12	Code Standard	1	1	23
No Uninitialized Member Variables	12	Code Standard	2	1	33
Header Guard Format	10	Code Standard	1	1	21
Include	10	Code Standard	2	1	31
Alternative Tokens	6	Code Standard	1	1	17
Halstead Bugs	6	Metric	3	3	39
Halstead Volume	5	Metric	3	3	38
Header Guard Must Close	5	Code Standard	2	1	26
No Unions	5	Code Standard	3	2	37
Order of Evaluation	5	Code Standard	3	1	36
Do Not Use C Types	4	Code Standard	2	1	25
Whitespace Around Assignment	4	Formatting	1	1	15
C-style String Constants	4	Code Standard	1	1	15
One Command Per Line	4	Formatting	1	1	15
Blank Lines Before Section	3	Formatting	1	1	14
Unused Variables	3	Code Standard	2	1	24

Whitespace Around Binary Operator	3	Formatting	1	1	14
Single Else-If Else Line	3	Formatting	1	1	14
File Length	2	Metric	3	3	35
Maximum Executable Lines of Code	2	Metric	2	3	25
TODO Comment Format	2	Formatting	1	1	13
Avoid Namespace Using-Directives	2	Code Standard	3	1	33
Avoid String Printing C Functions	1		1	1	12
Include Directory in Header	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Whitespace Around Colon	1	Formatting	1	1	12
Smallest Feasible Scope	1	Code Standard	1	2	13
C Standard Library	1	Code Standard	1	1	12
Empty Semicolon Statement	1	Formatting	1	1	12
No Boolean Vectors	1	Code Standard	3	1	32
Single Statement If-Else	1	Formatting	1	1	12
Total issues	**4217**		**Total Score**		

Table 5.9: Laserscan to pointcloud package analysis

Package - laserscan_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	811	Formatting	1	1	822
Line Length	464	Formatting	1	1	475
Opening/Closing Curly Brace	118	Formatting	1	1	129
Include Order	42	Code Standard	1	1	53
Non-const Reference Parameters	40	Code Standard	1	2	52
Integer Types	18	Code Standard	2	2	40
No Copyright Statement	13	Code Standard	1	1	24
Casting	12	Code Standard	3	1	43
Function Length	9	Metric	2	3	32
Include	8	Code Standard	2	1	29
Whitespace Before Comments	8	Formatting	1	1	19
No Header Guard	6	Code Standard	2	1	27
Cyclomatic Complexity	4	Metric	3	3	37
Whitespace After Comma	4	Formatting	1	1	15
Function Parameters	3	Metric	2	3	26
Halstead Bugs	2	Metric	3	3	35
Halstead Volume	2	Metric	3	3	35
Make Constructors Explicit	2	Code Standard	1	1	13
One Command Per Line	2	Formatting	1	1	13
Whitespace Before Comment Text	2	Formatting	1	1	13
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Unused Variables	1	Code Standard	2	1	22
Blank Lines In Code Blocks	1	Formatting	1	1	12
Total issues	**1574**		**Total Score**		**2022**

Table 5.10: Object recognition skill server package analysis

Package - object_recognition_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	108	Formatting	1	1	119
Line Length	29	Formatting	1	1	40
Opening/Closing Curly Brace	22	Formatting	1	1	33
Include Order	6	Code Standard	1	1	17
Integer Types	3	Code Standard	2	2	25
No Copyright Statement	3	Code Standard	1	1	14
Include	2	Code Standard	2	1	23
Non-const Reference Parameters	2	Code Standard	1	2	14
End of Namespace Comment	2	Formatting	1	1	13
No Header Guard	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Function Length	1	Metric	2	3	24
Total issues	**180**		**Total Score**		**378**

Table 5.11: Mesh to pointcloud package analysis

Package - mesh_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	163	Formatting	1	1	174
Line Length	71	Formatting	1	1	82
Opening/Closing Curly Brace	38	Formatting	1	1	49
Non-const Reference Parameters	20	Code Standard	1	2	32
Include Order	14	Code Standard	1	1	25
No Copyright Statement	4	Code Standard	1	1	15
End of Namespace Comment	3	Formatting	1	1	14
Integer Types	2	Code Standard	2	2	24
Halstead Bugs	2	Metric	3	3	35
Include	1	Code Standard	2	1	22
No Header Guard	1	Code Standard	2	1	22
Total issues	**319**		**Total Score**		**494**

Table 5.12: Pose to tf publisher package analysis

Package - pose_to_tf_publisher					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Indent With 2 Whitespace	491	Formatting	1	1	502
Line Length	171	Formatting	1	1	182
Opening/Closing Curly Brace	89	Formatting	1	1	100
Include Order	20	Code Standard	1	1	31
Non-const Reference Parameters	8	Code Standard	1	2	20
Smallest Feasible Scope	8	Code Standard	1	2	20
No Copyright Statement	3	Code Standard	1	1	14
Cyclomatic Complexity	3	Metric	3	3	36
Function Length	3	Metric	2	3	26
Whitespace Before Comments	3	Formatting	1	1	14
One Command Per Line	3	Formatting	1	1	14
Integer Types	2	Code Standard	2	2	24
Include	2	Code Standard	2	1	23
Halstead Bugs	2	Metric	3	3	35
Parenthesis & Whitespace	2	Formatting	1	1	13
Make Constructors Explicit	1	Code Standard	1	1	12
Halstead Volume	1	Metric	3	3	34
Order of Evaluation	1	Code Standard	3	1	32
File Length	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Whitespace Before Comment Text	1	Formatting	1	1	12
Blank Lines In Code Blocks	1	Formatting	1	1	12
Float Accuracy	1	Code Standard	3	1	32
Total issues	**818**		**Total Score**		**1244**

Table 5.13: Octomap server package analysis

Package - octomap_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	368	Formatting	1	1	379
Line Length	180	Formatting	1	1	191
Indent With 2 Whitespace	81	Formatting	1	1	92
Integer Types	55	Code Standard	2	2	77
Blank Lines In Code Blocks	51	Formatting	1	1	62
Function Length	36	Metric	2	3	59
Parenthesis & Whitespace	30	Formatting	1	1	41
Include Order	22	Code Standard	1	1	33
Non-const Reference Parameters	20	Code Standard	1	2	32
Whitespace Before Comments	20	Formatting	1	1	31
Include	14	Code Standard	2	1	35
Casting	13	Code Standard	3	1	44
Whitespace After Comma	12	Formatting	1	1	23
TODO Comment Format	12	Formatting	1	1	23
Indent Access Modifiers	11	Formatting	1	1	22
Avoid Namespace Using-Directives	10	Code Standard	3	2	42
Cyclomatic Complexity	9	Metric	3	3	42
Whitespace Around Binary Operator	9	Formatting	1	1	20
Whitespace at the End of Line	9	Formatting	1	1	20
End of Namespace Comment	7	Formatting	1	1	18
One Command Per Line	7	Formatting	1	1	18
Whitespace Around Assignment	6	Formatting	1	1	17
Header Guard Format	6	Code Standard	1	1	17
Whitespace Before Comment Text	5	Formatting	1	1	16
Order of Evaluation	4	Code Standard	3	1	35
Make Constructors Explicit	4	Code Standard	1	1	15
Redundant Empty Statement	4	Formatting	1	1	15
Halstead Bugs	3	Metric	3	3	36
Halstead Volume	3	Metric	3	3	36
Header Guard Must Close	3	Code Standard	2	1	24
Smallest Feasible Scope	2	Code Standard	1	2	14
File Length	1	Metric	3	3	34
Unused Variables	1	Code Standard	2	1	22
No Boolean Vectors	1	Code Standard	3	1	32
No Redundant Variables	1	Code Standard	1	1	12
Total issues	**1020**		**Total Score**		**1629**

Table 5.14: PCL conversions package analysis

Package - pcl_conversions					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Opening/Closing Curly Brace	113	Formatting	1	1	124
Non-const Reference Parameters	76	Code Standard	1	2	88
Line Length	46	Formatting	1	1	57
Include Order	21	Code Standard	1	1	32
Parenthesis & Whitespace	21	Formatting	1	1	32
Whitespace Before Comments	15	Formatting	1	1	26
No Namespace Indentation	10	Formatting	1	1	21
Whitespace Around Binary Operator	5	Formatting	1	1	16
End of Namespace Comment	3	Formatting	1	1	14
Cyclomatic Complexity	2	Metric	3	3	35
Header Guard Format	2	Formatting	1	1	13
Include	1	Code Standard	2	1	22
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	2	Metric	3	3	35
Indent Access Modifiers	1	Formatting	1	1	12
File Length	1	Metric	3	3	34
Function Length	1	Metric	2	3	24
Header Guard Must Close	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Whitespace Before Comment Text	1	Formatting	1	1	12
Integer Types	1	Code Standard	2	2	23
No Copyright Statement	1	Code Standard	1	1	12
Unused Variables	1	Code Standard	2	1	22
Total issues	**327**		**Total Score**		**744**

The initial analysis of the source code identified 66 different types of issues, from which 24 were of the Formatting kind, 34 were Code Standard issues, while the remaining 8 were Code Metrics issues. Moreover, by inspecting the results of the initial analysis conducted using High Assurance ROS, it is possible to infer that the majority of issues were of the Formatting kind, namely 24511 issues. Code Standard and metrics issues were 3175 and 356 respectively, which then encompassed for a total of 28040 issues. Additionally, from this initial analysis, the issues had an average Severity score of 1.61, and an average Effort to Solve score of 1.47.

5.3 Issues

This section provides detail on each issue discovered during the initial analysis. The causes for each issue, such as what code standards or rule it violates, were identified. Then, its type was identified and the severity level attributed was justified, along with the effort needed to solve it. The evaluation of severity and effort to solve was quantified using integer levels on a range of 1 to 3, with 1 being low, 2 being medium and 3 being high. Issues can be defined as follow:

- **Alternative Tokens** - This issue is triggered by the use of alternative tokens, such as *"or"* and *"and"*, instead of "||" and "&&". This is a code standard issue that violates Google C++ style guide [38]. This issue does not represent a threat and it is simple to fix. Therefore, it belongs to level 1 of Severity and of Effort to Solve.

- **Avoid C System Headers** - This issue is triggered by the use of C system headers when there are C++ counterparts. This is a code standard issue that violates Google C++ style guide [38]. This issue does not represent a threat, thus, it belongs to level 1 of Severity. To fix this issue, only a change in the header to the C++ version is necessary. Therefore this issue belongs to level 1 of Effort to Solve.

- **Avoid Namespace Using-Directives** - This issue is triggered by the use of a using-directive of a namespace. This is a code standard issue that violates Google C++ style guide [38] and MISRA C++ [36]. The *using directives* increases the scope of names that are being looked up. This can cause the compiler to find an identifier that is different from the one expected by the developer, which can be very dangerous. Thus, this issue belongs to level 3 of Severity. To solve this issue, *using declarations* could be used since they are a safer option [36]. The other solution is to just use the full declaration every time. This issue can be time-consuming to solve, thus it belongs to level 2 of the Effort to Solve.

- **Avoid Rvalue References** - This issue is triggered by the use of Rvalue references. This is a code standard issue that violates Google C++ style guide [38]. The Rvalue references are not widely understood, so it is not recommended to use them unless it is for moving constructors and forwarding references. Thus, this issue belongs to Severity level 2. Moreover, it could be difficult to solve, so it belongs to level 3 of Effort to Solve.

- **Avoid String Printing C Functions** - This issue is caused by the use of C printing functions. It is recommended to use *snprintf* instead since it has a limited buffer and can prevent buffer overflow. This issue can be a threat, thus it belongs to level 2 of Severity. The solution is simple, so it belongs to level 1 of Effort to Solve.

- **Avoid Thread-Unsafe Functions** - This issue is triggered by the use of thread-unsafe functions, in these cases by the use of the function *rand()*. This is a code standard issue that violates Google C++ style guide [38]. These functions are not designed to be accessed simultaneously and that can cause deadlocks and other risky situations. Thus, this issue

belongs to level 3 of Severity. To solve it, thread-safe alternatives to these functions. In this particular case, the *_rand_r()* function should be used instead. Since this issue is simple to fix, it belongs to level 1 of the Effort to Solve.

- **Avoid Unapproved Classes and Functions** - This issue is triggered by using functions or classes that belong to unapproved headers. This is a code standard issue that violates Google C++ style guide [38]. These functions themselves do not represent a threat. Although they originate from unverified parties, that does not mean they are unsafe. Thus, this issue belongs to level 1 of Severity. However, it could be hard to find a replacement for this function in other headers which difficults this issue's solving. Therefore, this issue belongs to level 3 of the Effort to Solve.

- **Avoid Unapproved Headers** - This issue is triggered by the use of unapproved headers. This is a code standard issue that violates Google C++ style guide [38]. These headers themselves do not represent a threat. They come from unverified parties that could not take so much care with the software, yet that also does not mean they are unsafe. Thus, this issue belongs to level 1 of Severity. Once more, it could be hard to find a replacement for these headers which difficults this issue's solving. Therefore, this issue belongs to level 3 of the Effort to Solve.

- **Blank Lines Before Section** - This issue is triggered by not having a blank line before an access modifier (public, private, protected). This is a formatting issue that violates Google C++ style guide [38]. The blank line before the access modifier for the first instance is not necessary, i.e., it is not necessary at the beginning of the class [38]. This issue does not represent a threat. Thus, it belongs to level 1 of Severity. To solve this issue, it is only necessary to add the blank line before the access modifier, making it simple to solve. Therefore it belongs to level 1 of Effort to Solve.

- **Blank Lines In Code Blocks** - This issue is triggered by the presence of a blank line in a code block. This is a formatting issue that violates Google C++ style guide [38]. According to Google, within a code block, a blank line should be used as a paragraph to separate two different things. This issue does not represent a threat, as it can only reduce the readability of the code. Thus, it belongs to level 1 of Severity. This issue is also simple to fix, so it belongs to level 1 of Effort to Solve.

- **C Standard Library** - This issue is triggered by the use of the C standard library in an unwrapped way. This is a code standard issue that violates MISRA C++ guidelines [36]. If this library is used, it should be placed on a separate file to ensure the absence of undefined behaviour. Since it can lead to undefined behaviour, this issue belongs to level 2 of Severity. This issue is simple to fix, so it belongs to level 1 of Effort to Solve.

- **Casting** - This issue is triggered by the use of a deprecated type of casting - the C type cast. This is a code standard issue that violates Google C++ style guide [38]. The new-style

casting can be more clearly identified. This style was created so that the developers could state their intentions and for the compilers to be able to detect more errors [39]. Old-style cast is dangerous, hence this issue belongs to level 3 of Severity. Despite dangerous, this issue is simple to fix. Therefore, it belongs to level 1 of Effort to Solve.

- **Complex Multi-line Comments and Strings** - This issue is triggered by comments or strings that are over one line. This is a formatting issue that violates Google C++ style guide [38]. It does not represent a threat, and the correct formatting can improve the readability. Thus, this issue belongs to level 1 of Severity. Since this issue is simple to fix, it belongs to level 1 of the Effort to Solve.

- **Cyclomatic Complexity** - This issue is triggered by functions that have over 10 or 15 Cyclomatic Complexity. This is a metric issue. Functions with high cyclomatic complexity represent a threat and they are hard to maintain, understand and very hard to test. Thus, it belongs to level 3 of Severity. Given these functions' characteristics, they will also be hard to change, making these issues hard to solve. Therefore, this issue belongs to level 3 of Effort to Solve.

- **Do Not Include Twice** - This issue is triggered by including the same header file twice. This is a code standard issue. Including the same header twice will cause a compilation error if the header does not have a header guard or something that prevents multiple inclusion. Thus, this issue belongs to level 2 of Severity. To solve these issues, one of the inclusions should be removed, making it simple to fix. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Do Not Use C Types** - This issues is similar to the issue **Integer Types**. It is a coding standard issue that violates Google C++ style guide [38]. This issue belongs to level 2 of Severity and to level 2 of Effort to Solve.

- **Do Not Use Default Lambda Captures** - This issue is triggered by the use of default lambda captures. This is a code standard issue that violates Google C++ style guide [38]. The default use of the lambda expression is dangerous and it is not clear. To use lambda expression, it is recommended to follow the appropriate format, as suggested by Google. Since it leads to code that is dangerous and hard to understand, this issue belongs to level 3 of Severity. To fix these issues, the format suggested by Google should be followed. However, since this feature can be hard to understand, this issue belongs to level 2 of Effort to Solve.

- **Empty Semicolon Statement** - This issue is triggered by the use of a semicolon to denote a empty statement, example *for(i;a;i++);*. This is a formatting issue that violates Google C++ style guide [38]. Curly braces are recommended to denote a empty statement, such as *for(i;a;i++) { }* , or closing braces in a new line. This issue does not represent a threat, thus

it belongs to level 1 of Severity. To solve this issue, the semicolon should be replaced by the empty curly-braces, which is simple. Thus, it belongs to level 1 of Effort to Solve.

