Universidade do Minho
Escola de Engenharia

André Filipe Faria dos Santos

Applying Coding Standards to the Robot Operating System

Agosto de 2015

Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Faria dos Santos

Applying Coding Standards to the Robot Operating System

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Manuel Alcino Pereira da Cunha
Doutor Nuno Filipe Moreira Macedo

Agosto de 2015

ACKNOWLEDGEMENTS

I express my gratitude to my supervisors, Professor Alcino Cunha and Nuno Macedo. They maintained a constant presence and guidance throughout the project, by reviewing my work, helping me stay on the right track, and pointing out possible routes and resources of relevance. Without their support I would have faced many more obstacles and dead ends along the way.

I also thank my colleagues and members of HASLab who contributed to an inspirational work environment, reviewed parts of my work, or have otherwise been helpful.

Last but not least, I thank my parents and Sofia for their personal support. They always listened to me, even though at times they did not fully understand what I talked about.

ABSTRACT

Software static analysis is a key component of formal software verification. It consists on inspecting code, using automated tools, to determine a set of relevant properties without executing the program. An example of such properties is the compliance with certain *coding standards* - sets of rules, subsets of the programming languages, defined to achieve high-quality software, minimizing risks and maintenance costs. Some coding standards, such as MISRA C++ or HIC++, are highly adopted nowadays, in safety-critical systems with high reliability requirements.

Recent developments in robotics increased human-robot interaction, and show a tendency to introduce robots in safety-critical applications, such as transportation and health. As a consequence, it is imperative to guarantee the reliability and quality of the software used to control these robots. This research project shall evaluate the suitability of existing coding standards in the context of the Robot Operating System (ROS), and then develop a generic platform to verify compliance with standards and assure high quality robotics software. Kobuki, a ROS mobile robot, is used as a case study.

Keywords: static analysis, coding standards, Robot Operating System, software engineering

RESUMO

A análise estática de software é parte integral da verificação formal de software, e consiste no uso de ferramentas que inspecionam código e determinam um conjunto de propriedades de interesse, sem nunca o executar. Um exemplo dessas propriedades, é o cumprimento de *coding standards* - conjuntos de convenções definidos com o objetivo de produzir software de alta qualidade, minimizando custos e riscos. Alguns *coding standards*, como MISRA C++ ou HIC++, são bastante adotados em sistemas críticos com altos requisitos de fiabilidade.

Os desenvolvimentos recentes na robótica não só aumentam a interação entre humanos e robôs, como aplicam cada vez mais os robôs em áreas críticas, como meios de transporte e saúde. Desta forma, torna-se imperativo garantir a qualidade e fiabilidade do software usado para os controlar. Neste projeto de investigação pretende-se avaliar a adequação de *coding standards* existentes no contexto do Robot Operating System (ROS), e desenvolver uma plataforma genérica para verificação de conformidade com *standards* e garantia de qualidade de software de robótica, tendo como estudo de caso o Kobuki, um robô móvel implementado sobre ROS.

Palavras-chave: análise estática, convenções de código, Robot Operating System, engenharia de software

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

The Third Industrial Revolution represents the transition from analog, mechanical and electronic technology to the digital technology we know and use today. It began somewhere in the late 1950s, marking the beginning of what is known as the Information Age. Along with it, software was born and quickly expanded to take control and make use of the emerging technologies, such as digital computers, cell phones, the Internet and, recently, robots. As these technologies mature and adapt to new contexts, software assumes more and more risks and responsibilities. Such risks range from monumental monetary losses to threats to human lives, and cannot possibly be ignored. Thus arises the need, and motivation, for certified development methodologies and safety guarantees in the software industry (Jackson, 2006).

Similarly to software, robots have been a target of much development and expansion in their applicability. Also similarly to software, robots have to assume increased risks and responsibilities, as they interact with humans, or replace them altogether in various fields (Rifkin and Kruger, 1996), such as military, industrial, transportation or medical devices. As robots became more autonomous and the tasks assigned to them became more complex, hardware-based robots were replaced with robots controlled by software, and the software became consequently more complex, as the tasks require planning, navigation, or speech recognition, for instance.

With the uprise of software in robotics, a variety of re-implementations of algorithms and controllers emerged, wasting valuable research and development time. This called for conventions, for common, tested and reusable *middleware*, a software layer above the operating system that provides a common interface for heterogeneous lower-level components. But, as Smart (2007) sets forth, specifying middleware for robots is quite more challenging than specifying middleware in typical software engineering environments, mainly due to the heterogeneity of the hardware and how robots are susceptible to its failures. However, this difficulty did not hold researchers back, and a variety of middleware systems were developed, with some of them being open source systems (Mohamed et al., 2008; Namoshe et al., 2008; Elkady and Sobh, 2012; Iñigo Blasco et al., 2012). Examples of successful open middleware systems include Player (Vaughan et al., 2003), and the Robot Operating System (Quigley et al., 2009). This project focuses on the latter.

The Robot Operating System[1] (ROS) is a framework that offers functionalities similar to an operating system, oriented to robots, in order to promote the collaborative development of software for robots. This open source system is developed in common general-purpose programming languages, mainly C++ and Python, and it is free for investigation and commercial use, with its audience increasing as of late. This growing community currently stands on tens of thousands of users worldwide. Many diverse projects are already based on ROS, including humanoid robots, autonomous vehicles, surgical robots (Hannaford et al., 2013) and industrial robots (ROS-Industrial[2]). These types of robots are made to interact with humans, or to perform tasks a human would normally be assigned to do, hence assuring the quality, reliability, safety and correctness of the software cannot be dismissed, in order to consequently assure the safety of humans. This situation calls for advanced and formal techniques of quality assurance.

Software static analysis consists on extracting valuable information about a program without the need to execute it, usually relying on automated tools. Analysing a program without executing it is a key component in software formal methods and quality assurance, since it avoids the manifestation of dangerous errors during runtime or testing, while being able to detect them. This is particularly true when testing is not feasible or easy, as is the case of the surgical robots mentioned above. Software errors in industrial robots, for instance, may trigger sudden movement of massive robots that can injure unaware human workers. These and other factors are part of what makes software static analysis a requirement when developing safety-critical products.

One of the many practical applications of static analysis is to determine compliance of a piece of software with certain *coding conventions*. Coding conventions, or coding standards, are rules or guidelines about how a programming language should be *spoken* or, rather, written. They limit the freedom and flexibility native to the programming language, in order to avoid common mistakes and achieve a safer subset of the language, resulting in increased system integrity (Goforth, 2013). Besides the impact they have in improving software quality, they are important for the effect they have on people. In professional environments, coding standards have proven to improve the performance of the programmers, and their ability to understand and solve problems efficiently (Soloway and Ehrlich, 1984).

Good coding standards strive to justify their rules, to base them on known best practices and to produce high-quality software. Examples of good coding standards include MISRA C (MISRA, 2004), for the C programming language, and HIC++ (Programming Research Ltd., 2013) and JSF-AV C++ (Lockheed Martin Corporation, 2005), for the C++ programming language. These standards focus on guiding the production of portable, maintainable, testable, safe and reliable software, targeting safety-critical systems with high-reliability requirements. MISRA C, for instance, is widely adopted in the automotive industry, among others. JPL C (Jet Propulsion Laboratory, 2009), a coding standard based on MISRA C, played a key role in assuring the reliability of Curiosity, the Mars rover launched by NASA, as shown by Holzmann (2014). Knowing that C++ has made its way into robotics systems,

1 http://www.ros.org/
2 http://rosindustrial.org/

and ROS, in particular, it stands to reason that these guidelines may also apply, or be adapted, to high-quality robotics software. The ongoing development of ROS, its increasing adoption and the fact that it is free software make ROS a good fit as a case study of the applicability of coding standards in the production of high-quality robotics software.

Verifying compliance with coding standards, as stated before, is part of the static analysis process, which relies on automated tools to be feasible. While there are plenty of capable static analysis tools that verify compliance with coding standards, none of them is equipped to consider the specifics of ROS (for instance, its package architecture). This lack of tools presents an opportunity to contribute to the ROS community, with an unified ROS-specific static analysis platform. In particular, one capable of verifying compliance with a set of standards, in order to promote better development practices, and higher quality robotics software.

This research project focuses on this very topic, the use of coding conventions in ROS applications. Its main aim is to study the applicability of existing coding conventions in high-quality robotics software, and then explore how these conventions could be adapted or improved to fit this specific context. Complementing these studies, the project also encompasses the development of a static analysis tool for ROS systems. With this in mind, the following contributions are expected from this project.

i) The project shall comprise a study of existing relevant C++ coding standards, with emphasis on high-reliability systems, and the tools available to check compliance with the studied standards.

ii) The project shall also include a basic study of ROS systems, in order to understand the entities and concepts involved in developing ROS applications, and thus decide on which logical layer the static analysis should focus on.

iii) A ROS static analysis tool shall be implemented, capable of verifying compliance of a ROS code base with a given set of coding rules.

iv) The developed tool shall also be capable of presenting its analysis results in a user-friendly way, using diagrams, colour schemes, filters, among other features, considering the specifics of ROS systems.

v) A relevant ROS application shall be used as a case study for the developed tool, and any analysis results shall be reported.

Chapter 2 presents the state of the art on this research topic, which provides more detail about ROS, about coding rules, and a study on relevant coding standards in the industry, including static analysis tools, mainly for the C++ programming language. Chapter 3 delves into the new tool developed under this project, and the work behind it. It details the architecture of this tool, and provides information on how users can use the tool, adapt it to their needs, and even extend its capabilities on their own. This chapter also includes a case study, which encompasses the development of extensions for the tool, and the analysis of the source code of an actual ROS robot, using this tool and the mentioned extensions. Finally, Chapter 4 summarises the research and results involved in this project, and revisits its expected contributions, while delving into each of them. It also provides some final remarks and an overview of interesting open issues, as prospects for future work.

2

STATE OF THE ART

Static analysis is a key factor in software quality assurance, since it allows the discovery of potentially hazardous vulnerabilities before the software is released, installed or otherwise deployed and executed. It consists on using automated tools to analyse programs without executing them. This analysis greatly varies in depth and complexity, ranging from type checking, to code metrics or even abstract interpretation. One possible aspect of static analysis is the compliance of the software with certain *coding standards*.

Coding standards, or coding conventions, are, in essence, sets of rules or guidelines designed to govern the process of code production in a software project, based on industry best practices, and, as such, they have been around for almost as long as programming itself. They are often applicable to a specific programming language, library, framework or environment, but they can also be language-independent, focusing mostly on style. In fact, style conventions are such a common concern that we see them rise even in uncommon contexts (for programming purposes), such as LaTeX, in Verna (2011). Here, we focus on coding standards that may be applied in ROS systems, since our goal is to improve the overall quality of robotics software. That is, coding standards related to the C++ language (one of the main programming languages in ROS), or even to ROS itself, deserve special emphasis in the context of this project.

This chapter covers the essentials to understand this project and its results. In particular, it covers coding rules, existing coding standards of relevance (which put together a set of coding rules), automated tools to verify compliance with standards, and some background on ROS and the general architecture of ROS systems.

2.1 CODING RULES

Coding rules are the core elements of coding standards documents. They dictate what a developer can and cannot do. Their scope of action ranges from stylistic and syntactic guidelines, to encouraging or discouraging the use of certain features of a programming language or library, and even to adopted development methodologies and tools. There is no common origin for coding rules. They can be representative of personal preference, based on industry best practices, consequence of imposed requirements, or a result of some reasoning process. However, Corden (2013) says that a coding rule is only a good rule when it is unambiguous, enforceable, peer reviewed, and when it has a clear justification, examples and benefits.

Given the broad scope of applicability of coding rules, it is not uncommon to have them classified into categories, finer-grained scopes of what their guidelines refer to. Below follows a list of the rule categories often found in C++ coding standards, and examples of each rule type.

GENERAL

This rule family deals mainly with language-independent issues. Rules concerning unnecessary constructs, the C++ language, subsets of the language or compliance and deviation against the standard all fit here.

General Rule	**Unnecessary Constructs**
There should not be any unreachable code.	

NAMING CONVENTIONS

This family of rules addresses the names given to various relevant entities in the coding process. These are often subclassified into file naming, type naming, function naming, variable naming, among others.

Naming Rule	**Variable Naming**
All words in a variable identifier will be composed of lowercase letters and separated by an underscore character.	

OTHER SYNTACTIC AND LEXICAL CONVENTIONS

Syntactic and lexical rules cover issues related to the syntax and lexicon of the language. Naming rules, albeit syntactic, have their own category. Character sets, character encoding, character sequences, style and formatting, however, all fit in this category.

Syntactic Rule	**Formatting**
No line of text in a code file should exceed 80 characters in length.	

COMMENTS

Code comments are arguably syntactic issues, but they are sometimes considered a sufficiently relevant entity on their own. These rules concern commented lines of code, code documentation, explanatory comments, and license statements.

Comment Rule	**Code Documentation**
The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code (AV Rule 130, Lockheed Martin Corporation, 2005).	

TYPES AND TYPE CONVERSIONS

These rules address types, type implementations, type casts, implicit type conversions and implementation portability.

Type Rule	**Type Conversion**
The constant `NULL` should not be used as an integer value.	

EXPRESSIONS

Expression rules encompass what is permitted in an expression. Issues related with expression evaluation, expression side-effects, overflow, or operators fall under this category. Some standards do not make it clear whether rules regarding implicit type conversions during expression evaluation should be addressed in this category, or under *Types and Type Conversions*.

Expression Rule	**Operators**
The operands of binary logical operators shall not contain side effects.	

STATEMENTS

The use of specific language statements, such as `continue` or `break`, assertions, control flow structures, or assignments are all governed by this category of rules.

Statement Rule	**Control Flow Structures**
Never use floating point variables as loop counters.	

DECLARATIONS AND DEFINITIONS

These rules manage issues related to the various types of declarations and definitions possible within the language. The boundaries of this rule family are sometimes blurred with various other families, but, in general, every rule concerning the scope of declarations, declarations of variables, types or namespaces, or the placement of declarations within the source files falls under this category.

Declaration Rule	**Scoping**
Declarations should be at the smallest feasible scope.	

FUNCTIONS

There are many rules regarding functions and methods that have nothing to do with their declaration statements, their names, or the style of the code. Those rules impose restrictions on function bodies, parameters, arguments or invocations, and they compose this rule family.

Function Rule	**Function Overloading**
All overloads of a function must be visible from where it is called.	

CLASSES

Just as with functions, there are rules regarding the use and definition of classes that do not fit other rule families. Whether to use structures or classes, the use of inheritance or restrictions over constructors and destructors are issues addressed by this set of rules.

Class Rule	**Inheritance**
Do not use multiple inheritance.	

TEMPLATES

This rule family provides guidelines on the use of templates, a C++ feature to write generic code. These rules include when to use templates over derived classes, template instantiation and template specialization.

Template Rule	**Template Specialization**
A template specialization should be declared before it is used.	

EXCEPTIONS AND ERROR HANDLING

Rules related to the use of exceptions and to error handling belong to this rule family. What to do when handling errors, how and where to report errors are issues addressed here.

Error Rule	**Exceptions**
Use exceptions to report errors, instead of returning error codes.	

PREPROCESSING

C++ is one of the programming languages that allows the programmer to make use of its preprocessor. Use of the preprocessor and its directives is also a target to various rules and guidelines. This rule family governs file inclusion, conditional compilation, macros, among other features available through the preprocessor.

Preprocessor Rule	**File Inclusion**
Include directly the minimum number of headers required for compilation.	

LIBRARIES

This set of rules provides the guidelines to follow regarding the use of libraries in a project, which libraries are allowed and to what extent they are allowed.

Library Rule	**Standard Libraries**
The signal handling facilities of signal.h *shall not be used.*	

CONCURRENCY

Rules regarding the use and control of concurrency are very specific, yet relevant enough to have their own rule family. Threads, data sharing and mutual exclusion are topics found here.

Concurrency Rule	**Mutual Exclusion**
Within the scope of a lock, ensure that no static path results in a lock of the same mutex.	

TESTING

Code tests are not always addressed by coding standards but, still, they represent a sufficiently important topic in software development to warrant a rule category.

Testing Rule	**Unit Testing**
Every publicly accessible function must be covered by unit tests.	

MISCELLANEOUS

The other rule categories capture much of what is typically specified with coding rules. However, there must be a family for uncommon rules that do not fit other categories. This rule family governs non-conforming code, code deprecation, or development tools, for instance.

Miscellaneous Rule	**Deprecation**
To deprecate a class, deprecate its constructor and any static functions.	

2.2 CODING STANDARDS

Coding standards are collections of coding rules, the logical unit directly above coding rules. While coding standards can be somewhat informal, defined by a small team or corporation for internal use, most adopted standards often take the form of an organised document, aiming to provide a formal definition of their rules, the reasoning behind each rule, and whether (and how) compliance with those rules may be verified. The main advantage of adopting coding standards, regardless of the formality and reasoning of a standard, is to have *consistency* across a code base.

In practice, to a programmer, complying with coding standards means restricting oneself to a subset of the programming language's features or syntax rules, in virtue of consistency, robustness and code readability, as said by Ambler (2000), Bloch (2008), among others. This tends to result in increased team productivity and reduced maintenance and production costs. However, as noted by Li and Prasad (2005), developers may show reluctancy in accepting and applying coding rules, especially when working individually, or when still learning.

The rule classification shown previously helps visualize how coding rules can be structured into a formal document, but it does not stop there. In some instances, the rules of a standard are also classified in compliance levels, usually *required rules* and *advisory rules*. That is, the authors of a standard consider that a given piece of software may accomplish different levels of compliance with the standard, depending on its purpose or the requirements it must satisfy, making the standard flexible and adaptable to various contexts. Furthermore, this categorisation puts into perspective the importance and reasoning behind each rule, while also allowing justified deviations from lower priority rules. Table 1 shows examples of how both categorisations can be applied.

Syntactic Rule	**Advisory**
A line of code should not exceed 80 columns.	
Declaration Rule	**Advisory**
Header files should contain logically related declarations only.	
Comment Rule	**Required**
Every source and header file should contain a license and copyright statement at the beginning of the file.	
Error Rule	**Required**
Do not throw an exception from a destructor.	

Table 1.: Examples of coding rules and their categorisation.

With respect to software quality, adopting code standards may show little to no significant improvement in the resulting software's measurable quality, as Capiluppi et al. (2009) state in the context of open source software projects, but it is not always so. When quality needs to be assured, a proper coding standard, a standard that is adequate to the context it is applied in, will discourage the use of features that tend to lead to vulnerabilities, in favour of safer, verifiable features and code readability. Thus, complying with a well-designed standard will often have a positive impact in the formal verification of safety-critical code, while also improving code metrics, such as portability, maintainability or testability. Yet, coding standards are not always about maintainability and readability, and sometimes these desired properties conflict with the requirements at hand, for coding conventions and software requirements, albeit related, are not interchangeable terms. Contexts where software security is required, for instance, call for conventions that favour security testability over code readability (Okubo and Tanaka, 2007).

Coding standards impose restrictions on how to develop a software product, as said before, while software requirements target the whole software product and the production process, describing what should be done, and how it should be done. This means that requirements are a broader concept, and coding standards, when adopted, are part of the software requirements, not the other way around. This explains why requirements rightfully hold much more focus than coding standards in the industry, but coding standards are far from disregarded. They are (at least informally) present in most software projects, but it is in the safety-critical software industry that they play a decisive role. Some standards, such as MISRA C and others discussed in the subsections below, are not only adopted but also required in sectors including aerospace, automotive and medical devices (Programming Research Ltd., 2014).

Even though the use of coding standards in a project is generally advantageous, adopting a coding standard is a decision that should not be taken lightly, since it affects practically the whole software development process. Consider a typical software development process, as shown in Figure 1. Complying with a coding standard is, in itself, a requirement, but it can also affect other requirements, thus affecting requirement specification and verification. Since coding standards provide rules or guidelines to write code, it has an obvious impact in the implementation phase. The verification phase has to include tools, or people, to verify the compliance with the adopted standards. Finally, the maintenance phase also sees minor impact, since any new code to fix vulnerabilities also has to comply with the standards, and any non-compliant code should also be refactored in this phase to be compliant.

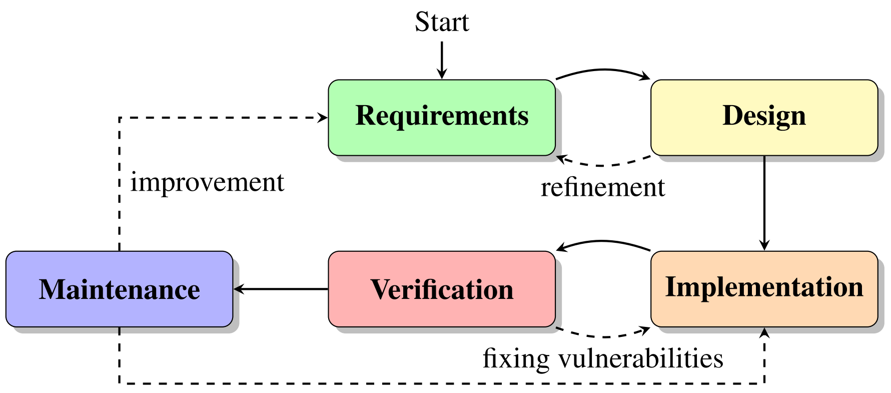


Figure 1.: Example of a software development process.

As we have seen so far, the field of coding conventions is immense, subject to much discussion and opinion, and yet expanding as a sound component of software engineering. Most widely used programming languages, if not all, are sure to have code conventions already in use. Naturally, the scope of this project focuses on a relevant sample. This project focuses on the use of coding conventions in safety-critical and high-quality software in robotics. Specifically, it focuses on software built on top of ROS, and the software that makes up the core components of ROS, using the C++ language. The remaining of this section presents the coding standards that form this relevant sample, selected for analysis and discussion throughout the work of this project.

2.2.1 *ROS C++ Style Guide*

ROS is the focus of this research work, and so any conventions already in use deserve consideration. As it turns out, ROS already provides developers with a C++ style guide, as seen in ROS C++ Style Guide[1]. This is a work in progress for a non-strict guide. It provides guidelines, instead of literal rules, meaning that developers are free to deviate, although it advises the use of documentation to justify the reasoning behind deviations. Above all guidelines, it recommends consistency. New code should be compliant with this guide, while edited code should be consistent with the local style in use.

As hinted by the name, a considerable part of the guidelines in this guide address stylistic concerns. These include naming conventions, license statements, code formatting, code documentation and code deprecation. On the technical side, it addresses some relevant and common concerns among C++ coding standards, such as the use of the C++ preprocessor, class inheritance, exceptions, global identifiers and code testing. It also discusses some less common issues, such as assertions and code portability. Many of these guidelines reference the guidelines provided by Google's C++ Style Guide (see Section 2.2.2), making it an important standard to consider.