- **End of Namespace Comment** - This issue is triggered by the absence of an end of namespace comment. This is a coding standard issue that violates Google C++ style guide [38]. This issue does not represent a threat, as it is only used to improve the readability of the code. Thus, it belongs to Severity level 1. Every namespace should end with a comment like this *// namespace nameOfNamespace*. To fix this issue, the comment should be added. Therefore, this issue belongs to level 1 of Effort to Solve, due to the easiness to fix it.

- **File Length** - This issue is triggered by a file with over 400 lines of code. This is a metric issue. A file too long is consequently too complex. Overly complex code is more prone to issues, which is dangerous. Thus, this issue belongs to level 3 of Severity. This is a very complex issue to solve, as it could be almost impossible to solve. Therefore this issue belongs to level 3 of Effort to Solve.

- **Float Accuracy** - This issue is triggered by code that expects floating point calculations to yield exact results. This is a code standard issue that violates MISRA C++ [36]. The test of equality or inequality of float points shall not be directly tested. This issue is highly dangerous and will yield unpredictable behaviour. Thus, this issue belongs to level 3 of Severity. However, this issue is simple to fix and a solution is suggested in MISRA C++ guidelines [36]. Therefore, it belongs to level 1 of the Effort to Solve.

- **Function Length** - This issue is triggered by functions with over 40 lines of code. This is a metric issue. Very long functions tend to be complex and hard to understand, which is dangerous and compromises the code's maintainability. Thus, this issue belongs to level 3 of Severity. Functions should be shorter and focused. To solve this issue, the code would need to be refactored. Therefore, this issue belongs to level 3 of the Effort to Solve.

- **Function Parameters** - This issue is triggered by a function that has more than 6 parameters. This is a metric issue. Functions with too many parameters are more difficult to use, and thus increasingly prone to cause errors. For this reason, this issue belongs to level 3 of Severity. This metric is also hard to change since the parameters are needed. Therefore, this issue belongs to level 3 of Effort to Solve.

- **Halstead Bugs** - This is an issue triggered by code with Halstead Bugs over 2. This is a metric issue. Triggering this issue indicates that the code is too complex. Halstead Bus over 2 is very dangerous, as it estimates that are more than 2 bugs being delivered with this code. Thus, this issue belongs to level 3 of Severity. Moreover, due to the complexity of the code, this issue is also very hard to solve. Therefore, it belongs to level 3 of Effort to Solve.

- **Halstead Volume** - This is an issue triggered by code with Halstead Volume over 8000. This is a metric issue. Triggering this issue implies that the code is too complex. Files that

are too complex are very dangerous, difficult to understand and to develop. Thus, this issue belongs to level 3 of Severity. Due to the difficulty to develop and understand the code, this issue is also very hard to solve. Hence, it belongs to level 3 of Effort to solve.

- **Header Guard Format** - This issue is triggered by a header guard that does not follow the ROS C++ style guide. This is a formatting issue. Following the standard is needed to ensure the uniqueness of the header guard. This is not a serious issue, as it can only cause problems if a file with the same guard exists, which is probably unlikely. Thus, this issue belongs to level 1 of Severity. This is an easily fixable issue, so it belongs to level 1 of Effort to Solve. The format to adopt regarding this issue is described on 5.1.

- **Header Guard Must Close** - This issue is triggered by the absence of the closing of the header guard. This issue is a code standard issue that violates both Google C++ style guide [38] and ROS C++ style guide [40]. The absence of the closing will lead to a compiling error. Thus, it belongs to level 2 of Severity. The closing of the header guard should also have a comment with the same name as the header guard 5.1. To solve this issue, the header guard should be closed using the appropriate format. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Include** - This issue is triggered by the absence of a header file, necessary for some code that is being used. This is a code standard issue that violates Google C++ style guide [38]. Not including the header will unable the code to compile. Thus, this issue belongs to level 2 of Severity. To solve this issue, the necessary headers should be added, respecting the include order. Therefore, this issue belongs to level 1 of the Effort to Solve.

- **Include Directory in Header** - This issue is triggered by the inclusion of a header without the proper directory. This is a code standard issue that violates Google C++ style guide [38]. For example, the file *project/src/base/code.h* should be included as: *#include "base/code.h"*. This issue does not represent a threat. Thus, it belongs to level 1 of Severity. To solve this issue, the necessary directory should be added to the header, which is simple. Therefore this issue belongs to level 1 of Effort to Solve.

- **Include Order** - This issue is triggered by the inclusion of headers without the proper order. This is a formatting issue that violates Google C++ style guide [38]. The order of inclusion is not always a threat, but some headers might need to be included before others to ensure that the code compiles. Thus, this issue belongs to level 1 of Severity. To solve this issue, the order of inclusion presented below should be followed, which is simple. Therefore, this issue belongs to level 1 of the Effort to Solve.

  1. File
     blank line
  2. C system files

3. C++ System files
   blank line
4. Other libraries .h files
5. Project .h files

- **Indent Access Modifiers** - This issue is triggered by the access modifiers within a class not being properly indented. This is a formatting issue that violates Google C++ style guide [38]. According to Google, the access modifiers should be indented by 1 space [38]. This issue does not represent a threat and it only improves the readability of the code. Thus, it belongs to level 1 of Severity. This issue is easy to fix, as it can even be fixed by an automatic tool. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Indent With 2 Whitespace** - This issue is triggered by the use of tabs to indent. This is a coding standard issue that violates both Google C++ style guide [38] and ROS C++ style guide [40]. Both these guides recommend indentation with 2 spaces. This issue does not represent a threat, as the indentation is only used to improve the readability of the code. Accordingly, it belongs to level 1 of Severity. This issue is simple to solve, so it belongs to level 1 of Effort to Solve. A solution to avoid this issue and avert pressing the space key several times to indent is to set the editor to place 2 spaces when the tab key is pressed, as Google suggests [38].

- **Integer Types** - This issue is triggered by the use of non-size-specific types of *int*. This is a Code standard issue that violates the Google C++ style guide [38]. Google suggests the use of size-specific types instead of *short* and *long* types. This issue is not usually a threat but it can cause some errors. For that reason, it belongs to level 1 of Severity. The choice of which size to pick can sometimes be difficult, but when in doubt, it the bigger type should be used. Therefore this issue belongs to level 1 of Effort to Solve.

- **Line Length** - This issue is triggered by a line of code with a length over 80 characters. This is a formatting issue that violates the Google C++ style guide [38], but it does not violate ROS C++ style guide [40], which allows lines with a maximum length of 120 characters [40]. Given that this is a formatting issue, it does not represent a threat. Therefore, it will not cause runtime errors or compilation errors. Thus, it belongs to Severity level 1. Although it is not a threat, it seriously affects the readability of the code and its maintenance, which might lead to the production of faulty code. Most of the issues of this type can be easily solved with some simple formatting, which can even be automatic, while others are impossible to fix due to the use of functions with a long name, impossible to break. Given its easily fixed, it belongs to level 1 of Effort to Solve.

- **Maintainability Index** - This issue is triggered by having a maintainability index under 65. This is a metric issue. A low maintainability index means that the code is difficult to maintain, complex, and risky. Thus, this issue belongs to level 3 of Severity. This is a very

complex problem to solve, as the better solution is to rewrite the code. For that reason, this issue belongs to level 3 of the Effort to Solve.

- **Make Constructors Explicit** - This issue is triggered by not having the constructor explicit. This is a code standard issue that violates Google C++ style guide [38]. This issue does not represent a threat, thus it belongs to level 1 of Severity. The *explicit* keyword helps to ensure the right format. To solve these issues, the keyword *explicit* should be added to constructors that can be called with no argument or with a single argument. Since it is simple to fix, it belongs to level 1 of Effort to Solve.

- **Maximum Executable Lines of Code** - This issue is triggered by having more than 50 lines of code. This is a metric issue. Very long functions tend to be complex and hard to understand, which is dangerous and compromises the maintainability. Thus, this issue belongs to level 3 of Severity. Functions should be shorter and focused. To solve this issue, the code would need to be refactored. Therefore, this issue belongs to level 3 of the Effort to Solve.

- **Newline at End of File** - This is triggered by the absence of a new line at the end of the file. This is a formatting issue that violates Google C++ style guide [38]. This does not represent a threat, as it only improves the readability. Thus, it belongs to level 1 of Severity. To solve this issue, a new line should be added at the end of the file. Therefore, this issue belongs to level 1 of the Effort to Solve.

- **No Boolean Vectors** - This issue is triggered by the use of a boolean vector. This is a code standard issue that violates MISRA C++ [36] and High integrity C++ code standard. The use of boolean vectors, *std::vector<bool>* does not comply with the requirements of a container and does not work as expected in all algorithms. Thus, this issue belongs to level 3 of Severity. To solve this, the boost container can be used, *boost::container::vector<bool>*. Since the solution is simple, this issue belongs to level 1 of the Effort to Solve.

- **No Copyright Statement** - This issue is triggered by the absence of a copyright statement on a file. This is a code standard issue, that violates both Google C++ style guide [38] and ROS C++ style guide [40]. According to Google [38], every file should contain a copyright statement and a license. However, this issue does not represent a threat. Thus, it belongs to level 1 of Severity. To solve this issue, the necessary copyright statement and license should be added. Hence, it belongs to level 1 of Effort to Solve.

- **No Header Guard** - This issue is triggered by the absence of a header guard on a header file. This is a Code Standard issue that violates the Google C++ style guide [38] and ROS C++ style guide [40]. The header guard is used to prevent multiple inclusion of the same header, thus preventing circular references between headers. This prevention reduces the compiling time and, most importantly, it prevents compilation errors caused by circular references. Since the absence of the header guard can cause a compilation error, this issue

belongs to Severity level 2. The header guard must have a unique name to ensure all the necessary headers are included. Google and ROS style guides suggest similar ways of archiving uniqueness of header guard names. Google suggests <PROJECT>_<PATH>_<FILE>_H_ whilst ROS suggests PACKAGE_PATH_FILE_H. This issue is simple to fix, so it is a level 1 on Effort to Solve. The analysed code contained another approach on this topic, by using *#pragma once*. Although this approach achieves the same result, it is not recommended by Google C++ style guide [38].

Listing 5.1: Header Guard Format

```
1 #ifndef PACKAGE_PATH_FILE_H
2 #define PACKAGE_PATH_FILE_H
3 ...
4 #endif  // PACKAGE_PATH_FILE_H
```

- **No Namespace Indentation** - This issue is triggered by indenting the content of a namespace. This is a formatting issue that violates both Google C++ style guide [38] and ROS C++ style guide [40]. According to Google C++ style guide [38], *"namespaces do not add an extra level of indentation"*, therefore their content should not be indented. This issue does not represent a threat, thus it belongs to level 1 of Severity. Since it is an indentation problem, it is simple to fix and it belongs to level 1 of Effort to solve.

- **No Redundant Variables** - This issue is triggered by the presence of initialized variables that are not used. Although it is not a threat, it might indicate that some other variable is being used instead, and that can be dangerous. Thus, this issue belongs to level 2 of Severity. To solve it, the variable that is not needed should be removed, which is easily fixable. Therefore, this issue belongs to level 1 of Effort to solve

- **No Uninitialized Member Variables** - This issue is triggered by a member variable of a class that is not initialized when the constructor is called. This is a code standard issue issue that violates both Google C++ style guide [38]. This issue belongs to level 2 of Severity and since it is simple to solve, it belongs to level 1 of Effort to Solve.

- **No Unions** - This issue is triggered by the use of unions. This is a code standard issue that violates MISRA C++ guidelines [36]. The use of unions can cause misinterpretation, which is dangerous. MISRA C++ guidelines forbid the use of unions for any purpose. Thus, this issue belongs to level 3 of Severity. This issue might not be very straightforward to solve, so it belongs to level 2 of Effort to Solve.

- **Non-const Reference Parameters** - This issue is triggered by functions where parameters are passed by reference, but not *const* reference. This is a code standard issue that violates Google C++ style guide [38]. According to Google, in C++, the keyword *const* should be used to pass by *const* reference. However, this makes it impossible for the function to

change the value of the passed reference. In those cases, the value should be passed by pointer. This issue does not represent a threat, but the recommended style is clearer. Thus, this issue belongs to level 1 of Severity. Despite that, it can be difficult to fix, mainly due to extensive and cross-file changes that are required. For these reasons, this issue belongs to level 2 of the Effort to solve.

- **One Command Per Line** - This issue is triggered by having more than one command on the same line. This is a code standard issue that violates Google C++ style guide [38]. Although this issue does not represent a threat, it compromises the readability of the code. Thus, this issue belongs to level 1 Severity. To solve it, every command just needs to be on its line, making this issue belong to level 1 of Effort to Solve.

- **Opening/Closing Curly Brace** - This issue is triggered by a opening or closing brace in a manner that is not conforming with Google C++ style guide [38] or with ROS C++ style guide [40]. These two styles, in some cases, are opposite to each other. Therefore when following one style, this issue will be triggered because it is not conforming with the opposite style. In some cases, the ROS C++ style adds too much empty vertical space, which might not improve the readability. The Google style, on the other hand, does not waste vertical space. This is a formatting issue and it does not represent a threat. Thus, it belongs to level 1 of Severity. This is a simple problem to fix, so it belongs to level 1 of Effort to solve. Despite that, the use an automated indenter is recommended in order to keep the code indentation uniform.

- **Order of Evaluation** - This issue is triggered by code with expressions that rely on the order of evaluation. This is a code standard issue that violates MISRA C++ guidelines [36], High Integrity C++ code standard, and Joint Strike Fighter Air Vehicle C++ code standard. An expression should yield the same result whichever order of evaluation is used. Hence, the order of evaluation should not be trusted, since it varies from compiler to compiler. This issue could be very dangerous, once depending on the compiler used, it might cause unwanted behaviour or unexpected results. Thus, it belongs to level 3 of Severity. Despite its severity, this issue is simple to fix. Therefore, it belongs to level 1 of Effort to Solve. An example of an issue of this kind and how to avoid it is present on Listing 5.2.

Listing 5.2: Order of evaluation example

```
1 a=v[j] + j++; //bad, a depends on whether
2 //v[j] or j++ is evaluated first
3
4 a=v[j]+j; //good, the result does not depend
5 j++;  //on the order of evaluation
```

- **Parenthesis & Whitespace** - This issue is triggered when the spaces around an opening or closing parenthesis are not correctly formatted. This is a formatting issue that violates

Google C++ style guide [38]. This issue does not represent a threat, yet it compromises the readability. Thus, it belongs to level 1 of Severity. Regardless, it is an easily solvable issue, so it belongs to level 1 of Effort to solve.

- **Redundant Empty Statement** - This issue is triggered by a semicolon after a closing curly-brace, which is redundant. This is a formatting issue that violates Google C++ style guide [38]. Since it does not represent a threat, it belongs to level 1 of Severity. To solve this issue, the semicolon should be removed, which is simple. Therefore this issue belongs to level 1 of Effort to Solve.

- **Single Else-If Else Line** - This issue is triggered by an else statement which is not on its own line. This is a formatting issue that violates Google C++ style guide [38]. The else statement should be on its own line, yet this issue is not a threat, as it only improves the readability. Thus, it belongs to level 1 of Severity. The solution for it is simple, so it belongs to level 1 of Effort to Solve.

- **Smallest Feasible Scope** - This issue is triggered by a variable that can be declared a level lower on the scope. This is a code standard issue that violates Google C++ style guide [38]. Since this does not represent a threat, it belongs to level 1 of Severity. This is also simple to fix, so it belongs to level 1 of Effort to Solve. An example of this issue is presented in Listing 5.3.

Listing 5.3: Smallest scope

```
1 int a;
2 {
3   a=5; // a could be declared in this scope
4 }
5 {
6   int a;  // this reduces the scope
7   a=5;
8 }
```

- **Storage Class Before Type** - This issue is triggered by an expression which contains the storage class after the type. This is a Code standard issue that violates Google C++ style guide [38]. The cause of this issue was *const static unsigned* and the correct way is *static const unsigned*. This issue does not represent a threat and it is simple to fix, so it belongs to level 1 of both Severity and Effort to Solve.

- **TODO Comment Format** - This issue is triggered by a TODO comment that does not follow the Google C++ style guide [38]. This is a code standard issue. Google suggests that this comment should have the TODO in all caps followed by identification of a person or identification of the bug and, finally, the context of the problem. This issue does not

represent a threat, but these comments are very useful to have bug free and best solutionable software. Since it is not a threat, it belongs to level 1 of Severity. This issue is also easy to fix. Therefore, it belongs to level 1 of Effort to Solve.

- **Unused Variables** - This issue is triggered by the presence of unused variables in the code. This is a code standard issue that violates MISRA C++ guidelines [36]. Unused variables are noise and should be removed. They can lead to the use of the wrong variables in some places, which is not ideal. For that reason, this issue belongs to level 2 of Severity. To solve it, those variables should be removed. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Whitespace After Comma** - This issue is triggered by whitespace before a comma. The whitespace should come after the coma. This is a formatting issue that violates Google C++ style guide [38]. It does not represent a threat, so it belongs to level 1 of Severity. To solve this issue, the right style should be followed. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Whitespace Around Assignment** - This issue is triggered by the absence of whitespace around an assignment. This is a formatting issues that violates Google C++ style guide [38]. There should always be a whitespace before and after an assignment, in order to improve the readability. This issue does not represent a threat, so it belongs to level 1 of Severity. The solution for it is to add the necessary whitespaces, which is easy to do. Therefore, it belongs to level 1 of Effort to Solve.

- **Whitespace Around Binary Operator** - This issue is triggered by the absence of whitespace around a binary operator. This is a formatting issue that violates Google C++ style guide [38]. Around a binary operator, there should be a whitespace before and after it. This issue does not represent a threat, thus it belongs to level 1 of Severity. To solve it, the necessary whitespaces need to be added, which is simple. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Whitespace Around Colon** - This issue is triggered by the presence of a whitespace around a colon. This is a formatting issue that violates Google C++ style guide [38]. Since it is a formatting issue, it does not represent a threat. Thus, it belongs to level 1 of Severity. To solve this issue, the whitespaces around this character need to be removed, which is simple. Therefore, this issue belongs to level 1 of Effort to Solve.

- **Whitespace Around Unary Operator** - This issue is triggered by the presence of a whitespace around a unary operator. This is a formatting issue that violates Google C++ style guide [38]. This issue does not represent a threat. Thus, it belongs to level 1 of Severity. To solve this issue, the whitespace should be removed, making it simple to fix. Therefore, it belongs to level 1 of Effort to Solve.

- **Whitespace at the End of Line** - This issue is triggered by the presence of a whitespace at the end of the line. This is a formatting issue that violates Google C++ style guide [38]. The present issue does not represent a threat, thus, it belongs to level 1 of Severity. To solve it, the whitespace should just be removed, so it belongs to level 1 of Effort to Solve.

- **Whitespace Before Comment Text** - This issue is triggered by the absence of a whitespace between the start of a comment that uses the block comment syntax,/* */, and the text. This is a formatting issue that violates Google C++ style guide [38]. This issue does not represent a threat, yet it improves the readability of the comments. Thus, this issue belongs to level 1 of Severity. To solve it, a space should be added between the comment start and the text, making it simple to fix. Therefore, it belongs to level 1 of Effort to Solve.

- **Whitespace Before Comments** - This issue is triggered by the absence of a whitespace between the start of a single line comment and the text of that comment. This is a formatting issue that violates Google C++ style guide [38]. This aims to improve the readability, so it is not a threat. Thus, this issue belongs to level 1 of Severity. To fix it, the whitespace should be added, which is simple. Hence, it belongs to level 1 of Effort to Solve.

5.4 First Iteration

For this first iteration, it was assumed that the code did not follow any code standard format. By the evaluation of the results in the initial analysis, and from table 5.57, it is pretty clear that most of the issues are of the formatting type. This means that these should be the first to be tackled. Since the code is vast, it would be impractical and extremely time consuming to correct all the formatting issues by hand. So, in order to tackle this kind of issues, a automatic approach was taken. The chosen tool was the *Clang-Format* along with Visual Studio Code. These tools were selected considering their characteristics: i) having predefined styles, ii) being configurable, and iii) being recommended by ROS.

The *Clang-Format* was used to format the code accordingly to Google C++ style guide. The decision to choose Google C++ style instead of ROS C++ Style was based on the fact that the source code already seemed to follow some of the guidelines of that style. After the use of this tool, some additional adjustments had to be done by hand. This was needed to ensure that the code would still compile. The adjustments performed by hand were mostly related with the include order issue, since some of the header files were not named to allow the correct order of inclusion. Once this iteration was concluded, the results were organized in the same way as the ones of the initial analysis and are presented in the following tables.

Table 5.15: Move arm skill server package analysis

Package - move_arm_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	22	Code Standard	2	2	44
Function Length	8	Metric	2	3	31
Casting	8	Code Standard	3	1	39
Include	5	Code Standard	2	2	27
Cyclomatic Complexity	4	Metric	3	3	37
No Copyright Statement	3	Code Standard	1	1	14
File Length	1	Metric	2	3	24
No Header Guard	1	Code Standard	2	1	22
Non-const Reference Parameters	1	Code Standard	1	2	13
Total issues	**53**		**Total Score**		**251**

Table 5.16: Arm action controller package analysis

Package - arm_action_controller					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	12	Code Standard	2	2	34
Function Length	7	Metric	2	3	30
Non-const Reference Parameters	5	Code Standard	1	2	17
Include	3	Code Standard	2	1	24
No Copyright Statement	3	Code Standard	1	1	14
Cyclomatic Complexity	3	Metric	3	3	36
Function Parameters	2	Metric	3	3	35
Maintainability Index	2	Metric	3	3	35
Include Order	1	Formatting	1	1	12
End of Namespace Comment	1	Formatting	1	1	12
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Casting	1	Code Standard	3	1	32
Total issues	**43**		**Total Score**		**371**

Table 5.17: Arm interface package analysis

Package - arm_interface					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Function Length	21	Metric	2	3	44
Include	20	Code Standard	2	1	41
Non-const Reference Parameters	15	Code Standard	1	1	26
No Copyright Statement	13	Code Standard	1	1	24
End of Namespace Comment	13	Formatting	1	1	24
Casting	11	Code Standard	3	1	42
Maintainability Index	7	Metric	3	3	40
Include Order	6	Code Standard	1	1	17
Halstead Bugs	6	Metric	3	3	39
No Header Guard	6	Code Standard	2	1	27
Line Length	5	Formatting	1	1	16
Halstead Volume	5	Metric	3	3	38
Cyclomatic Complexity	4	Metric	3	3	37
Blank Lines In Code Blocks	3	Code Standard	1	1	14
Function Parameters	3	Metric	2	3	26
Order of Evaluation	1	Code Standard	3	1	32
Avoid Thread-Unsafe Functions	1	Code Standard	3	2	33
Avoid Rvalue References	1	Code Standard	3	2	1
Complex Multi-line Comments and Strings	1	Formatting	1	1	12
Total issues	**163**		**Total Score**		**609**

Table 5.18: Ur modern driver package analysis

Package - ur_modern_driver					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	194	Code Standard	1	2	206
Integer Types	44	Code Standard	2	2	66
Include	40	Code Standard	2	1	61
No Header Guard	35	Code Standard	2	1	56
Make Constructors Explicit	22	Code Standard	1	1	33
Newline at End of File	19	Formatting	1	1	30
Function Length	17	Metric	2	3	40
Avoid Unapproved Headers	16	Code Standard	1	3	29
Avoid C System Headers	13	Code Standard	2	1	34
Do Not Use C Types	17	Code Standard	2	2	39
Casting	12	Code Standard	3	1	43
Halstead Bugs	11	Metric	3	3	44
Halstead Volume	8	Metric	3	3	41
Avoid Rvalue References	8	Code Standard	3	2	40
Avoid Namespace Using-Directives	6	Code Standard	3	2	38
Maximum Executable Lines of Code	3	Metric	2	3	26
Maintainability Index	3	Metric	3	3	36
Avoid Unapproved Classes and Functions	3	Code Standard	1	3	16
Redundant Empty Statement	3	Formatting	1	1	14
Unused Variables	2	Code Standard	2	1	23
File Length	2	Metric	2	3	25
Header Guard Format	2	Code Standard	1	1	13
No Copyright Statement	2	Code Standard	1	1	13
TODO Comment Format	2	Code Standard	1	1	13
Order of Evaluation	2	Code Standard	3	1	33
Do Not Include Twice	1	Code Standard	1	1	12
Include Directory in Header	1	Code Standard	2	1	22
Header Guard Must Close	1	Code Standard	2	1	22
Whitespace Before Comments	1	Formatting	1	1	12
Do Not Use Default Lambda Captures	1	Code Standard	3	2	33
C Standard Library	1	Code Standard	1	1	12
Total issues	**492**		**Total Score**		**1125**

Table 5.19: Robotiq ethercat package analysis

Package - robotiq_ethercat					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	10	Code Standard	2	2	32
Function Length	4	Metric	2	3	27
Include Order	8	Formatting	1	1	19
Avoid C System Headers	2	Code Standard	2	1	23
No Copyright Statement	2	Code Standard	1	1	13
Non-const Reference Parameters	2	Code Standard	1	1	13
Halstead Bugs	1	Metric	3	3	34
Halstead Volume	1	Metric	3	3	34
Include	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
End of Namespace Comment	1	Formatting	1	1	12
Make Constructors Explicit	1	Code Standard	1	1	12
Storage Class Before Type	1	Code Standard	1	1	12
Maintainability Index	1	Metric	3	3	34
No Redundant Variables	1	Code Standard	1	2	13
Smallest Feasible Scope	1	Code Standard	1	2	13
Total issues	**40**		**Total Score**		**383**

Table 5.20: Robotiq c model control package analysis

Package - robotiq_c_model_control					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	6	Code Standard	2	2	28
Include Order	1	Formatting	1	1	12
No Copyright Statement	3	Code Standard	1	1	14
End of Namespace Comment	2	Formatting	1	1	13
Non-const Reference Parameters	2	Code Standard	1	2	14
Include	1	Code Standard	2	1	22
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Total issues	**17**		**Total Score**		**137**

Table 5.21: Dynamic robot localization package analysis

Package - dynamic_robot_localization					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	1887	Formatting	1	1	1898
Non-const Reference Parameters	561	Code Standard	1	2	573
No Copyright Statement	207	Code Standard	1	1	218
Include Order	217	Code Standard	1	1	228
Include	114	Code Standard	2	1	135
No Header Guard	73	Code Standard	2	1	94
Function Length	73	Metric	2	3	96
Cyclomatic Complexity	52	Metric	3	3	85
Casting	12	Code Standard	3	1	43
Whitespace Before Comment Text	11	Formatting	1	1	22
Integer Types	8	Code Standard	2	2	30
Make Constructors Explicit	8	Code Standard	1	1	19
File Length	6	Metric	2	3	29
End of Namespace Comment	6	Formatting	1	1	17
Header Guard Format	6	Code Standard	1	1	17
Avoid Thread-Unsafe Functions	5	Code Standard	3	2	37
Unused Variables	5	Code Standard	2	1	26
Function Parameters	4	Metric	2	3	27
Halstead Bugs	4	Metric	3	3	37
Redundant Empty Statement	4	Formatting	1	1	15
Header Guard Must Close	3	Code Standard	2	1	24
Complex Multi-line Comments and Strings	3	Formatting	1	1	14
Halstead Volume	3	Metric	3	3	36
Do Not Use C Types	3	Code Standard	2	1	24
Order of Evaluation	2	Code Standard	3	1	33
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
TODO Comment Format	1	Formatting	1	1	12
Blank Lines In Code Blocks	1	Formatting	1	1	12
Whitespace Before Comments	1	Formatting	1	1	12
Blank Lines Before Section	1	Formatting	1	1	12
Total issues	**3283**		**Total Score**		**3881**

Table 5.22: Phoxi camera package analysis

Package - phoxi_camera					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Complex Multi-line Comments and Strings	148	Formatting	1	1	159
Integer Types	93	Code Standard	2	2	115
Non-const Reference Parameters	46	Code Standard	1	1	57
Casting	35	Code Standard	3	1	66
Function Length	25	Metric	2	3	48
Cyclomatic Complexity	17	Metric	3	3	50
Avoid C System Headers	13	Code Standard	2	1	34
No Copyright Statement	12	Code Standard	1	1	23
No Uninitialized Member Variables	12	Code Standard	1	1	23
Make Constructors Explicit	11	Code Standard	1	1	22
Include	10	Code Standard	2	1	31
Header Guard Format	10	Code Standard	1	1	21
Line Length	7	Formatting	1	1	18
Alternative Tokens	7	Code Standard	1	1	18
Halstead Bugs	6	Metric	3	3	39
Include Order	5	Code Standard	1	1	16
Order of Evaluation	5	Code Standard	3	1	36
Halstead Volume	5	Metric	3	3	38
Header Guard Must Close	5	Code Standard	2	1	26
No Unions	5	Code Standard	3	2	37
Empty Semicolon Statement	5	Code Standard	1	1	16
Do Not Use C Types	4	Code Standard	2	1	25
C-style String Constants	4	Code Standard	1	2	16
Unused Variables	3	Code Standard	2	1	24
Maximum Executable Lines of Code	2	Metric	2	3	25
File Length	2	Metric	2	3	25
Maintainability Index	2	Metric	3	3	35
Avoid Namespace Using-Directives	2	Code Standard	3	1	33
TODO Comment Format	2	Formatting	1	1	13
Include Directory in Header	1	Code Standard	2	1	22
One Command Per Line	1	Formatting	1	1	12
Smallest Feasible Scope	1	Code Standard	1	2	13
C Standard Library	1	Code Standard	1	1	12
No Boolean Vectors	1	Code Standard	3	1	32
Total issues	**508**		**Total Score**		**1181**

Table 5.23: Laserscan to pointcloud package analysis

Package - laserscan_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	125	Formatting	1	1	136
Non-const Reference Parameters	40	Code Standard	1	2	52
Function Length	25	Metric	2	3	48
Integer Types	18	Code Standard	2	2	40
No Copyright Statement	13	Code Standard	1	1	24
Casting	12	Code Standard	3	1	43
Include	8	Code Standard	2	1	29
Include Order	6	Code Standard	1	1	17
No Header Guard	6	Code Standard	2	1	27
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Halstead Bugs	2	Metric	3	3	35
Halstead Volume	2	Metric	3	3	35
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Unused Variables	1	Code Standard	2	1	22
Total issues	**267**		**Total Score**		**627**

Table 5.24: Object recognition skill server package analysis

Package - object_recognition_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	3	Code Standard	2	2	25
No Copyright Statement	3	Code Standard	1	1	14
Function Length	3	Metric	2	3	26
Include	2	Code Standard	2	1	23
Non-const Reference Parameters	2	Code Standard	1	2	14
End of Namespace Comment	2	Formatting	1	1	13
Include Order	1	Code Standard	1	1	12
No Header Guard	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Total issues	**18**		**Total Score**		**183**

Table 5.25: Mesh to pointcloud package analysis

Package - mesh_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	25	Formatting	1	1	36
Non-const Reference Parameters	20	Code Standard	1	w	32
Include Order	5	Code Standard	1	1	16
No Copyright Statement	4	Code Standard	1	1	15
End of Namespace Comment	3	Formatting	1	1	14
Function Length	3	Metric	2	3	26
Include	2	Code Standard	2	1	23
Integer Types	2	Code Standard	2	2	24
Halstead Bugs	2	Metric	3	3	35
Maintainability Index	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**68**		**Total Score**		**277**

Table 5.26: Pose to tf publisher package analysis

Package - pose_to_tf_publisher					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Line Length	17	Formatting	1	1	28
Function Length	10	Metric	2	3	33
Non-const Reference Parameters	8	Code Standard	1	1	19
Smallest Feasible Scope	8	Code Standard	1	2	20
Float Accuracy	4	Code Standard	3	1	35
No Copyright Statement	3	Code Standard	1	1	14
Cyclomatic Complexity	3	Metric	3	3	36
Integer Types	2	Code Standard	2	2	24
Include	2	Code Standard	2	1	23
Halstead Bugs	2	Metric	3	3	35
File Length	1	Metric	2	3	24
Include Order	1	Code Standard	1	1	12
Order of Evaluation	1	Code Standard	3	1	32
Halstead Volume	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Make Constructors Explicit	1	Code Standard	1	1	12
Total issues	**65**		**Total Score**		**404**

Table 5.27: Octomap server package analysis

Package - octomap_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	55	Code Standard	2	2	77
Function Length	46	Metric	2	3	69
Non-const Reference Parameters	20	Code Standard	1	2	32
Casting	19	Code Standard	3	1	50
Include	13	Code Standard	2	1	34
TODO Comment Format	12	Formatting	1	1	23
Avoid Namespace Using-Directives	10	Code Standard	3	2	42
Cyclomatic Complexity	9	Metric	3	3	42
End of Namespace Comment	7	Formatting	1	1	18
Header Guard Format	6	Code Standard	1	1	17
Include Order	4	Code Standard	1	1	15
Redundant Empty Statement	4	Formatting	1	1	15
Order of Evaluation	4	Code Standard	3	1	35
Make Constructors Explicit	4	Code Standard	1	1	15
Header Guard Must Close	3	Code Standard	2	1	24
Halstead Volume	3	Metric	3	3	36
Halstead Bugs	3	Metric	3	3	36
Smallest Feasible Scope	2	Code Standard	1	2	14
Line Length	1	Formatting	1	1	12
File Length	1	Metric	2	3	24
Unused Variables	1	Code Standard	2	1	22
No Boolean Vectors	1	Code Standard	3	1	32
No Redundant Variables	1	Code Standard	1	1	12
Total issues	**229**		**Total Score**		**696**