Besides this C++ guide, ROS also provides developers with a more general, language-independent guide, seen in ROS Developer's Guide[2]. Its guidelines are not very oriented at writing code, but rather at the enveloping details of code production, common to all of ROS's development languages. These guidelines cover source control, debugging, compilation, licensing, releasing and documentation, among other issues. This way, it is possible to consider a new category of rules, for ROS specific issues, besides the traditional rule categories mentioned in section 2.1. An example of such rules is provided in table 2, regarding the package architecture used in ROS.

ROS Rule	**Packages**
Every package must have a manifest.xml *file, located in the package's top directory.*	

Table 2.: Example of a ROS specific rule.

1 `http://wiki.ros.org/CppStyleGuide`
2 `http://wiki.ros.org/DevelopersGuide`

2.2.2 *Google C++ Style Guide*

Google has its own C++ style guide which is freely available for anyone who wants to adopt it (see Google C++ Style Guide[3]). This style guide has been revised several times, currently standing on revision 4.45, since September 2014. Besides being a free C++ style guide, it also serves as the set of coding conventions in use for C++ open-source projects developed by Google. Such projects are compliant with this guide.

Even though this guide is a basis for a considerable part of the ROS C++ Style Guide described in Section 2.2.1, there are considerable differences between the guides. This guide is more extensive, covering more topics and features of the C++ programming language, and, contrary to ROS, some of its rules are required, not advisory. Its rules are also more structured than those of ROS, in the sense that a majority of them provides definitions, exceptions, examples and justification with upsides and downsides. Not unlike ROS, however, this guide reinforces consistency, especially local consistency, above all other rules, and its rules are only organised by topic or feature. Some rules do not state their compliance levels clearly, leaving whether the rule is a requirement or recommendation to the reader's interpretation. The rules also lack indexing, they are structured mostly in statements or paragraphs that sometimes encompass multiple smaller rules.

Regarding its contents, this guide is very focused on the actual coding. It covers formatting, naming conventions, code comments, the C++ preprocessor, header files, scoping, classes, portability, exception handling, lambda expressions, libraries, among other features. Remarkably, it does not cover testing or concurrency, for instance, but it does cover compliance verification. Google provides a tool, *Cpplint*, to that end.

2.2.3 *High Integrity C++ Coding Standard*

The High Integrity C++ Coding Standard[4] (or HIC++, Programming Research Ltd., 2013) was first published on October 2003, by Programming Research Limited (PRQA), a provider of static analysis tools with more than 25 years of experience in the field. It is a coding standard aimed at the production of high-quality C++ code, with the guiding principles of maintainability, portability, readability and safety. It is a widely adopted standard, having been downloaded more than 24,000 times since its release.

PRQA was involved in the production of the MISRA C standard (MISRA, 1998). From MISRA C's success, and from the work and research in C++ best practices available at the time, they took the opportunity to create one of the first professional C++ coding standards, HIC++ (Corden, 2013). This standard was then the basis for other well established standards in the industry, such as MISRA C++ and JSF AV C++ (Sections 2.2.4 and 2.2.5, respectively). With the advent of ISO C++ 2011 (C++11),

3 http://google-styleguide.googlecode.com/svn/trunk/cppguide.html
4 http://www.programmingresearch.com/high-integrity-cpp/

the language has undergone relevant changes, exposing gaps in the existing coding standards, both due to guidelines that became invalid or irrelevant, and the introduction of new features. In October 2013, celebrating the standard's 10th anniversary, PRQA released version 4.0 of the standard, addressing the language updates and revising previous rules, making the rules in the standard more enforceable and manageable (Basalaj and Corden, 2013; Corden, 2014). Figure 2, taken from the white paper by Basalaj and Corden (2013), shows the rule overlap between HIC++ (version 4.0), MISRA C++ and JSF AV C++.

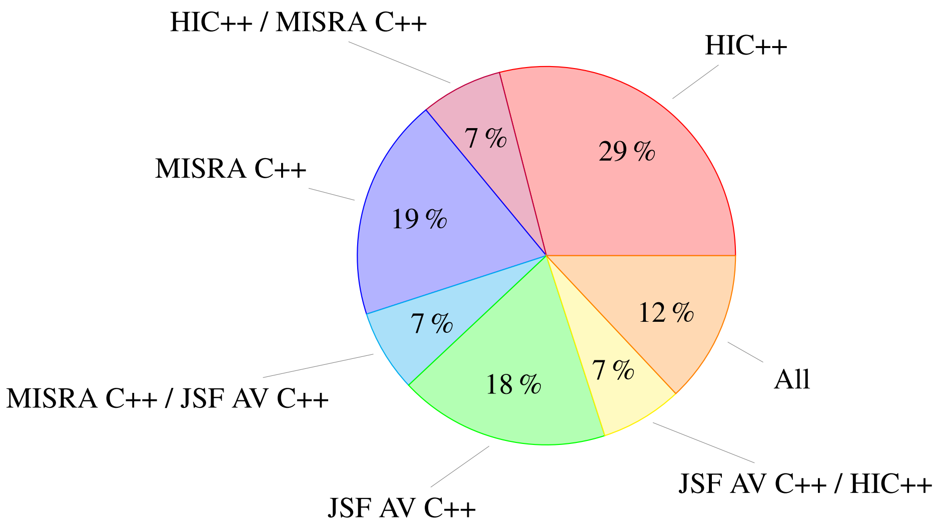


Figure 2.: Rule overlap between HIC++, MISRA C++ and JSF AV C++.

This standard classifies its rules by issue, covering most rule families defined before, but not by compliance. All rules are requirements, and they are written in such a way that makes enforcement by source code analysis possible (Basalaj and Corden, 2013). However, limited deviation is tolerated, when necessary and supported by written justification. Some of its rules constrain expression values and are, thus, theoretically undecidable. To solve this, HIC++ takes a different approach from other common standards, introducing the concept of *demonstrability*. A rule is demonstrable, in a piece of source code, if the issue it addresses cannot occur in practice, even though it could occur in theory. That is, the constrained expression should be appropriately guarded, for instance using assertions, to guarantee compliance. Table 3 and Listing 2.1, taken from the HIC++ Coding Standard as created by PRQA, illustrate this concept.

HIC++ Rule 4.2.2	**Required**
Ensure that data loss does not demonstrably occur in an integral expression.	

Table 3.: Example of rule demonstrability, as defined by HIC++.

```
#include <climits>
```

```
#include <stdexcept>
#include <cstdint>

uint32_t inv_mult (uint32_t a, uint32_t b)
{
    return ((0 == a) || (0 == b)) ? UINT_MAX
            : (1000 / (a * b)); // @@- Non-Compliant: could wraparound -@@
}

void foo ()
{
    inv_mult (0x10000u, 0x10000u);
}

uint32_t safe_inv_mult (uint32_t a, uint32_t b)
{
    if ((b != 0) && (a > (UINT_MAX / b)))
    {
        throw std::range_error ("overflow");
    }
    return ((0 == a) || (0 == b)) ? UINT_MAX
            : (1000 / (a * b)); // @@+ Compliant: wraparound is not possible +@@
}
```

Listing 2.1: Example of rule demonstrability, as defined by HIC++.

2.2.4 *MISRA C++ Coding Standard*

The C++ programming language has conquered significant territory in the world of embedded and safety-critical systems, where previously C and Assembly were dominant languages. This is a consequence of the widespread use of C++, its higher-level features, its flexibility, and the fact that the programs generated by C++ compilers achieve similar performance marks as those generated by C compilers. Besides everything else C++ had already inherited from C, including misunderstood and dangerous features, it inherited the potential for some professional-level coding standards to emerge, when critical systems became one of its targets.

The MISRA consortium has been involved in providing industrial-strength guidelines for critical systems for more than twenty years, with works such as MISRA (1994) and later the MISRA C Coding Standard (MISRA, 1998) for the C programming language. As MISRA C proved to be a well established standard and C++ emerged in critical systems, MISRA released, in 2008, a coding standard to make the best use of this language, the MISRA C++ Coding Standard[5] (MISRA, 2008).

5 http://www.misra-cpp.com/

MISRA C++ is a strict coding standard, effectively forming a safe subset of the C++ programming language. Its rules were based on previous work in C++ coding standards, such as HIC++ and JSF AV C++, and so it is expected that part of them overlap with these standards, as illustrated in Figure 2, even though this figure addresses a later revision of HIC++. As stated by Basalaj (2011), MISRA C++ aims to promote software readiness for production and safety analysis, eliminate or reduce unpredictability, improve clarity and maintainability, avoid common programmer errors and incorporate good practices. Contrary to other coding standards referenced here, MISRA C++ is not a free standard. A copy must be purchased from MISRA, in order to use it.

The rules in this standard focus on features and the use of the language, leaving style conventions to the user. In fact, the standard recommends that, in conjunction with it, the development team should also adopt a style guide, focusing on style issues. The standard organises its rules by topics (e.g.: expressions, declarations, exception handling), and then classifies them by compliance levels. It defines three compliance levels, as follows.

ADVISORY Advisory rules should be followed, whenever possible. However the developer is free to deviate if compliance is not practical. While these rules concern features of importance, deviations are generally acceptable in the development process. Examples of such features include commenting out code, using explicit integer sizes and signedness, or using recursion.

REQUIRED Required rules are requirements, compliance is mandatory. If compliance is not possible, a formal deviation must be raised. These rules compose the majority of the document, and cover the most diverse topics, such as exceptions, functions, classes, the preprocessor, and more.

DOCUMENT Document rules do not allow any kind of deviation, formal or otherwise. These rules impose documentation requirements, such as documenting the use of static analysis tools, floating point arithmetic, assembly, or character encodings.

Each rule provides the reasoning behind it, plus examples and exceptions, if applicable. Even though the rules themselves do not cover it, the standard also provides guidelines regarding developer training, developement tools, source metrics, testing, how to raise deviations against the rules, and how to claim compliance with the standard.

2.2.5 *JSF Air Vehicle C++ Coding Standard*

The Joint Strike Fighter Air Vehicle C++ Coding Standard (Lockheed Martin Corporation, 2005) was published in 2005, making it one of the first professional C++ coding standards. As implied by its name, this coding standard was initially put together for use on the Joint Strike Fighter projects, whose software is made for air vehicles. This standard was based on previous safety-critical coding standards for C, such as MISRA C, and other literature on C++ best practices. It gathered experts in the C++

language to propose and evaluate its rules, one of them being Bjarne Stroustrup[6], the designer of the C++ programming language.

JSF AV C++ was made to be a strict standard that forms a safe subset of C++. It strives to produce code that is reliable, portable, maintainable, testable, reusable, extensible and readable. Due to its strictness and carefully picked rules, not only is it required for air vehicle software development, as it is also recommended for other high-reliability software, as the standard states. In fact, some standards, and standard revisions that came after it, reference its rules, as is the case of HIC++, discussed in Section 2.2.3. This standard was also part of research in coding standards conducted at NASA, as said by Goforth (2013). From it, another coding standard was made, the Orion Coding Standard, used in a NASA flight software project.

The rules in this standard cover many topics of the C++ programming language, and the standard organises them by the topics they cover. These include libraries, classes, templates, functions, and expressions, among others. Conversely to HIC++ and MISRA C++, JSF AV C++ includes explicit rules regarding testing and style conventions. Also conversely to these standards, JSF AV C++ prohibits some features of the C++ language, such as C++ exceptions. Besides organization by topic, the standard also classifies its rules by three levels of compliance, defined as follows.