Table 5.28: PCL conversions package analysis

Package - pcl_conversions					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	77	Code Standard	1	2	89
Function Length	16	Metric	2	3	39
Include Order	2	Code Standard	1	1	13
Cyclomatic Complexity	2	Metric	3	3	35
Halstead Bugs	2	Metric	3	3	35
Header Guard Format	2	Code Standard	1	1	13
End of Namespace Comment	2	Formatting	1	1	13
Include	1	Code Standard	2	1	22
Halstead Volume	1	Metric	3	3	34
File Length	1	Metric	2	3	24
Header Guard Must Close	1	Code Standard	2	1	22
Integer Types	1	Code Standard	1	2	13
No Copyright Statement	1	Code Standard	1	1	12
Unused Variables	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Total issues	**111**		**Total Score**		**420**

This first iteration allowed the elimination of 14 types of issues, from the 66 in the initial analysis. This resulted in a total of 52 issues at the end of the first iteration, caused by the reduction of the formatting from 24 to 10. After the first iteration, the total number of issues decreased from the initial 28043 to 5357 issues, after the remaining 22686 issues were solved. Both formatting and code standard issues decreased. However, the metric issues increased. What led to this were the changes made to respect the line length, which cause an increase in the use of vertical lines. This increase caused more functions to have over 40 lines of code, which triggered more metric issues. Overall, this was still a successful iteration, since it allowed to solve around 80 % of the initial issues and the majority of the formatting issues. It is important to notice that, on the results of this iteration, the issues related with curly braces were ignored. This decision had to do with the fact that, when following Google C++ Style, the curly braces are in most cases in different places from where they would be if ROS C++ style had been used. Therefore, since a style was correctly being used, there was no reason to consider those issues. The average severity and the average effort to solve also increased their values after this iteration, from 1.61 to 1.85 and from 1.47 to 1.68 respectively. The fact that most of the issues eliminated were not a threat and were also easy to fix led to this change.

5.5 Second Iteration

For the second iteration, one of the issues with a higher score was the *line length*. Since the automatic formatting did not solve this, the source code was analysed to understand the root of this issue. There were two explanations: functions with long names could not be solved and comments with several repeated characters to separate different parts of the code. Regardless, this could be solved by reducing the number of repeated characters without removing the code separation.

Another issue with a high count was the *Non-const Reference Parameters*. This issue was caused by variables being passed by reference, but not using the keyword *cont* as recommended by the Google C++ style guide. This issue has two possible solutions. The first is to use the keyword *const* if the variable does not need to be changed inside the function and the other, which requires more effort, is to pass by a pointer and to change the code according to this demand. However, since the second solution was the one that needed to be applied more, it was opted to leave the code as is, to avoid cross package errors that could be hard to track. Also, this issue did not represent a threat.

The issues of the type *Integer types* were also among the issues with a higher count. These issues were mostly triggered by the use of the type *size_t*, but had also others, such as the use of the type *short* or *long*. The usage of the type *size_t* is allowed by Google C++ style guide when it is appropriate, which was the case, and for that reason it was not changed. When types such as *short* were being used, they were replaced by size specific types, such as *int16_t* to replace short.

In these iterations, issues with *whitespaces*, issues with *copyright* (by adding a copyright statement to each file) and issues with *constructors* (by making constructors with single argument explicit) were also fixed. Furthermore, issues related with *casting* were also solved during this iteration. However, HAROS still identified 2 casting issues. Yet, while inspecting the code, it was found that these were not casting issues, but were still being registered as such.

Finally, in this iteration the issues with the *floating point* were solved. These issues were caused by float point expressions that were expecting a exact equality, which is not complying with MISRA C++ guidelines, deeming it unsafe. The solution for these issues was to rewrite the expressions in a way that did not test equality directly and that was compliant with the guidelines.

On the following tables, the results that this iteration produced in each package, in its issues, and its score are presented.

Table 5.29: Move arm skill server package analysis

Package - move_arm_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	22	Code Standard	2	2	44
Function Length	8	Metric	2	3	31
Include	5	Code Standard	2	2	27
Cyclomatic Complexity	4	Metric	3	3	37
File Length	1	Metric	2	3	24
No Header Guard	1	Code Standard	2	1	22
Non-const Reference Parameters	1	Code Standard	1	2	13
Total issues	**42**		**Total Score**		**198**

Table 5.30: Arm action controller package analysis

Package - arm_action_controller					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	12	Code Standard	2	2	34
Function Length	7	Metric	2	3	30
Non-const Reference Parameters	5	Code Standard	1	2	17
Include	3	Code Standard	2	1	24
Cyclomatic Complexity	3	Metric	3	3	36
Function Parameters	2	Metric	3	3	35
Maintainability Index	2	Metric	3	3	35
Include Order	1	Code Standard	1	1	12
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**38**		**Total Score**		**313**

Table 5.31: Arm interface package analysis

Package - arm_interface					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	21	Code Standard	2	2	43
Function Length	21	Metric	2	3	44
Include	20	Code Standard	2	1	41
Non-const Reference Parameters	15	Code Standard	1	2	27
Maintainability Index	7	Metric	3	3	40
Include Order	6	Code Standard	1	1	17
Halstead Bugs	6	Metric	3	3	39
No Header Guard	6	Code Standard	2	1	27
Line Length	5	Formatting	1	1	16
Halstead Volume	5	Metric	3	3	38
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Avoid Thread-Unsafe Functions	1	Code Standard	3	2	33
Total issues	**120**		**Total Score**		**428**

Table 5.32: Ur modern driver package analysis

Package - ur_modern_driver					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	194	Code Standard	1	2	206
Integer Types	44	Code Standard	2	2	66
Include	40	Code Standard	2	1	61
No Header Guard	35	Code Standard	2	1	56
Function Length	17	Metric	2	3	40
Avoid Unapproved Headers	16	Code Standard	1	3	29
Avoid C System Headers	13	Code Standard	2	1	34
Do Not Use C Types	17	Code Standard	2	2	39
Halstead Bugs	12	Metric	3	3	45
Halstead Volume	8	Metric	3	3	41
Avoid Rvalue References	9	Code Standard	3	2	38
Avoid Namespace Using-Directives	6	Code Standard	3	2	38
Maximum Executable Lines of Code	3	Metric	2	3	26
Maintainability Index	3	Metric	3	3	36
Avoid Unapproved Classes and Functions	3	Code Standard	1	3	16
Redundant Empty Statement	3	Formatting	1	1	14
Unused Variables	2	Code Standard	2	1	23
File Length	2	Metric	2	3	25
Header Guard Format	2	Code Standard	1	1	13
No Copyright Statement	2	Code Standard	1	1	13
TODO Comment Format	2	Code Standard	1	1	13
Order of Evaluation	2	Code Standard	3	1	33
Do Not Include Twice	1	Code Standard	1	1	12
Include Directory in Header	1	Code Standard	2	1	22
Header Guard Must Close	1	Code Standard	2	1	22
Whitespace Before Comments	1	Formatting	1	1	12
Do Not Use Default Lambda Captures	1	Code Standard	3	2	33
C Standard Library	1	Code Standard	1	1	12
Total issues	**441**		**Total Score**		**1021**

Table 5.33: Robotiq ethercat package analysis

Package - robotiq_ethercat					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	10	Code Standard	2	2	32
Function Length	4	Metric	2	3	27
Include Order	8	Code Standard	1	1	19
Avoid C System Headers	2	Code Standard	2	1	23
Non-const Reference Parameters	2	Code Standard	1	2	14
Halstead Bugs	1	Metric	3	3	34
Halstead Volume	1	Metric	3	3	34
Include	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Storage Class Before Type	1	Code Standard	1	1	12
Maintainability Index	1	Metric	3	3	34
No Redundant Variables	1	Code Standard	1	2	13
Smallest Feasible Scope	1	Code Standard	1	2	13
Total issues	**36**		**Total Score**		**346**

Table 5.34: Robotiq c model control package analysis

Package -robotiq_c_model_control					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	6	Code Standard	2	2	28
Non-const Reference Parameters	2	Code Standard	1	2	13
Include Order	1	Code Standard	1	1	12
Include	1	Code Standard	2	1	22
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Total issues	**12**		**Total Score**		**110**

Table 5.35: Dynamic robot localization package analysis

Package - dynamic_robot_localization					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	561	Code Standard	1	2	573
Include Order	217	Code Standard	1	1	228
Include	114	Code Standard	2	1	135
No Header Guard	73	Code Standard	2	1	94
Function Length	73	Metric	2	3	96
Line Length	61	Formatting	1	1	72
Cyclomatic Complexity	52	Metric	3	3	85
Integer Types	8	Code Standard	2	2	30
File Length	6	Metric	2	3	29
Header Guard Format	6	Code Standard	1	1	17
Avoid Thread-Unsafe Functions	5	Code Standard	3	2	37
Unused Variables	5	Code Standard	2	1	26
Function Parameters	4	Metric	2	3	27
Halstead Bugs	4	Metric	3	3	37
Redundant Empty Statement	4	Formatting	1	1	15
Header Guard Must Close	3	Code Standard	2	1	24
Complex Multi-line Comments and Strings	3	Formatting	1	1	14
Halstead Volume	3	Metric	3	3	36
Do Not Use C Types	3	Code Standard	2	1	24
Casting	2	Code Standard	3	1	33
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
TODO Comment Format	1	Formatting	1	1	12
Do Not Use C Types	3	Code Standard	2	1	24
Order of Evaluation	2	Code Standard	3	1	33
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
TODO Comment Format	1	Formatting	1	1	12
Total issues	**1210**		**Total Score**		**1600**

Table 5.36: Phoxi camera package analysis

Package - phoxi_camera					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Complex Multi-line Comments and Strings	148	Formatting	1	1	159
Integer Types	93	Code Standard	2	2	115
Non-const Reference Parameters	46	Code Standard	1	1	57
Function Length	25	Metric	2	3	48
Cyclomatic Complexity	17	Metric	3	3	50
Avoid C System Headers	13	Code Standard	2	1	34
No Uninitialized Member Variables	12	Code Standard	1	1	23
Include	10	Code Standard	2	1	31
Header Guard Format	10	Code Standard	1	1	21
Line Length	7	Formatting	1	1	18
Alternative Tokens	7	Code Standard	1	1	18
Halstead Bugs	6	Metric	3	3	39
Include Order	5	Code Standard	1	1	16
Order of Evaluation	1	Code Standard	3	1	32
Halstead Volume	5	Metric	3	3	38
Header Guard Must Close	5	Code Standard	2	1	26
No Unions	5	Code Standard	3	2	37
Empty Semicolon Statement	5	Code Standard	1	1	16
Do Not Use C Types	4	Code Standard	2	1	25
C-style String Constants	4	Code Standard	1	2	6
Unused Variables	3	Code Standard	2	1	24
Maximum Executable Lines of Code	2	Metric	2	3	25
File Length	2	Metric	2	3	25
Maintainability Index	2	Metric	3	3	35
Avoid Namespace Using-Directives	2	Code Standard	3	2	33
TODO Comment Format	2	Formatting	1	1	13
Include Directory in Header	1	Code Standard	2	1	22
One Command Per Line	1	Formatting	1	1	12
Smallest Feasible Scope	1	Code Standard	1	2	13
C Standard Library	1	Code Standard	1	1	12
No Boolean Vectors	1	Code Standard	3	1	32
Total issues	**446**		**Total Score**		**1065**

Table 5.37: Laserscan to pointcloud package analysis

Package - laserscan_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	40	Code Standard	1	2	52
Function Length	25	Metric	2	3	48
Integer Types	18	Code Standard	2	2	40
Include	8	Code Standard	2	1	29
Include Order	6	Code Standard	1	1	17
No Header Guard	6	Code Standard	2	1	27
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Halstead Bugs	2	Metric	3	3	35
Halstead Volume	2	Metric	3	3	35
Avoid C System Headers	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Unused Variables	1	Code Standard	2	1	22
Total issues	**117**		**Total Score**		**424**

Table 5.38: Object recognition skill server package analysis

Package - object_recognition_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	3	Code Standard	2	2	25
Function Length	3	Metric	2	3	26
Include	2	Code Standard	2	1	23
Non-const Reference Parameters	2	Code Standard	1	2	14
Include Order	1	Code Standard	1	1	12
No Header Guard	1	Code Standard	2	1	22
Cyclomatic Complexity	1	Metric	3	3	34
Total issues	**13**		**Total Score**		**156**

Table 5.39: Mesh to pointcloud package analysis

Package - mesh_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	20	Code Standard	1	2	32
Include Order	5	Code Standard	1	1	16
Function Length	3	Metric	2	3	26
Include	2	Code Standard	2	1	23
Integer Types	2	Code Standard	2	2	24
Halstead Bugs	2	Metric	3	3	35
Maintainability Index	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Include Order	2	Code Standard	1	1	13
Total issues	**36**		**Total Score**		**212**

Table 5.40: Pose to tf publisher package analysis

Package - pose_to_tf_publisher					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Function Length	11	Metric	2	3	34
Non-const Reference Parameters	8	Code Standard	1	2	20
Smallest Feasible Scope	8	Code Standard	1	2	20
Cyclomatic Complexity	3	Metric	3	3	36
Integer Types	2	Code Standard	2	2	24
Include	2	Code Standard	2	1	23
Halstead Bugs	2	Metric	3	3	35
File Length	1	Metric	2	3	24
Include Order	2	Code Standard	1	1	13
Halstead Volume	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**41**		**Total Score**		**285**

Table 5.41: Octomap server package analysis

Package - octomap_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	55	Code Standard	2	2	77
Function Length	46	Metric	2	3	69
Non-const Reference Parameters	20	Code Standard	1	2	32
Include	13	Code Standard	2	1	34
TODO Comment Format	12	Formatting	1	1	23
Avoid Namespace Using-Directives	10	Code Standard	3	2	42
Cyclomatic Complexity	9	Metric	3	3	42
Header Guard Format	6	Code Standard	1	1	17
Include Order	4	Code Standard	1	1	15
Redundant Empty Statement	4	Formatting	1	1	15
Order of Evaluation	4	Code Standard	3	1	35
Header Guard Must Close	3	Code Standard	2	1	24
Halstead Volume	3	Metric	3	3	36
Halstead Bugs	3	Metric	3	3	36
Smallest Feasible Scope	2	Code Standard	1	2	14
Line Length	1	Formatting	1	1	12
File Length	1	Metric	2	3	24
Unused Variables	1	Code Standard	2	1	22
No Boolean Vectors	1	Code Standard	3	1	32
No Redundant Variables	1	Code Standard	1	1	12
Total issues	**199**		**Total Score**		**613**

Table 5.42: PCL conversions package analysis

Package - pcl_conversions					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	77	Code Standard	1	2	88
Function Length	16	Metric	2	3	39
Include Order	2	Code Standard	1	1	13
Cyclomatic Complexity	2	Metric	3	3	35
Halstead Bugs	2	Metric	3	3	35
Header Guard Format	2	Code Standard	1	1	13
Include	1	Code Standard	2	1	22
Halstead Volume	1	Metric	3	3	34
File Length	1	Metric	2	3	24
Header Guard Must Close	1	Code Standard	2	1	22
Integer Types	1	Code Standard	1	2	13
Unused Variables	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Total issues	**108**		**Total Score**		**395**

Overall, 2498 issues were solved in this iteration, which reduced the total of issues to solve to 2859 in the end of this iteration. The total number of issues types still unsolved was also reduced from the initial 52 to 45. Moreover, the formatting issues decreased from 10 to 5 and the code standard issues from 34 to 32. However, the average severity and average effort to solve increased from 1.85 to 1.95 and from 1.68 to 1.83 respectively. This is justified by the fixing of more issues with lower severity and lower effort to solve. Nevertheless, this was also a successful iteration, since it led to a reduction of around 50% issues produced in the previous iteration.

5.6 Third Iteration

This third and final iteration focused on solving issues related to cyclomatic complexity, functions that were not thread safe and also analysed other issues to understand their causes.