SHOULD RULES Rules with the word *should* are regarded as advisory. In order to deviate, the developer must receive approval from the software engineering lead.

WILL RULES Rules with the word *will* are mandatory, although they do not require verification. In order to deviate, the developer must receive approval from the software engineering lead and the software product manager.

SHALL RULES Rules with the word *shall* are mandatory and must be verified. In order to deviate, the developer must receive approval from the software engineering lead and the software product manager, and must also document the deviation in the file that it occurs.

For each rule, the standard provides a rationale, references to MISRA rules, and examples and exceptions when applicable. Anderson (2008) classified these rationales into seven categories: *clarity*, *predictability*, *simplicity*, *defense*, *compliance*, *process*, and *performance*, with clarity being the dominant rationale behind a rule. He also analyses the rule overlap between JSF AV C++ and other safety-critical standards.

2.2.6 *CERT C++ Coding Standard*

The CERT C++ Coding Standard[7] was published by the CERT Division[8] of the Software Engineering Institute (SEI) at Carnegie Mellon University, a group of experts in software security, and it is a

6 http://www.stroustrup.com/index.html
7 https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637
8 http://www.cert.org/

continuous work in progress. It is a community effort, and so its rules are freely available for use, feedback and improvement. This standard takes a slightly different route from other standards, in that its primary concern is software security. Since it is a work in progress, it is also more up to date than other standards, in that it targets the C++14 update of the language.

Programs written in C++ are prone to have vulnerabilities that can often be traced back to common programming errors. Seacord (2005), one of the authors of the standard and a member of CERT, identifies many of these vulnerabilities and common errors, and provides guidelines to avoid them. Most coding standards also provide guidelines to avoid vulnerabilities, but only from a safety perspective. CERT considered that this is insufficient, given the uprise in security attacks and exploits in the software industry, and so they took the opportunity to define a secure coding standard (Seacord, 2006; Moore and Seacord, 2007). Unlike most coding standards, that aim for readability, portability or maintainability, the CERT C++ coding standard strives for software *security*, *dependability*, *trustworthiness* and *survivability*.

Table 4 illustrates a rule from CERT C++ that is not present in any of the remaining standards studied here. This rule addresses the use of algorithmic pseudorandom number generators (PRNG). In particular, it addresses the use of any PRNG that allows *seeding*, the ability to set the PRNG's initial state. Using a PRNG multiple times with the same initial state yields the same sequence of random numbers. This is considered a vulnerability, for instance in security protocols, since an attacker is able to predict the sequence of numbers after the first run of the program, effectively breaking the protocol. A way to assure security is to avoid any predictable or controllable source for the seed (such as the current time, or the process identifier), and instead use a sufficiently random source for the seed (such as a hardware based random number generator) and reset this seed periodically.

Rule MSC32-CPP	**Required**
Ensure your random number generator is properly seeded.	

Table 4.: Example of a CERT rule concerning security.

This is the most detailed coding standard analysed here, in terms of rule structure. As said by Seacord (2006), the standard makes a clear distinction between *rules* and *recommendations*. In essence, rules address dangerous or exploitable features, and adherence is mandatory to claim compliance. Recommendations are guidelines that provide good practices, and should generally be followed, although adherence is not required. The standard then goes further and provides a risk assessment section for each guideline, where a guideline is assigned a level and priority, based on *severity*, *likelihood* and *remediation cost* of the addressed issue. This allows a software product to claim compliance with the standard by levels, where Level 1 compliance is the lowest and Level 3 compliance is full compliance.

With respect to the contents of the rules, the standard organises them by topic, as is common practice in most standards. These topics include the C++ preprocessor, expressions, memory management, error handling, concurrency, among others. For each rule and recommendation, besides the risk assessment, the standard provides a rationale, examples and exceptions (if applicable), references to other standards and bibliography, and a list of tools that support automated verification of the guideline.

2.2.7 *JPL C Coding Standard*

The JPL C Coding Standard (Jet Propulsion Laboratory, 2009), developed by NASA's Jet Propulsion Laboratory, stands out from the other standards analysed in this project, because it is a coding standard for the C programming language, as opposed to C++. While its guidelines will not cover C++'s features, they are still useful guidelines in general. This standard is an effort to provide guidelines for mission critical flight software used at JPL, for the C language, and it is strongly based on the MISRA C Coding Standard (MISRA, 2004), and Holzmann (2006). Both references provide robust guidelines for safety-critical software, but they do not address multi-threaded software. This standard aims to fill that gap.

The standard covers the features of the C language, leaving out of its scope style conventions, tools, test requirements, and other issues that are not directly related to code. Based on MISRA C, the standard defines *shall* rules (requirements that must be verified) and *should* rules (deviations are allowed, when adequately justified). It then defines six levels of compliance, where each level addresses one section of the rules in the document. The first four levels concern language compliance, predictable execution, defensive coding and code clarity, and are, in essence, an extension of the rules in Holzmann (2006). The two last levels are related to compliance with MISRA C's *shall* and *should* rules.

This is a very strict standard, that highly promotes static verification of the code. To this end, it inhibits potentially hazardous behaviour, such as recursion, dynamic memory allocation, or loops without a statically determined upper bound.

2.3 STATIC ANALYSIS TOOLS

It has been established that the adoption of coding standards in a software project is generally advantageous. More so when the standards form safe subsets of the programming languages, and the safety and reliability of the software product is of special concern. However, manually verifying compliance with the standards is a daunting task, for any non-trivial project, if not even impossible in a reasonable time period. To counter this obstacle, many tools in the field of software static analysis automate compliance verification, when the rules do not require human analysis. For instance, a common naming convention is for names to be meaningful and descriptive. This is a type of rule that tools cannot generally verify.

In the case of C++, the programming language of interest in this project, given its widespread use, many static analysis tools capable of checking compliance emerged, with some of them being commercial, and others being free. The commercial tools are quite more capable than the free tools, and they often pack many more analysis and management features. Commercial tools also directly state their support for specific standards, while free tools are more extensible and of general utility. This section presents an overview of some relevant free and commercial tools, and how they support compliance verification.

2.3.1 *Free Tools*

Google provides a tool to verify compliance with their own coding standard, Cpplint[9]. Cpplint is a Python script that reads a source code file and identifies coding style errors. It cannot verify every rule from the standard, and it is known to report both false positives and false negatives, although it provides a mechanism to annotate the source code and ignore false positives. Since it is an open source script, it can naturally be adapted and extended to custom needs.

Clang[10] is a compiler front-end for the LLVM compiler, and supports C, C++, Objective C and Objective C++ programming languages. It was originally developed by Apple, and later became open source. Clang by itself does not provide compliance verification, and its *Clang Static Analyzer* can only detect some common errors. Instead, Clang is divided into a set of libraries that perform parsing, semantic analysis, preprocessing and other compilation functions. These libraries, then, provide the support for static analysis tools to be built on top of them. In other words, Clang parses the programs, and provides other tools or extensions the data they need to check the rules they define.

Vera++[11] is a free programmable tool for verification, analysis and transformation of C++ source code. In essence, it is a parser for C++ source files which presents the results of parsing to user-defined scripts. The scripts are the units responsible for processing and analysing the parsed source file names, source lines and tokens for each file, making Vera++ a very flexible tool. Compliance with coding standards, for instance, can be expressed in terms of rules, each rule being defined by a separate script. The scripts can access the parsed information and perform actions related to the given rule. The user can ask to run any given script or some defined set of scripts in a single program execution, and the scripts are programmable in the TCL, Python and Lua programming languages.

Another free static analysis tool of interest is Cppcheck[12], a tool for C and C++ code that performs various checks beyond typical compiler capabilities. Its checks include memory leaks, index bounds, dereferences to null pointers, unused or uninitialized variables, use of deprecated code, and more. This tool is widely used, even in relatively large projects, such as OpenOffice and the Linux kernel. Its usability is boosted by the fact that it comes integrated with other development tools, either out-of-the-box or as a plug-in. While it is not bound to any particular coding standard, or even to the concept of coding standards, some coding standards define rules against unused variables, for instance. As such, its analysis capabilities can be translated to coding rules. Additionally, as is the case with other

9 https://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
10 http://clang.llvm.org/
11 https://bitbucket.org/verateam/vera/wiki/Home
12 http://cppcheck.sourceforge.net/

free tools, Cppcheck is extensible, both with Python scripts and with XML rules, based on regular expressions.

2.3.2 *Commercial Tools*

Goanna[13] is a static analysis tool for C and C++ source code that is available either as a command line tool, or as a plugin for some integrated development environments (IDE). Goanna's analysis engine supports program interprocedural analysis, and performs value analysis on the program's variables, using model checking techniques, to report accurate software quality issues. From the coding standards studied here, Goanna directly supports MISRA C++ and CERT C++. Since it supports MISRA C, for the C programming language, and the JPL C standard includes MISRA C's rules, Goanna consequently supports part of the JPL C standard.

QA-C++[14] is a static analysis tool for C++ environments, developed by PRQA, the author of the HIC++ standard, providing compliance verification, dataflow analysis and code metrics. Its compliance packages include the HIC++, MISRA C++ and JSF AV C++ coding standards. It also provides configurable metric thresholds, to limit code complexity. More of its features include IDE integration, a source code editor, and mixed projects with both C and C++. This is the tool recommended by ROS[15] to assess code quality, which is related to (but not focused on) coding standards, and includes code metrics, for instance.

The LDRA Testbed[16] is part of a tool suite that provides static and dynamic analysis engines, and an interactive environment with code visualisation. LDRA's code visualisation not only shows coding standards compliance and quality metrics, but also shows where the source code deviates from a standard. LDRA's coding standards compliance tools allow the user to select combinations of standards, rule subsets, and individual rules. This allows the tool to check compliance of a single code base against multiple standards. It supports all coding standards studied here, except for ROS's and Google's coding standards.

ECLAIR[17] is a platform for the analysis, verification, testing and transformation of C and C++ programs. It covers a wide variety of coding standards, including all standards studied here, except for ROS's and Google's standards. ECLAIR focuses on exact results, excluding false positives and false negatives, when rules are decidable. For undecidable rules, it provides tradeoffs between computational complexity, number of false positives and number of false negatives. It also provides code metrics, that can be incrementally reported, showing where in the code the value was computed, or aggregated (e.g., maximized, summed, averaged) over a single function, translation unit, program or the whole project. At the verification level, ECLAIR comes with constraint propagation, symbolic

13 `http://redlizards.com/`
14 `http://www.programmingresearch.com/products/qacpp/`
15 `http://wiki.ros.org/code_quality/installation`
16 `http://www.ldra.com/en/testbed-tbvision`
17 `http://bugseng.com/products/eclair`

model checking, and abstract interpretation engines. With respect to testing, this tool can automatically synthesize sets of unit test inputs.

In summary, commercial tools often target high-quality software systems, and thus they provide no direct support for ROS's or Google's coding standards, except for any rules that these standards may have in common with other standards the tools support. Table 5 shows the support of the commercial tools mentioned above for the coding standards studied in this project.

Tool Name	HIC++	MISRA C++	JSF AV C++	CERT C++	JPL C
Goanna	No	Yes	No	Yes	MISRA C Rules
QA-C++	Yes	Yes	Yes	No	No
LDRA Testbed	Yes	Yes	Yes	Yes	Yes
ECLAIR	Yes	Yes	Yes	Yes	Yes

Table 5.: Support of paid tools for coding standards.

2.3.3 *SonarQube*

SonarQube[18], formerly *Sonar*, is an open source code quality management platform for software projects, that allows teams to manage, track and improve the quality of their projects. It deserves special consideration in this research project for a number of reasons, one of them being its popularity as a static analysis tool.