Among the metrics, the cyclomatic complexity is the easiest to change and improve. Despite that, it does not mean that it is a simple issue to fix. Some functions with high cyclomatic complexity are impossible to do in a less complex way, as their purpose is to verify a set of conditions that can not be changed. Others are simply just too complex, and it is therefore very risky to change them. Since this code belongs to robotic software, it is responsible for the implementation of very specialized and complex features, such as computer vision algorithms. Areas like this require some specialized expertise to implement those algorithms, which difficults the task of changing these algorithms. However, for some of these functions, it is possible to understand their purpose

without deep knowledge on the area. For some of those, it is possible to achieve the same result using less complex ways. During this iteration, it was possible to reduce the cyclomatic complexity of functions with cyclomatic complexity as high as 17. Above that value, it was opted no to change them since the code was functional and the probability to introduce errors was high. For these more complex functions, it is recommended that a developer with higher expertise in the area to remake them. Some cases studys where the cyclomatic complexity as reduced are presented in the next figures.

For the first case a loop was created to verify the each element of the matrix. This change allow decrease the cyclomatic complexity from 17 to 4.

```
bool isTransformValid(const Eigen::Matrix<Scalar, 4, 4>& transform) {
  if (!pcl_isfinite(transform(0, 0)) || !pcl_isfinite(transform(0, 1)) ||
      !pcl_isfinite(transform(0, 2)) || !pcl_isfinite(transform(0, 3)) ||
      !pcl_isfinite(transform(1, 0)) || !pcl_isfinite(transform(1, 1)) ||
      !pcl_isfinite(transform(1, 2)) || !pcl_isfinite(transform(1, 3)) ||
      !pcl_isfinite(transform(2, 0)) || !pcl_isfinite(transform(2, 1)) ||
      !pcl_isfinite(transform(2, 2)) || !pcl_isfinite(transform(2, 3)) ||
      !pcl_isfinite(transform(3, 0)) || !pcl_isfinite(transform(3, 1)) ||
      !pcl_isfinite(transform(3, 2)) || !pcl_isfinite(transform(3, 3))) {
    return false;
  }
  return true;
}
```

```
bool isTransformValid(const Eigen::Matrix<Scalar, 4, 4>& transform) {
  for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
      if (!pcl_isfinite(transform(i, j))) {
        return false;
      }
    }
  }
  return true;
}
```

Figure 5.1: Case study 1 - Before (top) and After (bottom)

The approach of the second case study takes advantage of the fact a C++ enumeration is used. Each element of the enumeration corresponds to a integer value by default. This values are consecutive, so the switch was replaced by a array which contained the correspondent string to each element of the enumeration. This change allowed the reduction of the cyclomatic complexity from 19 to 1.

```
static std::string SensorDataProcessingStatusToStr(
    const SensorDataProcessingStatus& status) {
  switch (status) {
    case ExceptionRaised:
      return "ExceptionRaised";
    case FailedInitialPoseEstimation:
      return "FailedInitialPoseEstimation";
    case FailedNormalEstimation:
      return "FailedNormalEstimation";
    case FailedPoseEstimation:
      return "FailedPoseEstimation";
    case FailedTFTransform:
      return "FailedTFTransform";
    case FillingCircularBufferWithMsgsFromAllTopics:
      return "FillingCircularBufferWithMsgsFromAllTopics";
    case FirstPointCloudInSlamMode:
      return "FirstPointCloudInSlamMode";
    case MinimumElapsedTimeSinceLastPointCloudNotReached:
      return "MinimumElapsedTimeSinceLastPointCloudNotReached";
    case MissingReferencePointCloud:
      return "MissingReferencePointCloud";
    case PointCloudAgeHigherThanMaximum:
      return "PointCloudAgeHigherThanMaximum";
    case PointCloudDiscarded:
      return "PointCloudDiscarded";
    case PointCloudFilteringFailed:
      return "PointCloudFilteringFailed";
    case PointCloudOlderThanLastPointCloudReceived:
      return "PointCloudOlderThanLastPointCloudReceived";
    case PointCloudSubscribersDisabled:
      return "PointCloudSubscribersDisabled";
    case PointCloudWithoutTheMinimumNumberOfRequiredPoints:
      return "PointCloudWithoutTheMinimumNumberOfRequiredPoints";
    case PoseEstimationRejectedByTransformationValidators:
      return "PoseEstimationRejectedByTransformationValidators";
    case SuccessfulPoseEstimation:
      return "SuccessfulPoseEstimation";
    case WaitingForSensorData:
      return "WaitingForSensorData";
  }
  return "";
}
```

Figure 5.2: Case study 2 - Before

```
static std::string SensorDataProcessingStatusToStr(
    const SensorDataProcessingStatus& status) {
  std::vector<std::string> stat = {
      "ExceptionRaised",
      "FailedInitialPoseEstimation",
      "FailedNormalEstimation",
      "FailedPoseEstimation",
      "FailedTFTransform",
      "FillingCircularBufferWithMsgsFromAllTopics",
      "FirstPointCloudInSlamMode",
      "MinimumElapsedTimeSinceLastPointCloudNotReached",
      "MissingReferencePointCloud",
      "PointCloudAgeHigherThanMaximum",
      "PointCloudDiscarded",
      "PointCloudFilteringFailed",
      "PointCloudOlderThanLastPointCloudReceived",
      "PointCloudSubscribersDisabled",
      "PointCloudWithoutTheMinimumNumberOfRequiredPoints",
      "PoseEstimationRejectedByTransformationValidators",
      "SuccessfulPoseEstimation",
      "WaitingForSensorData"};
  return stat[static_cast<int>(status)];
}
```

Figure 5.3: Case study 2 -After

During this iteration, the issues of *thread safety functions* were solved by replacing the function *rand* by an alternative function that was thread safe: the function *_rand_r*. The other issue that was taken care of was the *include* issue. To solve it, the needed includes for the code used on those files were added.

While analysing some of the remaining issues, it was noted that HAROS found a false positive issue. That issue was the *Alternative Tokens*. HAROS found the words *and* and *or*, but they were not being used as a replacement for *&&* and || respectively, which would be a violation of Google C++ style guide. This words were merely used on a comment, which is not a issue.

The effect that this iteration had on each package is presented in the following tables.

Table 5.43: Move arm skill server package analysis

Package - move_arm_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	22	Code Standard	2	2	44
Function Length	8	Metric	2	3	31
Cyclomatic Complexity	4	Metric	3	3	37
File Length	1	Metric	2	3	24
No Header Guard	1	Code Standard	2	1	22
Non-const Reference Parameters	1	Code Standard	1	2	13
Total issues	**37**		**Total Score**		**171**

Table 5.44: Arm action controller package analysis

Package - arm_action_controller					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	17	Code Standard	2	2	39
Function Length	6	Metric	2	3	29
Non-const Reference Parameters	5	Code Standard	1	2	17
Cyclomatic Complexity	2	Metric	3	3	35
Function Parameters	2	Metric	3	3	35
Maintainability Index	2	Metric	3	3	35
Include Order	1	Code Standard	1	1	12
Halstead Volume	1	Metric	3	3	34
Halstead Bugs	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**38**		**Total Score**		**292**

Table 5.45: Arm interface package analysis

Package - arm_interface					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	21	Code Standard	2	2	43
Function Length	21	Metric	2	3	44
Non-const Reference Parameters	15	Code Standard	1	2	26
Include Order	6	Code Standard	1	1	17
Halstead Bugs	6	Metric	3	3	39
No Header Guard	6	Code Standard	2	1	27
Line Length	5	Formatting	1	1	16
Halstead Volume	5	Metric	3	3	38
Maintainability Index	4	Metric	3	3	37
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Total issues	**96**		**Total Score**		**351**

Table 5.46: Ur modern driver package analysis

Package - ur_modern_driver					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	194	Code Standard	1	1	205
Integer Types	44	Code Standard	2	2	66
No Header Guard	35	Code Standard	2	1	56
Function Length	17	Metric	2	3	40
Avoid Unapproved Headers	16	Code Standard	1	3	29
Do Not Use C Types	17	Code Standard	2	2	39
Halstead Bugs	11	Metric	3	3	44
Halstead Volume	8	Metric	3	3	41
Avoid Rvalue References	8	Code Standard	3	3	38
Avoid Namespace Using-Directives	6	Code Standard	3	2	38
Maximum Executable Lines of Code	3	Metric	2	3	26
Maintainability Index	3	Metric	3	3	36
Avoid Unapproved Classes and Functions	3	Code Standard	1	3	16
Redundant Empty Statement	3	Formatting	1	1	14
Unused Variables	2	Code Standard	2	1	23
File Length	2	Metric	2	3	25
Header Guard Format	2	Code Standard	1	1	13
No Copyright Statement	2	Code Standard	1	1	13
TODO Comment Format	2	Code Standard	1	1	13
Order of Evaluation	2	Code Standard	3	1	33
Do Not Include Twice	1	Code Standard	1	1	12
Include Directory in Header	1	Code Standard	2	1	22
Header Guard Must Close	1	Code Standard	2	1	22
Whitespace Before Comments	1	Formatting	1	1	12
Do Not Use Default Lambda Captures	1	Code Standard	3	2	33
C Standard Library	1	Code Standard	1	1	12
Total issues	**386**		**Total Score**		**924**

Table 5.47: Robotiq ethercat package analysis

Package - robotiq_ethercat					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	10	Code Standard	2	2	32
Include Order	8	Code Standard	1	1	19
Function Length	4	Metric	2	3	27
Non-const Reference Parameters	2	Code Standard	1	2	14
Halstead Bugs	1	Metric	3	3	34
Halstead Volume	1	Metric	3	3	34
Cyclomatic Complexity	1	Metric	3	3	34
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Storage Class Before Type	1	Code Standard	1	1	12
Maintainability Index	1	Metric	3	3	34
No Redundant Variables	1	Code Standard	1	2	13
Smallest Feasible Scope	1	Code Standard	1	2	13
Total issues	**33**		**Total Score**		**301**

Table 5.48: Robotiq c model control package analysis

Package - robotiq_c_model_control					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	6	Code Standard	2	2	28
Non-const Reference Parameters	2	Code Standard	1	2	14
Include Order	1	Code Standard	1	1	12
Header Guard Format	1	Code Standard	1	1	12
Header Guard Must Close	1	Code Standard	2	1	22
Total issues	**11**		**Total Score**		**88**

Table 5.49: Dynamic robot localization package analysis

Package - dynamic_robot_localization					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	561	Code Standard	1	2	573
Include Order	217	Code Standard	1	1	228
No Header Guard	73	Code Standard	2	1	94
Function Length	73	Metric	2	3	96
Line Length	61	Formatting	1	1	72
Cyclomatic Complexity	46	Metric	3	3	79
Integer Types	8	Code Standard	2	2	30
File Length	6	Metric	2	3	29
Header Guard Format	6	Code Standard	1	1	17
Unused Variables	5	Code Standard	2	1	26
Function Parameters	4	Metric	2	3	27
Halstead Bugs	4	Metric	3	3	37
Redundant Empty Statement	4	Formatting	1	1	15
Header Guard Must Close	3	Code Standard	2	1	24
Complex Multi-line Comments and Strings	3	Formatting	1	1	14
Halstead Volume	3	Metric	3	3	36
Do Not Use C Types	3	Code Standard	2	1	24
Casting	2	Code Standard	3	1	33
Maintainability Index	1	Metric	3	3	34
TODO Comment Format	1	Formatting	1	1	12
Total issues	**1084**		**Total Score**		**1500**

Table 5.50: Phoxi camera package analysis

Package - phoxi_camera					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Complex Multi-line Comments and Strings	148	Formatting	1	1	159
Integer Types	93	Code Standard	2	2	115
Non-const Reference Parameters	46	Code Standard	1	2	58
Function Length	25	Metric	2	3	48
Cyclomatic Complexity	17	Metric	3	3	50
No Uninitialized Member Variables	12	Code Standard			12
Header Guard Format	10	Code Standard	1	1	21
Avoid C System Headers	7	Code Standard	2	1	28
Line Length	7	Formatting	1	1	18
Alternative Tokens	7	Code Standard	1	1	18
Halstead Bugs	6	Metric	3	3	39
Include Order	5	Code Standard	1	1	16
Halstead Volume	5	Metric	3	3	38
Header Guard Must Close	5	Code Standard	2	1	26
No Unions	5	Code Standard	3		35
Empty Semicolon Statement	5	Code Standard	1	1	16
Do Not Use C Types	4	Code Standard	2	1	25
C-style String Constants	4	Code Standard	1	2	16
Unused Variables	3	Code Standard	2	1	24
Maximum Executable Lines of Code	2	Metric	2	3	25
File Length	2	Metric	2	3	25
Maintainability Index	2	Metric	3	3	35
Avoid Namespace Using-Directives	2	Code Standard	3	1	33
TODO Comment Format	2	Formatting	1	1	13
Order of Evaluation	1	Code Standard	3	1	32
Include Directory in Header	1	Code Standard	2	1	22
One Command Per Line	1	Formatting	1	1	12
Smallest Feasible Scope	1	Code Standard	1	2	13
C Standard Library	1	Code Standard	1	1	12
No Boolean Vectors	1	Code Standard	3	1	32
Total issues	**430**		**Total Score**		**1016**

Table 5.51: Laserscan to pointcloud package analysis

Package - laserscan_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	40	Code Standard	1	2	52
Function Length	25	Metric	2	3	48
Integer Types	18	Code Standard	2	2	40
Include Order	6	Code Standard	1	1	17
No Header Guard	6	Code Standard	2	1	27
Cyclomatic Complexity	4	Metric	3	3	37
Function Parameters	3	Metric	2	3	26
Halstead Bugs	2	Metric	3	3	35
Halstead Volume	2	Metric	3	3	35
Maintainability Index	1	Metric	3	3	34
Unused Variables	1	Code Standard	2	1	22
Total issues	**108**		**Total Score**		**373**

Table 5.52: Object recognition skill server package analysis

Package - object_recognition_skill_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	3	Code Standard	2	2	25
Function Length	3	Metric	2	3	26
Non-const Reference Parameters	2	Code Standard	1	2	14
Include Order	1	Code Standard	1	1	12
No Header Guard	1	Code Standard	2	1	22
Total issues	**10**		**Total Score**		**99**

Table 5.53: Mesh to pointcloud package analysis

Package - mesh_to_pointcloud					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	20	Code Standard	1	2	32
Include Order	5	Code Standard	1	1	16
Function Length	3	Metric	2	3	26
Integer Types	2	Code Standard	2	2	24
Halstead Bugs	2	Metric	3	3	35
Maintainability Index	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**34**		**Total Score**		**189**

Table 5.54: Pose to tf publisher package analysis

Package - pose_to_tf_publisher					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	12	Code Standard	1	2	24
Function Length	11	Metric	2	3	34
Smallest Feasible Scope	8	Code Standard	1	2	20
Integer Types	2	Code Standard	2	2	24
Halstead Bugs	2	Metric	3	3	35
Include Order	2	Code Standard	1	1	13
File Length	1	Metric	2	3	24
Halstead Volume	1	Metric	3	3	34
No Header Guard	1	Code Standard	2	1	22
Total issues	**40**		**Total Score**		**230**

Table 5.55: Octomap server package analysis

Package - octomap_server					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Integer Types	55	Code Standard	2	2	77
Function Length	46	Metric	2	3	69
Non-const Reference Parameters	20	Code Standard	1	2	32
TODO Comment Format	12	Formatting	1	1	23
Avoid Namespace Using-Directives	10	Code Standard	3	2	42
Cyclomatic Complexity	9	Metric	3	3	42
Header Guard Format	6	Code Standard	1	1	17
Include Order	4	Code Standard	1	1	15
Redundant Empty Statement	4	Formatting	1	1	15
Order of Evaluation	4	Code Standard	3	1	35
Header Guard Must Close	3	Code Standard	2	1	24
Halstead Volume	3	Metric	3	3	36
Halstead Bugs	3	Metric	3	3	36
Smallest Feasible Scope	2	Code Standard	1	2	14
Line Length	1	Formatting	1	1	12
File Length	1	Metric	2	3	24
Unused Variables	1	Code Standard	2	1	22
No Boolean Vectors	1	Code Standard	3	1	32
No Redundant Variables	1	Code Standard	1		11
Total issues	**186**		**Total Score**		**578**

Table 5.56: PCL conversions package analysis

Package - pcl_conversions					
Issue	**#**	**Type**	**Severity**	**Effort to Solve**	**Score**
Non-const Reference Parameters	77	Code Standard	1	2	89
Function Length	16	Metric	2	3	39
Include Order	2	Code Standard	1	1	13
Cyclomatic Complexity	2	Metric	3	3	35
Halstead Bugs	2	Metric	3	3	35
Header Guard Format	2	Code Standard	1	1	13
Halstead Volume	1	Metric	3	3	34
File Length	1	Metric	2	3	24
Header Guard Must Close	1	Code Standard	2	1	22
Integer Types	1	Code Standard	1	2	13
Unused Variables	1	Code Standard	2	1	22
Maintainability Index	1	Metric	3	3	34
Total issues	**107**		**Total Score**		**373**

This last iteration could not solve as many issues as the previous ones, but most of the issues solved on this iteration were harder to solve. Most of the issues solved on this iteration were also more severe, which reflected on the decrease in the average severity. On this iteration, 256 issues were solved. This led to a decrease in the total number of issues from 2859 to 2603 at the end of this iterations. In this iteration, 2 code standard issues were also eliminated, reducing the total type of issues to 43 and the code standard issues to 30.