SonarQube offers continuous quality analysis of the source code, with detailed dashboards and reports, at the file, module or project levels. Its reports and dashboards can focus on measures, issues, or customised data, and their changes over time, providing a new perspective of the improvement of the team's work. Its analysis capabilities cover code duplication, coding standards, lack of test coverage, potential bugs, code complexity, documentation and design flaws. To ease access to these features, and integration of the tool into a team's development routines, SonarQube also offers integration support with many popular development tools, such as IDEs, build tools and other source code analysis tools.

While SonarQube itself is free, in order to support more than twenty programming languages, it adopts a plug-in system, and some of the plug-ins are commercial. In particular, the plug-in to analyse C++ code is commercial, and supports the MISRA C and MISRA C++ coding standards. Anyone can develop plug-ins for SonarQube, however, and this presents an interesting opportunity to extend an already popular tool with custom analysis rules and reports for ROS projects. While this is not the current route this project is following, this idea is kept as an open possibility for future work.

18 http://www.sonarqube.org/

2.4 ROBOT OPERATING SYSTEM

Writing robotics software is hard. It has to cross a whole stack of abstraction levels, from the hardware drivers to artificial intelligence and robotics algorithms. As more and more robotics researchers realized this, more effort was put in designing and implementing a common middleware layer, so the prototyping and development of robot applications could take a leap, reducing time and costs. Some of these middleware frameworks are successful open source projects with growing communities, as seen in a survey by Elkady and Sobh (2012). The Robot Operating System is one such project, and it is the focus of this project for its large community and its increasing maturity.

The Robot Operating System[19] (ROS) is an open source system that offers functionalities similar to an operating system for robotics software, in the form of libraries (*packages*), tools and middleware. It was originally created from a collaboration between Willow Garage, a team of experts in robotics, and the Computer Science Department of Stanford University, as presented in Quigley et al. (2009). In 2013, ROS's stewardship transitioned to the non-profit Open Source Robotics Foundation, who ensures that it remains free, open, and easy to share. ROS was designed with the goals of being a thin, multi-lingual, open source framework, consisting on the interoperability of many small tools and components, and based on a peer-to-peer architecture. This modularity allows users to determine to what extent they need ROS's functionalities, and allows ROS to interoperate with other robotics frameworks. Although its main development languages are C++ and Python, it also offers support for Java, LISP and others, to some extent, with support for C coming in ROS 2. This language neutrality displayed by ROS is backed by its tool and peer-to-peer philosophy.

ROS uses *Git* repositories for its source code, hosting them freely on GitHub[20], and the community follows this rule when contributing to ROS. In fact, ROS has a distribution repository[21] which hosts *distribution files* for its various release versions. These distribution files contain all repositories and packages featured in a release, the addresses of the remote repositories, the current version of each package and the supported platforms for that ROS release. GitHub is, thus, a central service for the ROS community. For ROS Indigo Igloo, the latest long-term support release of ROS, the distribution file features about 700 repositories. Each repository may contain multiple packages, but packages are considered the smallest build and release units in a ROS environment. Some packages are labelled as *metapackages*, which is in essence a package of packages, an aggregation unit. Although a package is a module, a logical component that depends on other components, packages could be categorised by three main types: *core* packages, *libraries* and *applications*. Core packages are the main packages that compose the ROS environment, tools and libraries. These provide the barebones of a ROS system and its networks, and make a minimal ROS installation. Libraries may be developed by the community and provide general utility, without a specific purpose. These are to be used as components for the

19 `http://www.ros.org/`
20 `http://github.com/`
21 `https://github.com/ros/rosdistro`

applications, packages that represent a robotics system, the high level controller of a robot with some logic or instructions to execute (for instance, making a robot move in circles).

A typical ROS system consists on a number of processes, or *nodes*, connected under a peer-to-peer topology. The nodes then communicate with each other following a *publish-subscribe* model, where some nodes subscribe to topics of their interest, and other nodes publish messages (data) under the appropriate topics (Quigley et al., 2009; Elkady and Sobh, 2012; Iñigo Blasco et al., 2012). These nodes can represent various entities or fulfill various purposes. For instance, a sensor's driver might be implemented as a node, in which case it would publish the sensor's data as messages under some topic, while subscriber nodes would receive the messages and consume the sensor's data. The messages are specified in a neutral interface definition language which is then used to generate code for the languages supported by ROS. Another means of communication between nodes is the use of remote procedure calls (*services* in ROS terminology), which are very similar to topics, but the messages are defined in request-reply pairs. At a finer grained level, sometimes a node can represent an entity that is too large or too abstract, and further division is needed. This is accomplished using *nodelets*, subnodes inside a node. Since each node is given its own process, nodelets are, instead, given a thread pool. Nodelets favor the multithreading capabilities of the hardware, and have intra-process communication at their disposal, reducing communication overhead. All these features make ROS incredibly flexible and adaptable to heterogeneous environments and architectures. Moreover, ROS is designed in such a way as to be compatible with robot simulators, such as Gazebo[22], which makes testing possible, to an extent, when a physical robot is not available. Figure 3 illustrates how these concepts connect to each other in a typical ROS system.

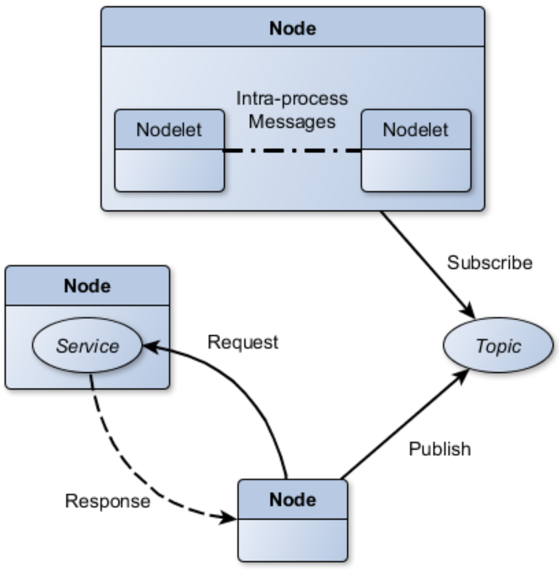


Figure 3.: Simplified diagram of a ROS system.

22 http://gazebosim.org/

As an example of this architecture, we can see it in practice in Kobuki[23]. Kobuki is a low-cost mobile robot base, implemented using ROS, and designed with education and research in robotics in mind. It is used, for instance, as a base for Turtlebot 2[24], a popular ROS robot also used in research and illustrated in figure 4. Kobuki alone already provides a set of interesting features, such as keyboard remote operation, auto-docking and a safety controller, that overrides movement commands based on sensor information. For instance, when launching in the Gazebo simulator a minimal Kobuki application that makes the robot walk around randomly, while also allowing keyboard remote operation, it is possible to inspect (using the *rosnode* tool, provided by ROS) that there are eight active nodes. One of the nodes is the simulator itself, that mimics the low level components of the robot and subscribes to movement commands. There is a node to represent Kobuki's base which makes use of a number of nodelets, to reduce the overhead of internal communication, and provides the commands for the low level components. Above this level, there are nodes for keyboard operation, prioritization of commands, making the robot walk randomly, and essentially every loaded application. Kobuki's documentation[25] provides a diagram (shown in figure 5) of a similar scenario, in which we can see keyboard and Android operation, prioritization of commands (*cmd_vel_mux*) and a safety controller that moves or stops the robot based on sensor information (*kobuki_safety_controller*).

Figure 4.: Turtlebot 2, a robot that uses Kobuki as a mobile base.

The main application field of ROS is in service robotics, with examples of robots folding towels, cleaning up the office[26], or cooking. However, ROS has also been used in applications as diverse as autonomous cars[27], micro air vehicles, and surgical robots (Hannaford et al., 2013). A project dedicated to industrial robots (ROS-Industrial[28]) has also emerged, aiming to extend ROS's capabilities to manufacturing.

23 http://kobuki.yujinrobot.com/home-en/about/
24 http://kobuki.yujinrobot.com/home-en/about/reference-platforms/turtlebot-2/
25 http://wiki.ros.org/kobuki/Tutorials/Kobuki's%20Control%20System
26 http://www.willowgarage.com/pages/software/ros-platform
27 http://www.ros.org/news/2010/03/robots-using-ros-stanfords-junior.html
28 http://rosindustrial.org/

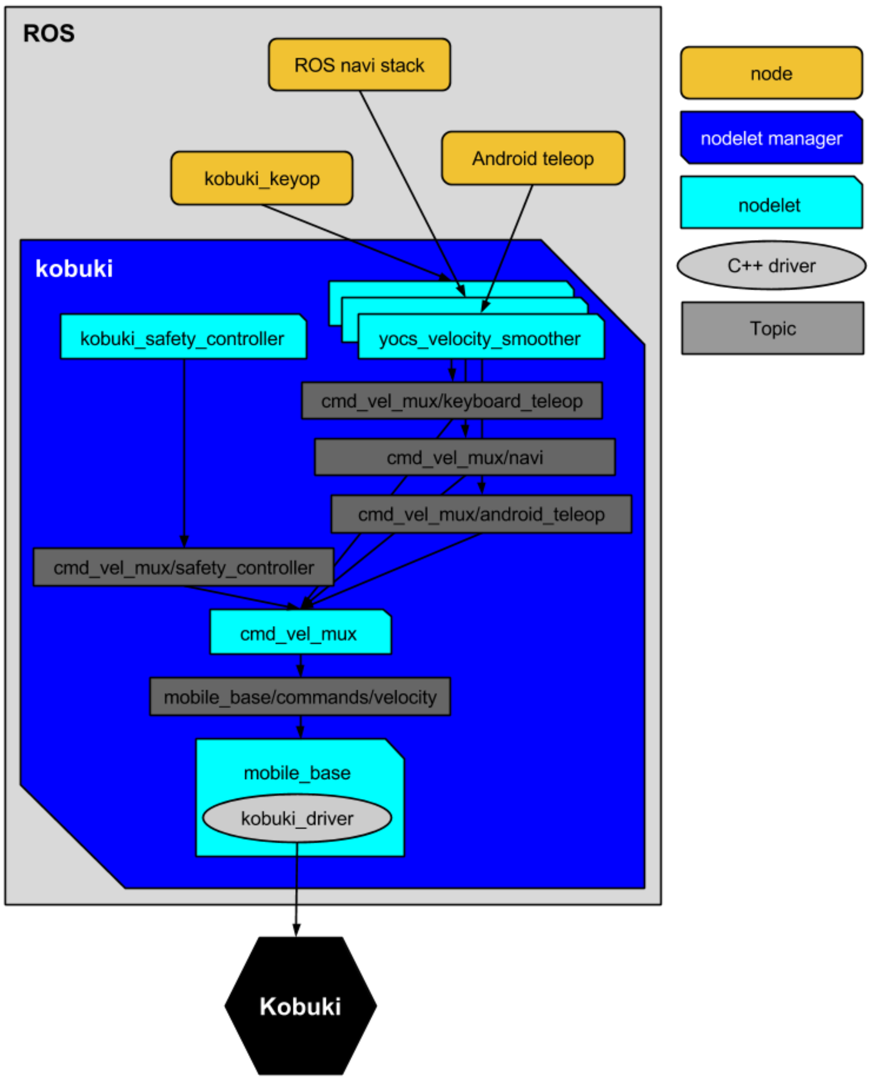


Figure 5.: Kobuki's control system in a basic usage scenario.

2.5 SUMMARY

In this chapter, we have seen how coding standards play a part in high quality software, and how these have been adopted in ROS, the robot middleware studied in this project.