Finally, the results of the initial analysis and the following iterations are summarized in Table 5.57 and in Figure 5.4.

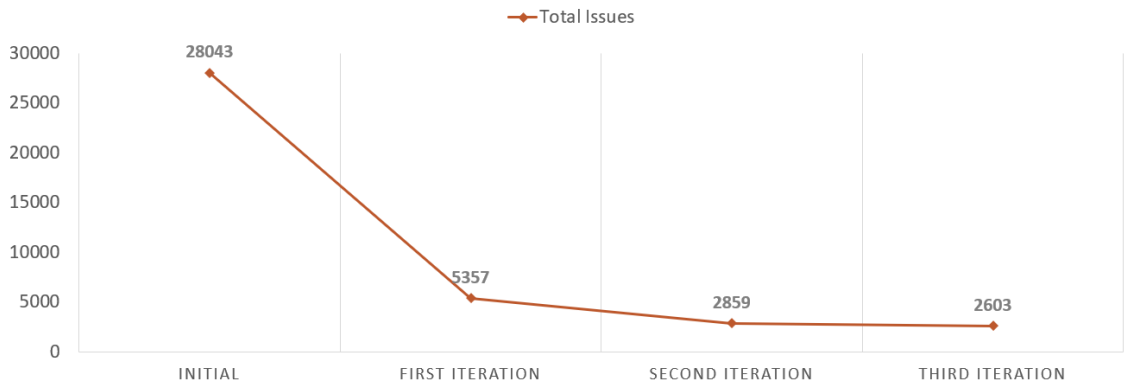

Figure 5.4: Evolution of number of issues with the iterations

Table 5.57: Results of initial analysis and the following iterations

		Types of Issues	Issues	Average Severity	Average Effort to Solve	Total Score
Initial	Formatting	24	24511	1.61	1.47	34414
	Code Standard	34	3175			
	Metric	8	356			
	Total	66	28043			
First	Formatting	10	2327	1.85	1.68	10545
	Code Standard	34	2288			
	Metric	8	478			
	Total	52	5357			
Second	Formatting	5	253	1.95	1.83	7267
	Code Standard	32	2126			
	Metric	8	480			
	Total	45	2859			
Third	Formatting	5	253	1.93	1.90	6485
	Code Standard	30	1883			
	Metric	8	467			
	Total	43	2603			

5.7 Architecture Analysis

The architecture analysis is the differentiator feature that separates the HAROS framework from the remaining static analysis tools. For this feature, it is necessary to inform HAROS which launch files should be analysed. Then, with that information, HAROS extracts the nodes that are being

launched by that file and the arguments that are being passed during the launch. However, it is not capable of finding a node that is being launched conditionally, which is very unfortunate. The developers at CRIIS are adopting a methodology where the launch of each sub-system of the mobile manipulator is conditional. This means that, in the future, if HAROS is still not capable of detecting condition launches, the architecture analysis will not be possible without changing the launch files.

So, on the yaml file that configures the analysis made by HAROS, the launch files for the TM, locate skill, move arm skill, and gripper skill were passed. This resulted in the architecture present in Figure 5.5. The white circles represent the discovered nodes. It is possible to see that no topics were discovered.

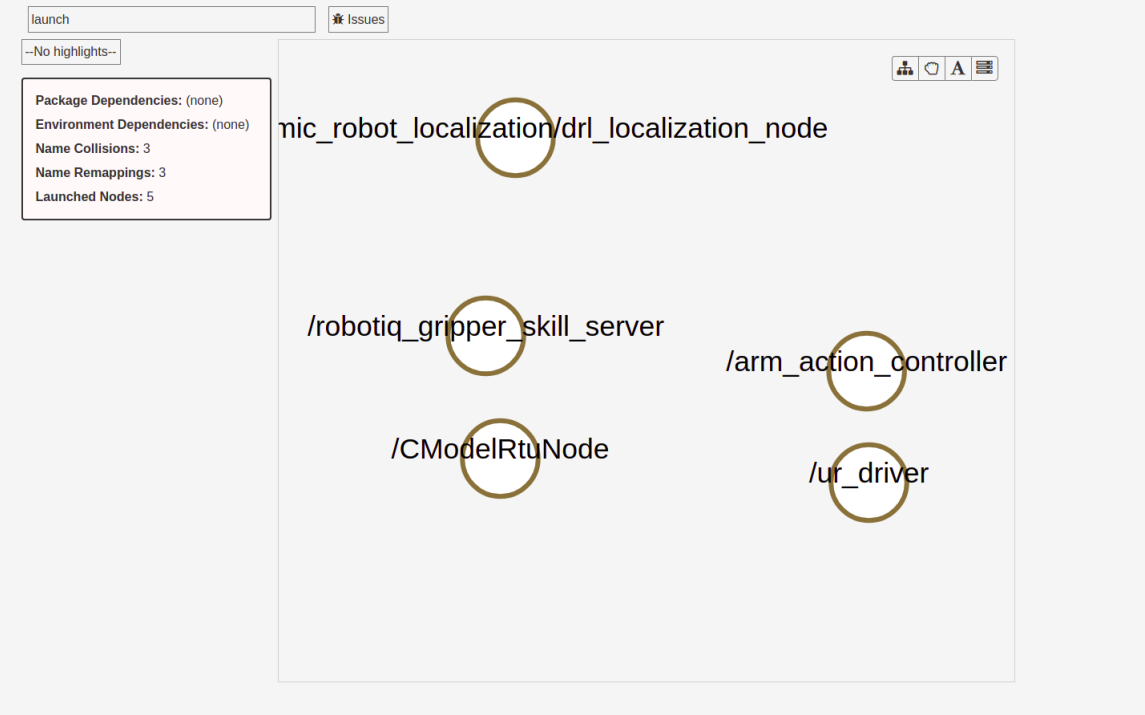

Figure 5.5: Initial iteration architecture model

To detect the topics, additional information was necessary. In order to obtain that information, there were two possible solutions. The first was to compile the work space where the packages were installed using *catkin_make* and to add additional commands, with the full command being: *catkin_make -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DCMAKE_CXX_COMPILER = /usr/bin/clang++-3.8*. The second solution was to extract the information with the following command: *cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -DCMAKE_CXX_COMPILER= /usr/bin/clang++-3.8 src*. Both options force the use of the Clang compiler, and the first also forces the use of *catkin_make* to compile the packages. This is restrictive since it forces the use of predefined options that might not be ideal for the developer. At CRIIS the prefered compilation tool is *catkin build*.

With the information on how to extract the architecture model, the instructions were followed, but both failed to succeed. Therefore, it was not possible to extract the architectural model of the FASTEN project. However, HAROS provides other feature mainly targeted for Python nodes that provides *hints* about which topics these nodes subscribe and publish to. The outcome of these additional hints is illustrated in Figure 5.6. This *hints* could also be provided for the remaining of the nodes. However, that would defeat the purpose of the model extraction tool, which was to extract the model and assert its validity and correctness.

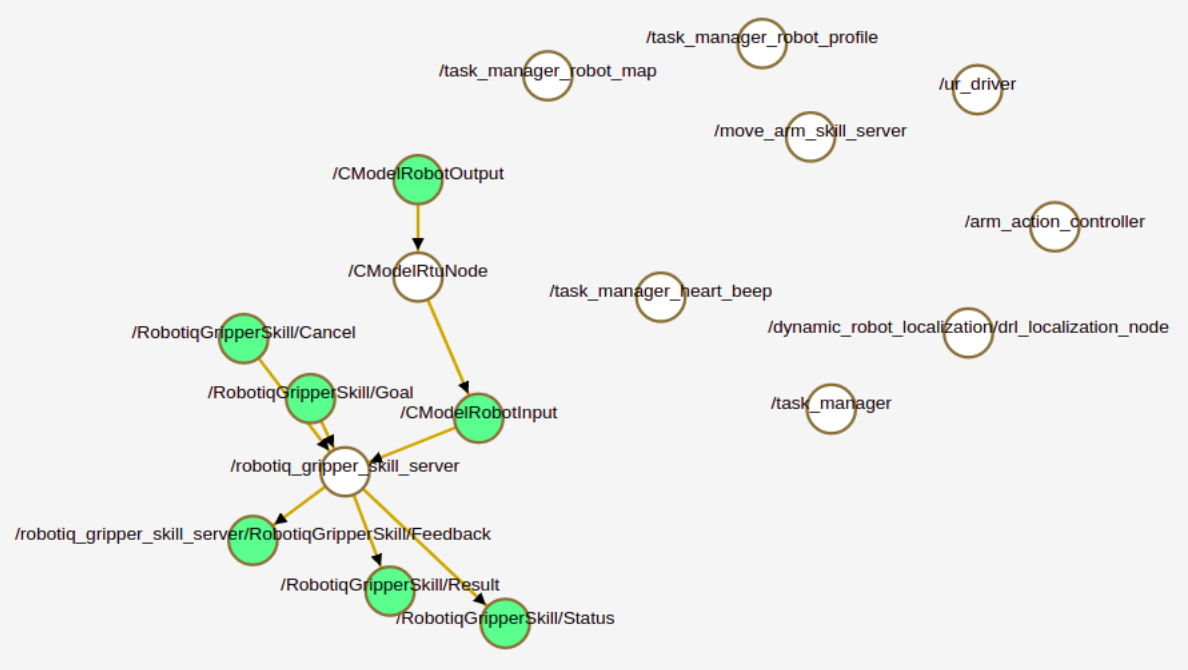

Figure 5.6: Model with hints

Since it was not possible to extract the architectural model it is impossible to conclude anything about its validity. It is however possible to make conclusions about the HAROS framework for this application. The requirements for the use of HAROS to extract the model are restrictive, as it should be compiler independent and also allowed to work with the various tools used to compile ROS workspaces. If the extraction had been successful, the visualization component would provide a good insight into what to expect from the launched nodes. In the future, with alternatives provided for the found problems, it would be appropriate to revisit this model extraction and testify its true potential.

5.8 Best Practices

This section contains advice on the best programming practices to follow. These recommendations are mostly based on the findings of the several iterations of this analysis. The remaining are general good practices that should systematically be followed in every project.

- **Be Consistent** - While developing new software or maintaining it, it is essential to follow the same style throughout the code. This consistency ensures the readability is constant for all the code.
- **Automatic Formatting** - Automated formatting tools are extremely useful. They allow to eliminate human error on formatting and keep the code consistent. ROS C++ guide [40] also suggests the use of automated tools. They provide a style guide on clang-format and instructions on how to use them [41]. The clang-format uses YAML, so it can be configured to the desired formatting style. It also has some predefined styles such as: LLVM, Google, Chromium, Mozilla, WebKit, and Microsoft [42].
- **Document the code** - For maintenance purposes, it is fundamental to have the code well documented. Each function should have a name indicative of it purpose, but some additional and relevant comments, such as indications of a certain algorithm works, are a plus. The documentation should provide information not only about *what it does* but also, and most important, *how it does it*. This insightful information enables the maintainability and allows the transfer of knowledge. Furthermore, the documentation should also be maintained and updated as the code changes.
- **Use static software verifications throughout the development** - Static code analysis provides insightful information about the state of the software and issues that it might contain. These tools should be used since the early development phase, not only to ensure the quality, safety, and dependability of the code but also to prevent wasting time and money on testing overly complex software. For ROS software, the HAROS framework is a valid option. Nonetheless, for general purpose software, there are other solutions, such as some of the ones integrated in integrated development environments (IDEs), such as Visual Studio IDE.
- **Set Tab to Spaces** - This is a very useful tip that will save time and effort. Making this change will guarantee that the code is indented with spaces instead of tab, as recommended. It will also keep the code uniformly indented.
- **Use new-style casting** - The new-style casting is more easily identified and is designed in a way that allows compilers to detect more errors caused by its misuse. Moreover, it is also a safer option.
- **Do no use namespace using-directives** - The of of these directive increases the scope of names that are being looked up. This can cause the compiler to find an identifier that is different from the one expected by the developer, which can be very dangerous.
- **Naming headers** - The header files should be named in a way that not only identifies its purpose, but that also allows its inclusion in a correct order.
- **Do not trust the order of evaluation** - **Never** trust the order of evaluation. The code should be developed to yield the same result whichever order of evaluation the compiler follows. Not following this implies the code is not safe nor dependable.

- **Do not comment out code** - With modern version control tools, it is a bad practise to comment out code. If the code is no longer relevant, it should simply be deleted.
- **Safety-critical** - If the code is being developed to safety-critical applications it is suggested that more strict code standard such as MISRA C++.

5.9 Suggestions For HAROS

For this analysis, HAROS was very useful and suited. However, this tool is still not completely developed and some enhancements could be made to provide additional and useful features. Therefore, some other adjustments could be made to improve and facilitate the usability of this tool.

- **Allow to indicate the used formatting style** - Google and ROS have different styles that are sometimes opposite. This means that some issues will always be present, such as the curly braces. That leads packages to sometimes detect thousands of issues, even if the code is correctly formatted. Indicating the style being used to format would correct this issue, making it easier to find issues relative to the format being used.
- **Model Extraction** - The model extraction, that allows the architecture analysis, is the differentiator and revolutionary feature of this tool. However, the process to achieve this is not as straightforward. Also, it is restrictive compiler-wise, since it needs to Clang compiler.
- **Support for Conditional launch** - The conditional launches are extremely useful since they allow to only launch a sub-system that needs testing with minor changes on the launch files. So, the capability to find if nodes are being launched conditionally and present its architecture is extremely important.
- **Summarized report** - Currently, HAROS does not provide a summarized report for each package. For analysis purpose such as the one performed in this work, it would be practical to have a summarized report. Such a report could include some of the information that HAROS makes available for the project, such as the lines of code, average cyclomatic complexity, and average function length for each package. Additionally, the report could contain the type of issues found for each package and its number, similarly to the tables presented in this analysis.
- **Provide the Severity of the issue** - HAROS could provide the severity of each issue with values, as what is proposed in section 5.3. This could help more inexperienced developers to assess which issues should they tackle first.

Chapter 6

Conclusion and Future Work

Given the large number of issues found in this analysis, it is safe to conclude that, currently, the bulk of effort in developing robotic software is not placed on code quality. If static analysis tools were of standardized use among the robotic software developers, the number of issues found on it would be considerably lower. Software with fewer issues would be more dependable and safer, which are two key attributes for modern Cyber-Physical Systems, collaborative robots, and Industry 4.0. However, there are not many static analysis tools that take the architecture of ROS in consideration nor that can be easily used by ROS developers.

On this work, the HAROS framework was used. This is a tool being developed by HASLab, with the particularity that was designed with ROS architecture in mind. The HAROS framework not only analyses the source code but also the launch files, allowing it to obtain additional information about the architecture. Then, with the collected information, the necessary changes can be made to improve the quality of the software.

In order to use this tool, a case study where it could be applied was needed. Therefore, the FASTEN project was the ideal candidate for this. This case study holds many common features of modern collaborative robots and it is still in the development phase. Being in development implies that there is still room to improve its software quality.

Once the case study was chosen, it was necessary to fully understand it and its capabilities. Thus, an assessment of the case study was performed, detailing not only the software but also the hardware. On the software, its high-level architecture was detailed to understand the flow of information and to grasp how the capabilities were achieved. Then, the implementation level was studied to understand how the high-level architecture was implemented.

With the case study understood, it was then possible to apply HAROS to analyse it. Since this tool provides both source code and architectural analyses, it is only natural to approach these analyses separately. For the source code analysis, an iterative methodology was adopted. This methodology allowed to analyse the code, and then address some issues, re-analysing the code to assess the effectiveness of the fixes and to determinate which issue to tackle next to repeat the cycle. For the architectural analysis, a iterative methodology was also proposed, to allow it to collect information about the architecture and access if it was correct. If not, corrective measures

should be taken and re-accessed in a cyclic way.