Coding rules are important not only in defining a safe subset of a programming language, but also for their effect on people. Having a clear set of rules from end to end of the developing process helps people understand and review each other's work, improves readability and consistency of the code base, and lets people focus more on fulfilling requirements, rather than how to write a particular solution. Furthermore, when assembling a standard, coding rules can be assigned various levels of compliance and importance, often being labelled, at least, as either mandatory or advisory.

Given the broad context and possibilities for coding rules, it is no wonder that different standards may opt to focus on different subjects. We have seen how Google and ROS define a sort of style guide, emphasizing formatting and readability, while HIC++ and MISRA C++, for instance, are much stricter standards with safety and quality concerns. It is also visible in the set of considered standards that, in general, the more strict and reliability oriented a standard is, the better the document is structured and organised. It is clear that, in order to pursue a quality oriented standard for ROS, it needs to take much from the latter group of standards. Verification of compliance, however, is a complex subject on its own. There are excellent and very capable tools, but these are commercial. Free tools for C++ static analysis adopt a generic approach, being rather limited *out-of-the-box* and acting as a sort of parser, while providing an extensible interface for the end user. In any case, but more so when limited to free tools, it is probable that various tools have to be used in conjunction, in order to have a satisfying analysis coverage. With such diverse interfaces and outputs, a single unified platform, such as the one produced in this project, is of significant help when putting together a tool set.

Regarding ROS, we now know it is a package based system. Each package represents, essentially, a library or component. This approach means that both the system and its applications are easily extensible, and adaptable to various contexts, which are great qualities for an open source project. As for the implementation of a ROS system, it is, in general, a short scale distributed system, in the sense that each component has its own process, a *node*, and these processes communicate under a *publish-subscribe* model. That is, when a node needs to share information, it sends a message to a certain *topic*, and then the middleware handles the delivery of the message to all nodes that are subscribed to that topic.

Summing up, this chapter covers about every major topic and concept of relevance to this project. It provides an introduction to software quality based on coding standards, some popular coding standards for the C++ language, tools to automate verification, and the Robot Operating System.

3

CONTRIBUTION

This research project focuses on the code quality of projects developed for the ROS middleware, and ways to assess these quality measurements. From the previous study on coding rules and standards, it is clear that the only feasible way to measure quality of a non-trivial project is by using automated tools. Hence the study of existing analysis tools and their adaptability and extensibility to analyse ROS applications. Considering the limitations of the existing tools and the lack of ROS specific tools, the development of a unified and generic tool (or, rather, a toolset) ensued, taking place as one of the main contributions of the project. The main features this tool aims for are automatic source code fetching, extensibility, analysis and quality measurements of the source code (especially with respect to coding standards) through a *plug-in* system, and interactive reports of the analysis results using graphic models, such as diagrams. Figure 6 illustrates the workflow of the developed tool, including these features, while section 3.1 of this chapter discusses the tool in more detail, encompassing its architecture, its functionality, other related tools, and how to use it in practice.

A case study followed the development of a basic version of the tool, as a way to leverage the tool's capabilities and observe its results in practice. In this case study we work through the adaptation of Cpplint and Cppcheck (both free analysis tools, mentioned in section 2.3.1) into analysis plug-ins for this tool. The former is adapted by altering its source code, while the latter is used as a child process instead. The tool is then used to analyse an existing ROS project. Section 3.2 covers this adaptation process, and an overview of the achieved results.

3.1 ROS STATIC ANALYSIS TOOL

There are many tools to perform static analysis on C++ and Python code, the main development languages in ROS. However, as stated in section 2.3, when discussing some relevant tools, a considerable part of these tools are paid services, and, even though they achieve excellent analysis results, none of those tools is tailored to consider specific details of ROS. When developing a ROS application or robot, one cannot ignore the concept of *package*, a way to organize source code files and to establish dependencies between source files, a module, in essence. Packages are the smallest build and release units in ROS software development. Being able to analyse code quality at function and file scopes is a necessity, but being able to incorporate the package scope into the analysis allows for more rule

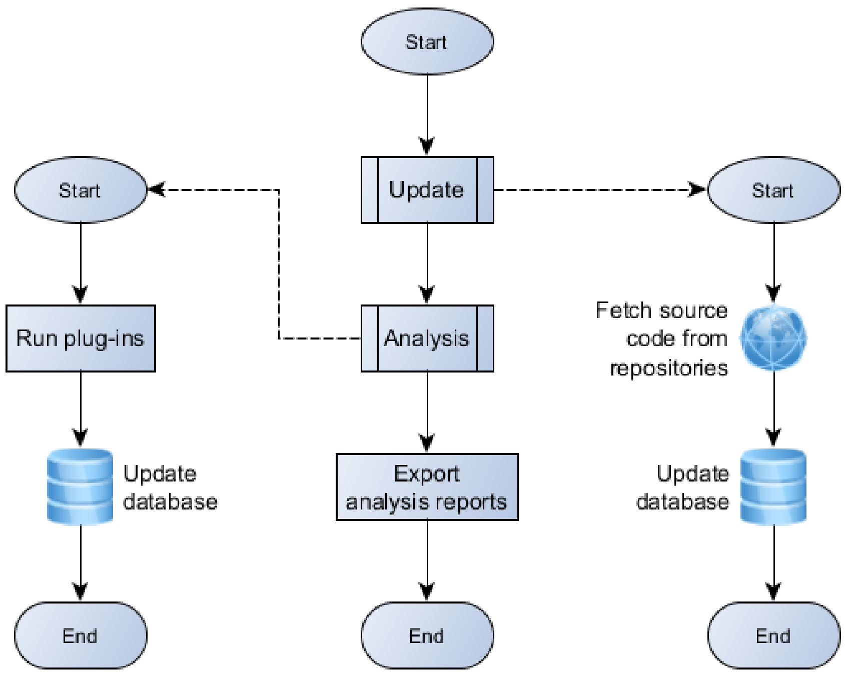


Figure 6.: Simplified workflow of the developed tool.

definitions, and a more refined analysis report. Besides, some conventions in the ROS coding and quality standards apply to packages, reinforcing their importance. With this in mind, developing a new generic open source tool, specifically tailored for ROS projects, becomes justifiable.

Even though this project focuses on coding standards, software quality standards go beyond coding conventions, to the domain of quality metrics, for instance. This tool should be suitable to as many types of quality analysis as possible, and so a static analysis model based on rules and compliance, as is generally the case with coding standards, was found to be the most flexible. Some relevant quality standards already assume a compliance model, by declaring minimum and maximum allowed values for each considered quality metric (thus, forming a rule: a metric must be within the allowed range of values). By also assuming a compliance model, this tool can easily tell whether a package is compliant with a given quality standard, it is a matter of whether there are any violations of the standard's rules. Other practical implications include a simplification of data structures (only non-compliance occurrences need to be registered) and homogeneity of analysis reports.

The idea of performing some sort of quality analysis on ROS code and reporting the results through interactive graphics is not new. The RosEco project[1] already does this, to a limited extent. It narrows its analysis to a few quality metrics, based on how active the source repositories are, or how many dependencies a package has, and then assigns a score to each package, for each metric. This all happens by looking at package manifests and by querying the source repositories, there is no quality

1 http://http404error.github.io/roseco/

analysis at the source code level. The results are then rendered using a web browser, in the form of a directed graph, where each node represents a ROS package. The nodes' colour depends on their metric scores, and the user interface offers some controllers for the user to focus the graph on a specific package and its dependencies, or to change the selected quality metric, affecting how the nodes are coloured. Since RosEco is an open source project, with a permissive license and equipped mostly with features this project aims to achieve, it provided a solid start point to the development of a new, more sophisticated tool. Some of the main differences between this tool and RosEco include:

- automatic fetching of source code from remote version control repositories;
- customisable package sets for analysis, not limited to the official ROS distribution;
- source code static analysis, based on a set of rules;
- customisable analysis rules, actions, and outputs through a *plug-in* system;
- a refined visualiser, with more user control knobs and extended analysis reports, besides graph colouring.

Despite their differences, this tool kept some characteristics from RosEco, mostly from a software architecture point of view. In particular, it kept a clear division between the component responsible for performing the analysis and the component responsible for rendering the analysis results, to the point that each of the components is implemented in different programming languages. Subsections 3.1.1 and 3.1.2 describe these components, respectively, while subsection 3.1.3 provides guidance in using and extending the tool.

3.1.1 *Analysis Component*

The analysis component is the main component of the tool, and it is implemented in Python, as is the case with RosEco. It runs as a console program, and it is responsible for everything but data presentation. This includes managing source code repositories, running static analysis, and keeping a database updated. Its execution is phased in startup operations, and then three stages, the *update* stage, the *analysis* stage and the *export* stage.

During startup operations, the tool parses arguments provided by the user, and then loads a configuration file. Through the startup arguments, users control which of the following stages will be executed, and to what extent of their functionality. For instance, users can explicitly disable all operations that require a network connection. In the current version of the tool, the configuration file is used only to provide a list of the plug-ins that should be dynamically loaded for later execution. This concept of extending the tool via plug-ins is supported only for the analysis stage, since it is the only stage that is entirely dependent of each user's needs, and thus needs to adapt in order for the tool to be generic and flexible. A possible upgrade for the tool is to extend its plug-in support to the remaining stages.

Following startup operations, the first stage of execution is the update stage, during which the tool attempts to update its database and local source code files. In order to do so, the tool reads a distribution file and a filter file. The distribution file is a concept taken from ROS. It is a file that contains all known packages, and the address of their remote repositories, on GitHub. The filter file is a smaller file that defines the set of packages that should be considered and analysed by the tool, in the current execution. With this information, the tool proceeds to update the local source code and its repository information. It either pulls changes from the remote repositories, or clones them entirely if no local version exists. The repository information, although not currently used, was kept from the RosEco project. It includes contributors to the repositories, repository issues, dates and statistics. These operations require a network connection, and they can be skipped by users.

During the update stage, the tool also updates its analysis rules, so that it knows which rules are available during analysis. The rules are declared in an external file, as a way to improve maintainability and extensibility. This external file follows the YAML[2] format, in order to keep it human-readable. Listing 3.1 illustrates how rules are currently declared. For each rule, there is an identifier, a description, a scope and a set of *tags*. The description and scope hint at what the rule is, and how it is applied. In the example given, a *file* scope means that the rule is meant to be applied to a whole source code file. The tags are user-defined labels with no practical application in source code analysis. They serve as a way to categorise, filter and sort rules, but their main application is in the visualisation component, described in subsection 3.1.2.

```
%YAML 1.1
---
-
    id:             1
    name:           MIN_COM_RATIO
    scope:          file
    description:    "Minimum lines of comments: 20%"
    tags:
        - metrics
        - nasa-satc
        - his
        - uai
        - ros
        - comments
        - comment-ratio
-
    id:             2
    name:           MAX_COM_RATIO
    scope:          file
    description:    "Maximum lines of comments: 30%"
```

2 http://yaml.org/

```
    tags:
        - metrics
        - nasa-satc
        - comments
        - comment-ratio
```

Listing 3.1: Example of rule declaration, with two of the rules currently in use.

The tool verifies the source code in the analysis stage, looking for violations against the defined set of rules. This may involve calculating code quality metrics, or checking that coding conventions are respected, for instance. Implementing the static analysis checks for every rule one might be interested in analysing is beyond the scope of this project, and it is not how this tool is intended to be used. Instead, this tool supports extensibility through the aforementioned plug-ins.

A plug-in, in this context, is a subcomponent, dynamically loaded, responsible for performing a part of the analysis process. The plug-in has access to a small interface, provided by the main component, that abstracts the internal database and data structures, while allowing the plug-in to register its results. Listing 3.2 illustrates how this interface is implemented as a Python class. For each non-compliance occurrence, a plug-in registers the broken rule, the ROS package and, if possible, the source file, line number and function name.

```python
class PluginContext:
    """Constructor, internal data structures and helper functions omitted."""

    def getRuleInfo(self, name=None):
        """Provides database entries for registered rules."""

    def getPackageInfo(self):
        """Provides database entries for registered packages."""

    def getFileInfo(self, ext=None):
        """Provides database entries for source code files."""

    def writeNonCompliance(self, rule_id, package_id, file_id = None,
            line = None, function = None, comment = None):
        """Registers a non-compliance occurrence.
        The rule and package identifiers are mandatory.
        Optional fields include the source file, line number,
        function name, and a custom comment with additional information."""
```

Listing 3.2: Summary of the Python class that provides an interface between the main tool and plug-ins.

The current philosophy, regarding plug-ins and rule verification, is to have the plug-ins be a bridge between free analysis tools, and this tool's database, as a way to reuse the capabilities of other tools, while adapting their results to fit the data structures in use here. However, free static analysis and code quality tools are limited. As the supported rules become more complex, or uncommon in other tools, some analysis checks might still have to be manually implemented in a plug-in.

A result of separating rule declaration from rule verification (*i.e.*, updating the rules database, as opposed to verifying the source code) is that the data structure for the analysis rules does not need to hold any information on how compliance with the rules is verified, as seen in the example above. Another result of this plug-in model is that the source code of the tool itself never changes to accommodate new rules. Only the plug-ins and the configuration files loaded on startup need to adapt.

The final stage of execution for the analysis tool exports data files, with data from the database, as a way to interoperate with other tools. In particular, this functionality is used to interoperate with the visualisation component of this tool, that relies on these data files to present analysis results to the user. In the current version of the tool, it exports a file with the rule set in use, a file with a summary of the packages included in the initial filter file, and then exports a file with analysis details for each of those packages. The package summaries contain general details about each package, such as their name, their description, contributors, dependencies on other packages, and a count of rule violations per rule. The analysis files contain information about each rule violation in the package, as detailed as possible, with the same level of detail that is available to plug-ins when registering occurrences (rule, package, file name, line number, function name and a custom comment), plus the list of tags associated with the broken rule. All exported files are under the JSON[3] format, as illustrated in listings 3.3 and 3.4.

```
[
  {
    "Name": "kobuki_keyop",
    "Metapackage": false,
    "Description": "Keyboard teleoperation for Kobuki: relays commands from a
        keyboard to Kobuki.",
    "Wiki": "http://ros.org/wiki/kobuki_keyop",
    "Repositories": ["yujinrobot-release/kobuki-release", "yujinrobot/kobuki"],
    "Authors": ["Daniel Stonier"],
    "Maintainers": ["Daniel Stonier"],
    "Analysis": {
      "Noncompliance": {
        "2":        2,
        "3":        2,
        "18":       1,
        "20":       10,
        "24":       13,
        "10001":    15,
        "10004":    55,
        "10010":    2,
        "10025":    12,
        "10106":    6,
        "10206":    4,
        "10208":    1
```

3 http://json.org/

```
      }
    },
    "Edge": ["yocs_cmd_vel_mux", "yocs_velocity_smoother"]
  }
]
```

Listing 3.3: Example of an exported package summary with one package.

```
[
  {
    "rule": "Maximum cyclomatic complexity: 10",
    "file": "command.cpp",
    "line": 210,
    "function": "serialise",
    "comment": "Cyclomatic complexity is greater than 10.",
    "tags": ["metrics","nasa-satc","his","cyclomatic-complexity"]
  }
]
```

Listing 3.4: Example of exported analysis details for a package.

3.1.2 *Graphic Component*

This data presentation component is implemented in HTML and JavaScript, as is the case with RosEco's graph viewer component. RosEco, in turn, is also based on another project, the X-Trace[4] project. This component is, in essence, a web application that displays the data exported from the analysis component, using graphs and other visual representations. It resorts to common web development libraries, such as *Angular* and *Angular Mobile* to provide structure to the application, and *D3* and *Dagre* for graph rendering.

From the package summaries exported by the analysis tool, this component builds a diagram, in the form of a directed graph, where each node represents a package, and each edge represents a package dependency. These summaries contain a count of non-compliance occurrences detected during analysis, per rule, for each package. Such information is used to colour the nodes, in a lightness scheme in which darker nodes have more reported non-compliance occurrences. Figure 7 shows an example of a graph, as rendered by the application.

Each analysis rule has a set of associated tags (or labels), for categorisation purposes. Besides rule categorisation, the tag system serves a second purpose as user-defined filters. The application allows the user to filter the reported rule violations by a set of tags, as shown in figure 8, so that, for a given package, it is possible to know both the total number of rule violations, and the total number of

4 http://www.x-trace.net/

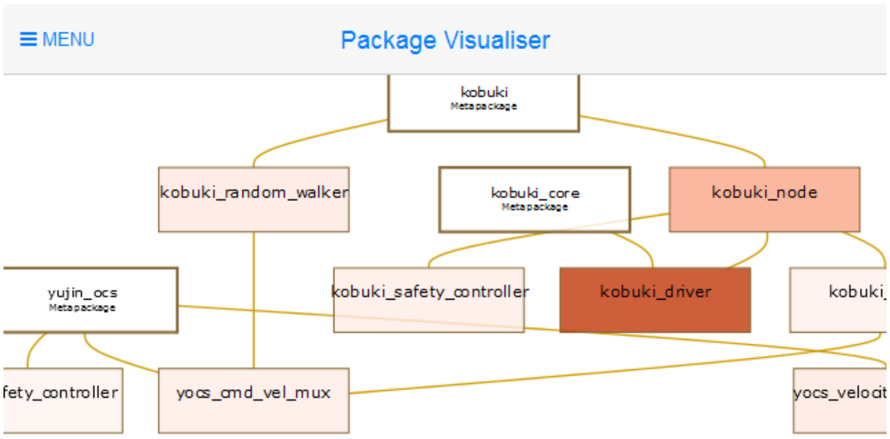


Figure 7.: Standard graph view. Darker nodes represent more non-compliant packages.

violations under some specific labels. Filtering works both ways, so that it is also possible to exclude certain tags from the reported rule violations.

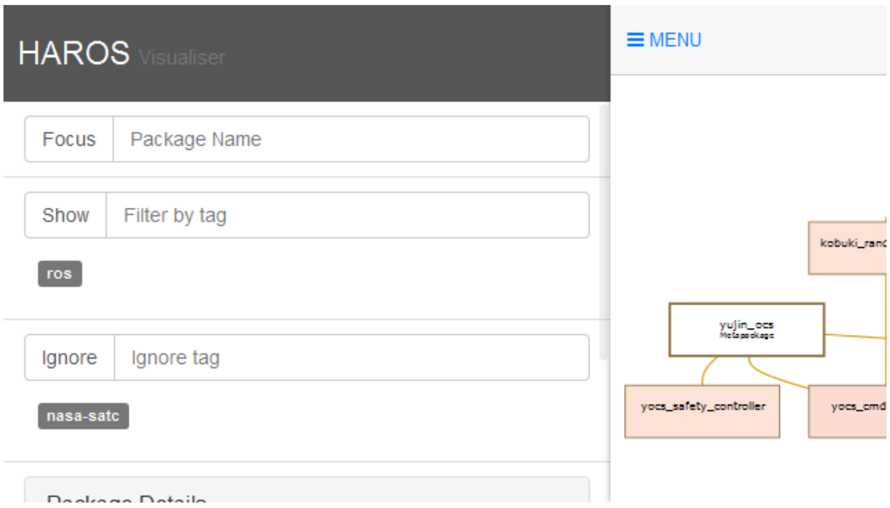
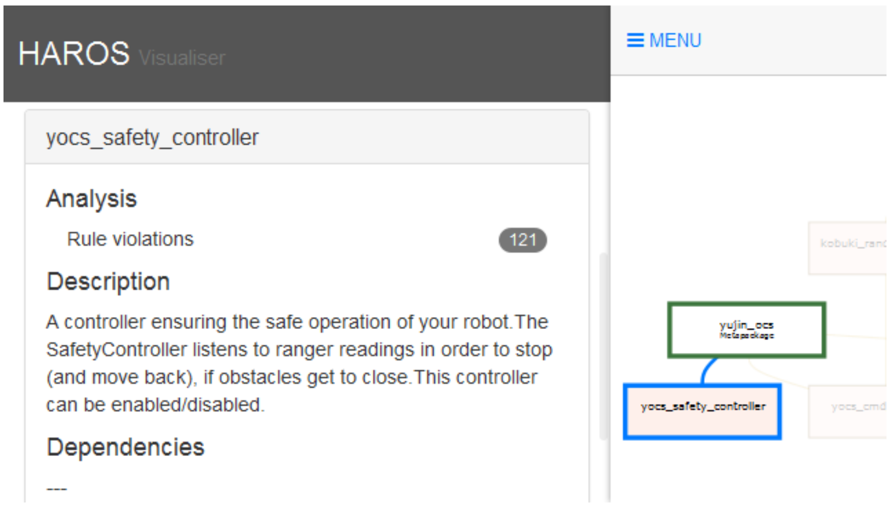


Figure 8.: Side bar menu with two active tag filters, *ros* and *nasa-satc*.

The component adjusts node colours to display the intersection of violations with the active sets of tags. That is, when some tag filters are set, all nodes are drawn in the diagram, but each node is coloured based on the sum of rule violations with any of the tags in the positive filter, minus the rule violations with any tags in the ignore filter. Otherwise, when no tag filters are defined, the colours represent the total number of rule violations for each package – which is equivalent to having every possible tag in the positive filter. Take, for instance, a positive filter of a, an ignore filter of b, and a package with violations in rules $r1$, $r2$ and $r3$. Assume that $r1$ is tagged as a, $r2$ is tagged as b, and

$r3$ is tagged as both a and b. With these filters in place, such package would be coloured based on the value $violations(r1)$. Rules $r2$ and $r3$ would be ignored, in this case, because the tag b is being ignored. If no filters were set, the package would be coloured based on the total number of violations, $violations(r1) + violations(r2) + violations(r3)$. The lightness of the resulting colour depends on the difference between this sum and the maximum value in the set of analysed packages. In short, the colouring system is relative. The darkest node does not necessarily represent a package with many rule violations, it just represents the package with most rule violations in the graph.