The analysis of the source code had 3 iterations, wherein each several issues were tackled. The initial analysis found 66 kinds of issues and a total of 28043 issues. Most of the issues on this initial analysis were formatting issues. For this reason, the formatting was the focus of the first iteration. To tackle this kind of issues, the code was formatted automatically. The first iteration allowed the elimination of 22686 issues, which represent around 80 % of the initial issues. For the second iteration, there were still many candidate issues to be tackled. Among these, some were left untouched since it was dangerous to change them, while others were considered not to be an issue. The tackled issues for this iteration were copyright, line length, casting, constructor, whitespace, and floating point issues. Some of these fixed issues where dangerous, so the changes had a positive impact on the safety of the code. The fixes made for the second iterations allowed a reduction of 2498 issues, which is an approximate 50 % reduction from the previous iteration. Finally, the third and last iteration focused on the cyclomatic complexity, thread-safety and include issues. The cyclomatic complexity is very often difficult to solve, and it is dangerous since functions with high cyclomatic complexity are either very hard or impossible to test. This iteration allowed to solve 256 issues, from which some were dangerous, thus improving the safety and dependability of the software.

Overall the source code analysis allowed to solve 25440 issues, which represents a reduction of 90% from the initial results. Some of the fixed issues were dangerous, thus allowing to improve the safety and dependability of the code. This analysis also allowed to perceive common issues and compile a *Best Practices* guide, which will help the developers at CRIIS to avoid them in the future thus creating safer and more dependable software.

The architectural analysis was not so successful. It was not possible to extract the model and visualize it, since some of the used packages were not compatible with the tools used by HAROS.

In the future, it would be interesting to revisit this work. Especially to try to reuse the HAROS framework to extract the model and finally verify the architecture of the FASTEN case study. Furthermore, it would also be important to continue the source code iterations in order to try to achieve 0 issues and further improve the safety of this project.

Appendix A

Article Submitted to ROBOT 2019

Applying Software Static Analysis to ROS: The Case Study of the FASTEN European Project

Tiago Neto[1,2], Rafael Arrais[1,2], Armando Sousa[1,2], André Santos[2,3], and Germano Veiga[1,2]

[1] Faculty of Engineering of the University of Porto, Portugal,
[2] INESC TEC - INESC Technology and Science, Portugal,
[3] Universidade do Minho, Braga, Portugal,
tiago.f.neto@inesctec.pt

Abstract. Modern industry is shifting towards flexible, advanced robotic systems in order to meet the increasing demand for custom-made products with low manufacturing costs, and to promote a collaborative environment for humans and robots. As a consequence of this industrial revolution, some traditional, mechanical- and hardware-based safety mechanisms are discarded in favor of a safer, more dependable robot software. This work presents a case study of assessing and improving the internal quality of a European research mobile manipulator, operating in a real industrial environment, using modern static analysis tools geared for robotic software. Following an iterative approach, we managed to fix about 90% of the reported issues, resulting in code that is easier to use and maintain.

Keywords: Software Static Analysis, Safety, Mobile Manipulator, ROS

1 Introduction

The shifting of paradigm imposed by the ongoing Fourth Industrial Revolution is introducing a new set of constraints and opportunities for industrial enterprises. These constraints and opportunities are serving as a catalyst for the introduction of flexible, adaptable and collaborative human-robot hybrid systems which can enable even small and medium enterprises to adapt to paradigm changes in market demand, often characterized by increasing customization [2]. These systems are materializing as collaborative robotic solutions in industrial applications and as autonomous mobile robotics in sectors ranging from agriculture to intralogistics, operating in a dynamic and unstructured environments shared with humans.

Such advanced robotic systems, operating in cross-sectorial domains of activity, sensing and interacting with complex and unstructured environments require the integration and support of the technologies, models, and functional components that enable robotic operations. In this context, the safety of humans operating and interacting with potentially dangerous equipment is a core scientific and technological challenge. Thus, and to cope with market demand for

product customization or demanding field applications, contemporary robotics must drastically alter the safety assurance paradigm.

Traditionally, roboticists majorly relied on mechanical-based methodologies, such as physical barriers, to account for safety behaviour. However, as modern systems need to be flexible, adaptive and collaborative to adhere to the ongoing industrial revolution, software-based safety assurance mechanisms are emerging as a complement to traditional safety procedures. In addition, software-based safety assurance can play important social and psychological roles to foster the acceptance of robots in human-populated environments and to promote collaboration between humans and robots. This change affects the robotics ecosystem and calls for techniques to promote best software engineering practice guidelines for the development of safety-critical software, suitable for the robotics development environment.

In a clear contrast with the current necessities, particularly in the cutting edge of innovation efforts, this meticulous attention to software engineering guidelines and safety assurance of software-based components is often overlooked [5] due to the experimental nature of developments, the complexity of the systems, and the difficulties associated with validating the software-based safety mechanisms in physical hardware.

Over the last decade, frameworks such as the Robot Operating System (ROS) [3] have emerged as de facto standards for robotic software development, with an increasing presence in the industrial environment. ROS provides roboticists with abstractions and a vast amount of libraries that widely simplifies and speeds up the development of advanced robotic systems. However, these benefits come with a price, in particular, the intrinsic difficulty to fully assess and validate ROS-based software and external libraries in what regards their compliance with safety protocols or even guidelines for software engineering best practices.

The project_A (SAFER) project, in which this work is integrated, brings together the expertise of computer scientists, with a background on software system design and analysis, and experienced robot engineers, to overcome the aforementioned shortcomings of ROS-based software development. One of the project's main output is the High Assurance ROS (HAROS) tool, a static analyzer of ROS-based software, that can extract valuable information from the source code without the need for executing it (or even compiling it, in many cases). The application of this tool during the development process promotes compliance with software engineering best practices and can be a valuable tool to allow developers to assess the safety compliance of their software. Furthermore, by promoting the creation of better-structured source code, its readability, maintainability, and scalability are deeply improved, potentially resulting not only in increased safety compliance but also in long-term financial gains, as the produced source code is easier to work with.

In this paper, the application of the HAROS tool to a complete stack of ROS-based software powering a mobile manipulator operating in an industrial environment is explored, with the objective of assessing and iteratively improve the code quality. The remainder of the paper is organized as follows: Section 2

presents a conceptual overview of some of the discussed domains, as well as a brief state of the art of the subject; Section 3 presents a detailed description of the industrial utilization of the developed mobile manipulator, its hardware composition, and its software architecture; Section 4 highlights the principal scientific contribution of this research work, by presenting the methodology and results obtained from the application of the HAROS tool to guide ROS-based software development; and, finally, Section 5 draws some conclusions and points out some future work roadmap.

2 Related Work

A deciding factor in the adoption of robotic systems in real-world scenarios is related to the trust levels that humans have in their utilization. In order to fully promote the mass adoption of robotic systems in manufacturing, complying with the ongoing industrial revolution, users need to be fully confident in their operation. In what concerns these systems in a broader sense, trust can be defined as a combination of reliability, safety, security, privacy, and usability [7].

Static analysis techniques are one of many software engineering techniques that can elevate the quality of code, and thus also increase trustability in the developed system. This conceptually simple and time-efficient technique allows, since an early phase of development, the extraction of precious information from a program without running or even compiling it. Among the collected information, compliance of the code with given specifications, internal quality metrics and conformity with coding standards are amongst the most valuable metrics [6]. Static analysis tools evolved to be able to deal with industrial applications, containing millions of lines of code. In [1], the authors provide a comparative analysis of three of the most powerful and popular static analysis tools for industrial purposes, namely *PolySpace*, *Coverity* and *Klocwork*.

In the domain of robotics, ROS, an open-source tool-based framework that provides developers with a large set of libraries and abstractions to ease the difficult task of developing robotic software [3]. Since its introduction, ROS is increasingly being introduced in industrial applications. However, ROS does not impose strict development rules to ensure its safety. Due to the great diversity of ROS applications, there is no solution to completely analyse and verify ROS programs in a formal way and certify their safety to guarantee correct behaviour of robots.

As an alternative to the lack of intrinsic safety compliance mechanisms in ROS and the underlying difficulty to validate such compliance, software static analysis can yield valuable information about the behaviour of each of its subsystems and the interactions between them, thus allowing developers to preemptively verify if the source code is according to the requirements and, consequently and implicitly, improving its safety compliance capabilities [5].

Despite the potential of this technique, applying it to ROS is not so straightforward. As previously mentioned, ROS is very customizable, has a large number of primitives and can be written in several programming languages. This diver-

sity leads to an extremely complex and unfeasible ad hoc solution for an arbitrary ROS system. Nevertheless, for a more restricted set of ROS subsystems, and a bounded set of constraints, it could be achievable [5].

An example of a static analyser for ROS-based code is HAROS. HAROS is being developed having two fundamental ideas in mind: one is the integration with ROS specific settings, and the other is that it should not be restrictive, thus allowing the use of a wide range of static analysis techniques. The latter notion leads to HAROS allowing the integration and use of third-party analysis tools, as plug-ins [6]. This tool allows the fetching of ROS source code, its analysis and the compilation of a report in an automatic way. Therefore, it can be easily used, even by developers without extensive knowledge of ROS or static analysis techniques.

With HAROS, the user first chooses which packages should be analysed, and according to the required analysis, HAROS will dynamically load the adequate plug-ins. The properties that are analysed can be of two categories: rules or metrics. Rules report violations as individual issues, while metrics return a quantitative value, which can, in turn, result in a set of issues [6]. Once the configuration and analysis steps are concluded, the results are portrayed to the user in both a graphical form and by a list of issues, which can be filtered by their type. In its graphical form, the results are portrayed to the user in both a graphical form, and by a list of issues, which can be filtered by their type. In its graphical form, the results visually display the analyzed metrics, and, most importantly, the system-wide and intra-node architecture and properties.

On [5], the authors focused on interpreting the outputs of applying a static analysis provided by ROS on a set of popular and publicly available ROS packages. Collecting this kind of information is important to elucidate about less used or even misused features and is also useful for developers of static analysis tools to determinate which features are more relevant to be supported [5]. HAROS was also used by the authors of [4], to extract and analyze the architecture of a field robotic system for the agriculture domain at static time. This verification provides valuable information during the development phase, which was used to ensure that safety design rules were well implemented in the architecture of the studied robot, validating and improving the safety of the system [4].

In this work, HAROS is applied on an industrial robotic system not only with the purpose of validating this tool, but also, and more critically, to attempt to verify and improve the safety of the system and, indirectly, the maintainability of the source code, as it will be demonstrated in Section 4.

3 Case Study Description

The case study for the work was the H2020 Flexible and Autonomous Manufacturing Systems for Custom-Designed Products (FASTEN) project. This project aims to develop, demonstrate, validate, and disseminate a modular and integrated framework able to efficiently produce custom-designed products. In order to achieve that it integrates digital service/products manufacturing processes,

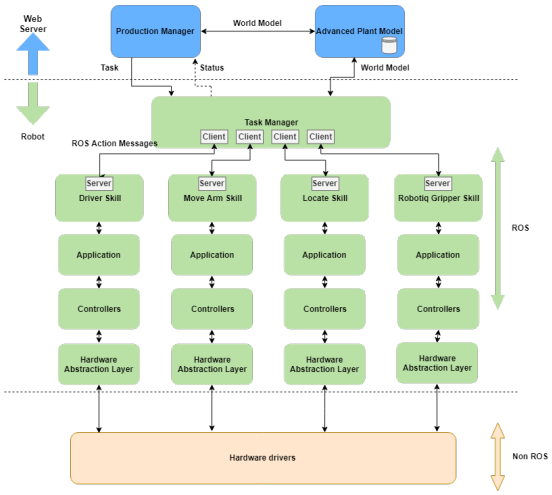

Fig. 1. High-level software architecture of the FASTEN robot system.

decentralized decision-making and data interchange tools. Thus, to achieve a fully connected and responsive manufacturing system, several technologies are being developed, as is the case of sophisticated self-learning, self-optimizing, flexible and collaborative advanced robotic systems. As proof of concept, a mobile manipulator, capable of assembling and transporting kits of aerospace parts is being developed. Currently, at the scenario, Embraer Portugal S.A. (Embraer Portugal S.A.), the industrial end-user of the project, stores the parts used for wing assembly in a Automated Warehouse System (AWS). The kitting operation, composed by the retrieval of components from the AWS is a repetitive, non-ergonomic and non-added-value task which can be automatized to improve performance and working conditions. Furthermore, by relying on an automatic solution to assemble kits, Embraer Portugal S.A. can further enhance the traceability of its intralogistics process.

For this, an automated solution is being developed (Fig. 2). It is composed by an Automated guided vehicle (AGV) with an omnidirectional traction configuration, fitted with a collaborative robotic manipulator. So, this mobile manipulator is capable of traversing the logistic warehouse in any direction and cooperate with human operators in the assembly of kits, increasing the automation level and freeing human operators for more added-value tasks.

The software architecture of this system is being developed with three main objectives in mind, that lead to three structural ideas. The first objective is to reduce the cost of adapting robot applications by promoting code re-usability. To achieve this, a skill-based robot programming approach was used. The second objective is to promote intuitive and flexible robot programming, achieved by task-level orchestration. The third objective is to support generic interoperability with manufacturing management systems and industrial equipment. As depicted

Fig. 2. FASTEN Mobile manipulator developed for application in an Embraer Portugal S.A. industrial plant.

in Fig. 1, this robotic system has a distributed architecture. In the server side implementation, there are two components, the Production Manager (PM) and the Advanced Plant Model (APM) [8], while on the robot side of the architecture, there are the skills and the Task Manager (TM). The APM keeps a near real-time model of the production environment. The PM is responsible to manage the production resources, control the execution of the production schedules and it is also responsible for monitoring the ongoing performance of the different production tasks.

On the robot, one of the most important components is the Task Manager (TM). The TM has two primary functions: it (i) provides integration between the robot and other modules on the system, like the APM or the PM, and (ii) is responsible for the orchestration of tasks, using the skills of the robot. On the TM there is a ROS Action Client for each skill and on each skill, there is a ROS Action server. This is due to the fact that skills are implemented using ROS Actions. The TM uses skills by defining a goal and sending it to the respective Action server. When the execution is completed it receives, from the skill Action server, the result and additional information about the outcome of the performed action.

For the H2020 FASTEN demonstrator, the robotic system has been instantiated with four different skills: (i) Move Arm Skill, (ii) Gripper Skill, (iii) Locate Skill, and (iv) Drive Skill. The Move Arm Skill is responsible for the movement of the robotic manipulator. The Gripper Skill is responsible for the actuation of the gripper. The Locate Skill is responsible for the recognition and localization of the parts that need to be handled. Finally, the Drive Skill is responsible for the movement of the robotic platform and ensuring that the movement is collision free. Each of these skills is organized in three different parts, which are the Application Layer, the Controllers Layer, and, finally, the Hardware Abstraction Layer. These three layers allow a goal received from the TM to be transmitted to the hardware drivers and then executed.

4 Software Quality Analysis

A software quality analysis was conducted on the ROS-based mobile manipulator software presented in the previous section. This software stack comprised the set of functional components, in the form of ROS source code and launch files, responsible for powering the FASTEN use case demonstrator. In total, 22 packages were analysed, from which 14 contained C++ source code, while the remaining contained Python source code or only ROS launch files. The C++ source code amounted to approximately 200,000 lines of code.

To conduct this analysis, the HAROS tool was used. After an initial overview analysis of the complete system, its source code issues were listed and grouped by category for each ROS package. The remainder of the analysis was iterative. This means that the source code issues and model inconsistencies discovered were addressed in several iterations. After each individual iteration, the obtained results were re-evaluated with the HAROS tool and the strategy for the next iteration was drawn. This iterative approach was elected due to the intrinsic difficulty to address all software issues in a single run, allowing developers to assess, in each iteration, if the proposed changes do not impose constraints on the integrity of the system. In addition, addressing all software problems in a single passage would most likely originate novel issues that would be hard to trace the origin of. Moreover, an iterative methodology was employed in order to promote the continuous integration paradigm.

The conducted analysis can be divided into two distinct phases. The Architecture Analysis, presented in subsection 4.1, allows developers to have the full-scale system-wide and intra-node overview of the system and assess if the developed architecture is according to the specifications. The Static Code Analysis, presented in subsection 4.2 refers to the reasoning on the source code of each software application that composes the system. This analysis allows developers to catch safety-critical issues, and assess if the code complies with normative standards and guidelines, thus empowering not only the safety of the whole robotic system but also the underlying code maintainability and scalability.

4.1 Architectural Analysis

The architecture analysis is the differentiator feature that separates the HAROS tool from the remaining static analysis tools. For this feature, it is necessary to inform HAROS which ROS launch files should be analysed. Then, with that information, HAROS extracts the ROS nodes that are being launched by that file and the arguments that are being passed during the launch. However, in its current version, HAROS is not capable of finding a node that is being launched conditionally.

As the FASTEN mobile manipulator development is adopting a methodology where the ROS launch file of each sub-system is conditional it was necessary to provide hints via a YAML configuration file required by HAROS. These hints provide HAROS with additional information about which ROS topics are subscribed or published by each ROS node that composes the system.

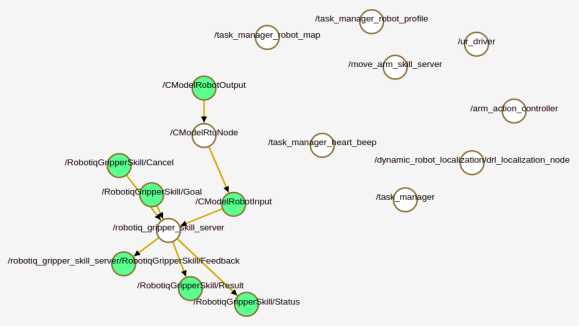

Fig. 3. Architectural analysis of the robotic system as displayed by the HAROS web-based visualization tool.

The visualization of the output of this architectural analysis in the HAROS user interface is depicted in Fig. 3. This visualization component provides a good insight into what is to be expected from the application ROS nodes. Nevertheless, since this extraction could not be automated and had to be provided by hints, the model extraction tool validity and correctness is questionable for the purposes of this case study.

4.2 Static Code Analysis

Initial Analysis This initial analysis contains the raw data collected using the HAROS tool. The issues were divided into 3 categories: Formatting, Code Standards, and Metrics. The first category, Formatting, encloses issues related to indention, whitespaces and the placement of braces. The second, the Code Standards, encloses issues related to the compliance with code standards, i.e. adhering to a specific style of programming or restricting oneself to a subset of the programming language. Finally, the Metrics, encloses issues related to internal quality code metrics, such as cyclomatic complexity or the maintainability index.

Since it was impossible and impractical to solve every issue with one run, the intervention process, guided by the issues reported by HAROS, was divided into several iterative steps. Furthermore, it was necessary to determine which issues would be tackled first. In order to elect the first issues to be tackled, a model, described by Equation 1 is proposed.

$$Score = K_1 \cdot Num + K_2 \cdot S + K_3 \cdot E; \tag{1}$$

This model attributes a score to each issue within a ROS package. The score is a weighted sum of the number of issues, Num, where S represents the severity of the issue and E represents and the effort to solve it. For this analysis, S and E were classified using a rank ranging from 1 (not severe, easiest to solve) to 3 (severe, hardest to solve). K_1 and K_3 were given the coefficient 1 while to K_2, which represented the severity, was given the coefficient 10. The biggest

coefficient weight was given to the severity so it could have a more pronounced impact on the total score of an issue.

The initial analysis of the source code resulted in the report of a total of 28,040 issues, as can be seen in detail on Table 1.

First Iteration For this first iteration, it was assumed that the code did not follow any code standard format since the code was developed by various development teams, and it also simplified the code format standard uniformization to be conducted. Analysing the results of the initial analysis, it is pretty clear that most of the issues are of the formatting type, as can be seen on Table 1, which means that they should be the first ones to be tackled. Since the code is vast it would be impractical and extremely time-consuming to correct all the formatting issues by hand. So, in order to tackle this kind of issues an automatic approach was taken. The chosen tool was the *Clang-Format* along with Visual Studio Code.

The *Clang-Format* was used to format the code accordingly to the Google C++ style guide. The decision to chose Google C++ style instead of ROS C++ Style was based on the fact that the portability of the majority of the source code to this style guide would be more straightforward. After the use of this tool, some additional adjustments had to be done by hand. This was necessary to ensure that the code still compiled. The adjustment done by hand were mostly related with include orders since the automatic tool rearranged the header files in such a way that compilation was not possible.

This first iteration allowed to eliminate 14 types of issues, from 67 in the initial analysis to 53 at the end of the first iteration. This was mostly because of the reduction of the Formatting issues from 24 to 10. On the total number of issues, it was registered a decrease of 22,686 issues. Even though the Formatting and Code Standard issues decreased, the Metric issues increased. The cause of this was the changes made to respect the line length that triggered an increase in the use of vertical lines. This increase originated a spike in the number of functions to have more than 40 lines of code, which, in its turn, triggered more Metric issues.

Second Iteration For the second iteration, one of the issues with a higher score was the *line length*. Since the automatic formatting did not solve this, the source code was manually analysed to understand the root of this issue. There were two explanations: (i) functions with long names could not be solved, and (ii) comments with section markers could not be automatically processed. Regardless, this could be solved by reducing the number of repeated characters without removing the code separation.

Another issue with a high count of occurrences was the *Non-const Reference Parameters*. This issue was caused by variables being passed by reference, but not using the keyword *const* as recommended by the Google C++ style guide. This issue has two possible solutions. The first is to use the keyword *const* if the variable does not need to be changed inside the function and the other, which

requires more effort, is to pass by a pointer and to change the code according to this demand. However, since the second solution was the one that needed to be applied more often, it was opted to leave the code as is, to avoid cross-package errors that could be hard to track. Also, this type of issue did not represent a safety threat.

The issues of the type *Integer types* were also among the issues with a higher count. These issues were mostly triggered by the use of the type *size_t*, but also by the use of the type *short* or *long*. The usage of the type *size_t* is allowed by Google C++ style guide when it is appropriate, which was the case for the totality of occurrences, and for that reason, it was not changed. When types such as *short* were being used, they were replaced by size specific types, such as *int16_t*.

In this iteration, issues with *whitespaces*, *copyright*, and *contructors* were tackled. The *copyright* issues were solved by adding a copyright statement to each file, while the *constructors* issues were addressed by making constructors with single argument explicit. Furthermore, issues related to *casting* were also solved during this iteration. However, at the end of the iteration, HAROS still identified 2 casting issues. Yet, while manually inspecting the code, it was found that these were not casting issues, but were, in fact, false positives.

Finally, in this iteration, the issues with the *floating point* were solved. These issues were caused by float point expressions that were expecting exact equality, which is not compliant with the MISRA C++ guidelines, deeming it unsafe. The solution for these issues was to rewrite the expressions in a way that did not test equality directly and that was compliant with the guidelines.

Overall, 2,498 issues were solved in this iteration, which reduced the total of issues to solve to 2,859 at the end of this iteration. The formatting issues decreased from 10 to 5 and the code standard issues from 34 to 32. However, the average severity and average effort to solve increased from 1.85 to 1.95 and from 1.68 to 1.83, respectively. This is justified by the fixing of more issues with lower severity and lower effort to solve. Nevertheless, this was also a successful iteration, since it led to a reduction of around 50% of issues reported in the previous iteration.

Third Iteration This third and final iteration focused on solving issues related to cyclomatic complexity, functions that were not thread safe and also analysed other issues to understand their causes.

Among the metrics, the cyclomatic complexity is the easiest to change and improve. Despite that, it does not mean that it is a simple issue to fix. Some functions with high cyclomatic complexity are impossible to do in a less complex way, as their purpose is to verify a set of conditions that can not be easily changed. Others are simply just too complex, and it is therefore very risky to change them without incurring in drastic changes to the behaviour of the software, as this code belongs to robotic software that is responsible for the implementation of very specialized and complex features, such as computer vision algorithms. Areas like this require some specialized expertise to alter those algorithms, which compli-

Table 1. Static code analysis results of the initial analysis and subsequent iterations.

		Types of Issues	Issues	Average Severity	Average Effort to Solve	Total Score
Initial	Formatting	24	24511	1.61	1.47	34414
	Code Standard	34	3175			
	Metric	8	356			
	Total	66	28043			
First	Formatting	10	2327	1.85	1.68	10545
	Code Standard	34	2288			
	Metric	8	478			
	Total	52	5357			
Second	Formatting	5	253	1.95	1.83	7267
	Code Standard	32	2126			
	Metric	8	480			
	Total	45	2859			
Third	Formatting	5	253	1.93	1.90	6485
	Code Standard	30	1883			
	Metric	8	467			
	Total	43	2603			

cate the task of changing these algorithms. However, for some of these functions, it is possible to understand their purpose without deep knowledge of the area. For some of those, it is possible to achieve the same result using less complex ways. Thus, during this iteration, it was possible to reduce the cyclomatic complexity of functions with a cyclomatic complexity score as high as 17. Above that value, it was opted not to change them due to the high probability to introduce errors. For these more complex functions, it is recommended intervention from a development team with higher expertise in the domain.

In spite of this last iteration not being able to solve as many issues as the previous ones, most of the issues solved on this iteration were harder to solve. Most of the issues solved on this iteration were also more severe, which reflected on the decrease in the average severity. On this iteration, 256 issues were solved, which led to a decrease in the total number of issues from 2,859 to 2,603 at the end of these iterations (around 9%). In this iteration, 2 Code Standard issues were also eliminated, reducing the total type of issues to 43 and the Code Standard issues to 30.

5 Conclusion

Overall, the source code analysis allowed to solve 25,440 issues, which represents a reduction of 90% of issues from the initial analysis. Some of the fixed issues were deemed to be dangerous and could potentially compromise the run-time functioning of the mobile manipulator. As such, the alterations performed by this work undoubtedly allowed the improvement of the safety and maintainability of the source code, and, correspondingly, the FASTEN mobile manipulator operation in an industrial environment.

With this analysis, it was also clear that the introduced improvements could benefit the development process in the long run. Thus, the methodology described in the paper is being applied during nominal development procedures. As such, the FASTEN mobile manipulator development teams are actively using the proposed methodology and applying the HAROS tool in a continuous integration fashion, as to check for potential issues prior to any source code commit.

In the future, this methodology will be applied to other use cases, as an attempt to replicate the improvements in the domains of code maintainability and safety to other robotic systems.

Acknowledgments

This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-029583. The research leading to these results has also received funding from the European Unions Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 777096.

References

1. P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.
2. H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.
3. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
4. A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ros computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69. IEEE, 2019.
5. A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos. Mining the usage patterns of ros primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3855–3860, Sep. 2017.
6. A. Santos, A. Cunha, N. Macedo, and C. Loureno. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, Oct 2016.
7. L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. Cyber-Physical Systems: A New Frontier. *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 1–9, 2008.
8. C. Toscano, R. Arrais, and G. Veiga. Enhancement of industrial logistic systems with semantic 3d representations for mobile manipulators. In A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira, editors, *ROBOT 2017: Third Iberian Robotics Conference*, pages 617–628, Cham, 2018. Springer International Publishing.

Bibliography

[1] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004. doi:10.1109/TDSC.2004.2.

[2] Ken and Tully. robots.ros.org.

[3] H. Wei, Z. Huang, Q. Yu, M. Liu, Y. Guan, and J. Tan. Rgmp-ros: A real-time ros architecture of hybrid rtos and gpos on multi-core processor. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2482–2487, May 2014. doi:10.1109/ICRA.2014.6907205.

[4] A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ros computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69, Feb 2019. doi:10.1109/IRC.2019.00018.

[5] Johannes Kuehn. ROS code quality, 2013. URL http://wiki.ros.org/code_quality, Last accessed on 2019-02-19.

[6] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & Information Systems Engineering*, 6(4):239–242, Aug 2014. URL: https://doi.org/10.1007/s12599-014-0334-4, doi:10.1007/s12599-014-0334-4.

[7] Morgan Quigley, Ken Conley, Brian P Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[8] A. Santos, A. Cunha, N. Macedo, and C. Lourenço. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, Oct 2016. doi:10.1109/IROS.2016.7759661.

[9] K. G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, Jan 1994. doi:10.1109/5.259423.

[10] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, pages i–, Dec 1998. doi:10.1109/IEEESTD.1998.243394.

[11] Mahmood Alfadel, Armin Kobilica, and Jameleddine Hassine. Evaluation of halstead and cyclomatic complexity metrics in measuring defect density. pages 1–9, 05 2017. doi:10.1109/IEEEGCC.2017.8447959.

[12] T. Honglei, S. Wei, and Z. Yanan. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications*, volume 1, pages 131–136, Dec 2009. doi:10.1109/IFCSTA.2009.39.

[13] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. doi:10.1109/TSE.1976.233837.

[14] Thomas J. McCabe and Arthur H. Watson. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication 500-235*, page 124, 9 1996.

[15] Verifysoft Technology GmbH. Halstead metrics. URL: https://www.verifysoft.com/en_halstead_metrics.html.

[16] Kurt Dean Welker. Software maintainability index revisited. *J. Def. Softw. Eng*, none, 08 2001.

[17] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27:44–49, 09 1994. doi:10.1109/2.303623.

[18] Verifysoft Technology GmbH. Measurement of maintainability index. URL https://www.verifysoft.com/en_maintainability.html, Last accessed on 2019-03-27.

[19] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 05 2012. doi:10.1155/2012/959013.

[20] Emmanouil Tsardoulias and Pericles Mitkas. Robotic frameworks, architectures and middleware comparison. 11 2017.

[21] Amanda Dattalo. ROS introduction, 2018. URL http://wiki.ros.org/ROS/Introduction, Last accessed on 2018-12-17.

[22] Aaron Martinez Romero. ROS concepts, 2014. URL http://wiki.ros.org/ROS/Concepts, Last accessed on 2019-02-09.

[23] Isaac Saito. actionlib, 2018. URL http://wiki.ros.org/actionlib, Last accessed on 2019-02-09.

[24] Bill Tonnies. actionlib detailed description, 2017. URL http://wiki.ros.org/actionlib/DetailedDescription, Last accessed on 2019-02-09.

[25] C. Scheifele, A. Lechler, C. Daniel, and W. Xu. Real-time extension of ros based on a network of modular blocks for highly precise motion generation. In *2016 IEEE 14th International Workshop on Advanced Motion Control (AMC)*, pages 129–134, April 2016. doi:10.1109/AMC.2016.7496339.

[26] L. Cavanini, P. Cicconi, A. Freddi, M. Germani, S. Longhi, A. Monteriu, E. Pallotta, and M. Prist. A preliminary study of a cyber physical system for industry 4.0: Modelling and co-simulation of an agv for smart factories. In *2018 Workshop on Metrology for Industry 4.0 and IoT*, pages 169–174, April 2018. doi:10.1109/METROI4.2018.8428334.

[27] B. Breiling, B. Dieber, and P. Schartner. Secure communication for the robot operating system. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–6, April 2017. doi:10.1109/SYSCON.2017.7934755.

[28] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass, and Peter Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98:192 – 203, 2017. URL: http://www.sciencedirect.com/science/article/pii/S0921889017302762, doi:https://doi.org/10.1016/j.robot.2017.09.017.

[29] Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Rosch:real-time scheduling framework for ros. pages 52–58, 08 2018. doi:10.1109/RTCSA.2018.00015.

[30] R. Halder, J. Proença, N. Macedo, and A. Santos. Formal verification of ros-based robotic applications using timed-automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50, May 2017. doi:10.1109/FormaliSE.2017.9.

[31] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. Javamop: Efficient parametric runtime monitoring framework. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1427–1430, June 2012. doi:10.1109/ICSE.2012.6227231.

[32] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 247–254, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-11164-3_20.

[33] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos. Mining the usage patterns of ros primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3855–3860, Sep. 2017. doi:10.1109/IROS.2017.8206237.

[34] M. Al-Nuaimi, H. Qu, and S. M. Veres. Computational framework for verifiable decisions of self-driving vehicles. In *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pages 638–645, Aug 2018. doi:10.1109/CCTA.2018.8511432.

[35] Johann Ingibergsson, Ulrik Schultz, and Marco Kuhrmann. On the use of safety certification practices in autonomous field robot software development: A systematic mapping study. *Lecture Notes in Computer Science*, 9459:335–352, 12 2015. doi:10.1007/978-3-319-26844-6.

[36] *MISRA-C : 2008: guidelines for the use of the C++ language in critical systems*. HORIBA MIRA, 2016.

[37] César Toscano, Rafael Arrais, and Germano Veiga. Enhancement of industrial logistic systems with semantic 3d representations for mobile manipulators. In Anibal Ollero, Alberto Sanfeliu, Luis Montano, Nuno Lau, and Carlos Cardeira, editors, *ROBOT 2017: Third Iberian Robotics Conference*, pages 617–628, Cham, 2018. Springer International Publishing.

[38] Google. Google c++ style guide. URL https://google.github.io/styleguide/cppguide.html, Last accessed on 2019-05-11.

[39] Bjarne Stroustrup. Bjarne stroustrup's c style and technique faq. URL http://www.stroustrup.com/bs_faq2.html, Last accessed on 2019-05-27.

[40] Paul Bouchier. Wiki, Mar 2018. URL http://wiki.ros.org/CppStyleGuide, Last accessed on 2019-04-27.

[41] Dave Coleman. Roscpp code format. URL https://github.com/davetcoleman/roscpp_code_format, Last accessed on 2019-05-11.

[42] The Clang Team. Clang 9 documentation. URL https://clang.llvm.org/docs/ClangFormatStyleOptions.html, Last accessed on 2019-05-11.