Regarding user interaction, the graph allows zooming and panning, to accommodate various graph sizes under various screen resolutions, and node selection. When a node is selected, the node, its dependencies and the nodes that depend on it are highlighted and detailed information about the selected node is shown to the user. This information includes the package name, the package description, a list of dependencies, and the current value of non-compliance occurrences (taking tag filters into account). Figure 9 depicts the effects of node selection on the graph and on the side bar. Additionally, users can focus the graph on a node of their choice. Setting the focus reduces the visible graph to the focused node, its dependencies and the nodes that depend on it. Clearing the focus renders the graph again with all nodes.



yocs_safety_controller

Analysis

Rule violations 121

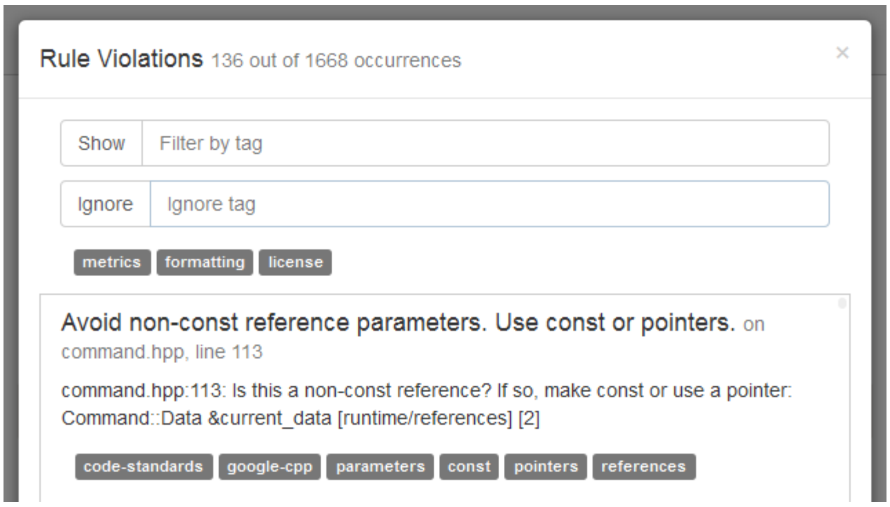
Description

A controller ensuring the safe operation of your robot.The SafetyController listens to ranger readings in order to stop (and move back), if obstacles get to close.This controller can be enabled/disabled.

Dependencies

Figure 9.: Side bar menu with the *yocs_safety_controller* package selected (blue outline).

In its current version, the component provides non-compliance details only at the package scope. For a given package, the user can inspect which rules were violated, along with all the information registered about each violation during the analysis (file name, line number, function name and more, when available), as shown in figure 10. This detailed inspection is also subject to its own tag filters, that work in the same way as in the general graph view. One possible enhancement for future versions is to widen this range of scopes, allowing users to inspect particular files, classes or functions.


Rule Violations 136 out of 1668 occurrences

Show Filter by tag

Ignore Ignore tag

metrics formatting license

Avoid non-const reference parameters. Use const or pointers. on command.hpp, line 113

command.hpp:113: Is this a non-const reference? If so, make const or use a pointer: Command::Data &current_data [runtime/references] [2]

code-standards google-cpp parameters const pointers references

Figure 10.: Non-compliance details for a selected package.

3.1.3 *User Guide*

When using the analyser, users can tune its execution to meet certain needs, or to avoid unneccessary operations. There are mainly two tuning knobs: command line arguments and startup files.

Command line arguments allow the user to skip some stages of execution (see subsection 3.1.1 above), or to skip certain operations in each stage of execution. The arguments are given in an UNIX fashion, using `--argument [options]`, or `-arg [options]` when the argument has a shorthand equivalent. When no arguments are provided, the tool goes through every step of its normal execution flow. The following list describes the available arguments, and explains each of them.

UPDATE The `--update` argument explicitly tells the tool to execute the update stage. It can receive optional arguments to restrict what updates should be performed. The available options are `repos`, to update source code and repository information, `source`, to update just the source code (this is mutually exclusive with `repos`), and `rules`, to update the set of analysis rules. It also provides the shorthand `-u`.

NO-UPDATE The `--no-update` argument skips the execution of the update stage. It is mutually exclusive with `--update`.

NO-NETWORK The `--no-network` argument skips operations that require a network connection, such as updating source code, or repository information. It also provides the shorthand `-n`.

ANALYSE The `--analyse` argument explicitly tells the tool to execute the analysis stage. It can receive optional arguments to restrict the types of analysis performed. Currently, the only supported optional argument is `rules`, to analyse source code non-compliance. It also provides the shorthand `-a`.

NO-ANALYSE The `--no-analyse` argument skips the execution of the analysis stage. It is mutually exclusive with `--analyse`.

EXPORT The `--export` argument explicitly tells the tool to execute the export stage. It can receive optional arguments to restrict what data is exported. The currently supported optional arguments are `packages`, to export package summaries, `rules`, to export the active rule set, and `analysis`, to export analysis results. It also provides the shorthand `-e`.

NO-EXPORT The `--no-export` argument skips the execution of the export stage. It is mutually exclusive with `--export`.

An usage example of these command line arguments is given in listing 3.5. In this example, the tool executes all three stages, but the update stage skips source code and repository information updates, since these require network operations (inhibited by the `-n` argument), and the export stage only exports the detailed analysis results. The analysis stage is unaffected, and, thus, every analysis plug-in is executed.

```
$ python main.py -n -e analysis
```

Listing 3.5: Usage example of the command line arguments supported by the tool.

The startup files include the configurations file, the rules file and the distribution filter file. In the rules file, users can change the set of known analysis rules to the tool. Note, however, that editing or removing existing rules may cause plug-ins to encounter errors or report inaccurate results. The distribution filter file allows users to change the set of analysed packages. The only restriction regarding this filter, is that the packages included in the filter must be included in the complete distribution file. In the configurations file, users can select which plug-ins the tool should load and execute, in their respective execution stages. The only restriction is that plug-in names and file names must match. For instance, a plug-in named *plugin* should be found in a file named *plugin.py*.

Besides using the tool itself, users can extend the tool's functionality, by writing plug-ins for it. Plug-ins start as Python scripts that the tool dynamically loads and runs. All that is required from a plug-in is for it to define a function named `plugin_run`, which receives one argument and is the entry point for the plug-in's functionality. Analysis plug-ins rely on the interface provided by the tool to register their results (see listing 3.2). An object with such interface is the argument passed by the main tool to the plug-in, via the argument of the `plugin_run` function. From that point on, the plug-in is free to perform its analysis and register its results as it sees fit. This includes, for instance, spawning a subprocess to run another program.

Using plug-ins to spawn subprocesses is an useful practice, as it allows interoperability with existing tools, or other frameworks, implemented in different programming languages. This reinforces the idea that plug-ins are, ideally, a bridge between existing tools and this tool, converting analysis results into non-compliance occurrences. Suppose that there is a Python analysis tool called *py_verifier*. Listing 3.6 presents the general structure of a plug-in that would reuse this tool to extract results.

```python
import os
import subprocess

# This function is the entry point for the plug-in.
def plugin_run(api):
    # Retrieve every known Python file, by file name extension.
    # Each file is represented as a tuple, consisting of
    # (id, file name, relative path, package id)
    files = api.getFileInfo(ext="py")
    for f in files:
        # Calculate the absolute file path of the file,
        # using the relative path and the file name.
        file_path = api.getPath(f[2], file_name = f[1])
        # Spawn a subprocess and collect its results.
        results = run_py_verifier(file_path)
        for r in results:
            # Register each non-compliance occurrence found,
            # using as much information as possible, even though
            # only the rule id and package id are mandatory.
            api.writeNonCompliance(r["rule id"], f[3],
                    file_id=f[0], line=r["line number"],
                    function=r["function name"],
                    comment=r["informative comment"])

def run_py_verifier(file_path):
    # Call a subprocess with the tool, and convert its results.
    # ...
```

Listing 3.6: Example structure of a plug-in, using a made up tool called *py_verifier*.

This example plug-in starts by using the provided interface (the `api` argument) to retrieve all Python files known to the database. Note that these files belong only to the packages included in the filter file defined before. This action is seen in the first line of code, by using the `getFileInfo` function. This function returns a list of all files, representing each file as a tuple with the file identifier, the file name, the relative file path (package and sub-directories) and the identifier of the package the file belongs to. With the list of target files at hand, the plug-in proceeds to iterate over each file. For each of the files, it uses the interface again to extract the absolute file path in the file system (the `get-Path` function), and then spawns a subprocess using the made up *py_verifier* tool. Since it is a made up tool, the example abstracts the conversion of the tool's results to a format usable by the plug-in. Finally, for each of the analysis results, the plug-in uses the `writeNonCompliance` function to register a rule violation. In order to register a rule violation, only the rule and package identifiers are mandatory. However, the plug-in tries to make the non-compliance occurrence as detailed as possible, registering also the file identifier, line number, function name, and additional details, if available.

3.2 CASE STUDY AND EVALUATION

At this point, with a basic implementation of the analysis tool, the plug-in system, and the visualisation tool, we are all set to take an existing ROS application as a case study of the tool. However, as per the previous discussion up to this point, we know that the tool does not perform analysis by its own, it needs analysis plug-ins. Thus, prior to delving into the case study itself, we work through the integration of existing tools into plug-ins for this tool. Such tasks serve as an extended test of the tool's capabilities, its extensibility, and its performance when handling hundreds, or even thousands of rule violations.

For this case study, we work with Cpplint and Cppcheck, both free C++ static analysis tools in active development, and with Kobuki, as the chosen ROS application for its popularity. The former tool, developed by Google, checks violations against Google's C++ coding standard, and coding style issues. In fact, at its current version, the tool has a strong focus on coding style, while its ability to find programming issues is still limited. In contrast, the latter is not bound to any coding standard, and focuses on programming errors often undetected by compilers, such as memory leaks, use of redundant code, or use of deprecated functions. Kobuki, as mentioned in section 2.4, is a low-cost mobile robot base, implemented using ROS, and designed with education and research in robotics in mind. Subsection 3.2.1 covers the required first step of implementing the plug-ins, while subsection 3.2.2 overviews the analysis and its results.

3.2.1 *Cpplint and Cppcheck Plug-ins*

The choice of Cpplint for this case study came quite naturally, given the reasons to back it up. First, and foremost, it is a free and open source tool for C++ code. C++ is one of the main development languages in ROS applications, and the fact that this is an open source tool makes adaptation and extensibility easier. Besides, Cpplint is implemented in Python, the same programming language used in the analysis tool developed for this project, and the language in which plug-ins are developed. This enhances interoperability between both tools. Regarding the coding standards, Google's C++ coding standard is a well known standard, already adopted in existing projects, and it is also the basis for the ROS C++ coding standard, which is one of the most important coding standards to consider in this project, given our focus on ROS software. Finally, supporting the tool's importance, one of the official ROS packages, *roslint*, provides a wrapper for Cpplint, slightly modified to mind a handful of differences between the two coding standards. Changing Cpplint into a plug-in for our analysis tool, while also incorporating the changes made by the ROS team, we can cover some rule violations from two coding standards with the same plug-in.

The reasons to back up the choice of Cppcheck as a second plug-in are similar. It is a free tool to analyse C++ code, it is under active development, and it is a widely used tool, with achieved results in large software projects. Its emphasis on finding programming errors also works well as a complement to Cpplint, allowing this couple of tools to have minimum overlap in their capabilities as plug-ins. Since Google's coding standard focuses more on style, finding programming errors and suspicious patterns allows the analysis to cover other stricter standards, such as MISRA C++, HIC++ or JSF-AV C++.

When adapting an existing tool to a plug-in, for this tool, there are essentially two options: either the plug-in is a thin layer that creates a child process to run the existing tool and then processes the output, or the tool's source code is adapted or imported to fully integrate it in the plug-in. This case study explores both routes. Since Cpplint is already implemented in Python, we opt for full integration in the plug-in. This means that Cpplint's source code must be adapted to change its entry point, as it is no longer a stand-alone tool, and it must change its error reporting, from printing to the screen to registering a non-compliance occurrence, using the plug-in interface. And these are, in essence, the only amendments needed in the tool's source code. Cppcheck's plug-in, on the other hand, is a thin layer over a child process that runs the Cppcheck tool. The plug-in does little more than running the tool, redirecting its output, and then processing said output and convert it to rule violations.

Cpplint works by parsing user arguments (since it is a command-line tool), loading configuration files, and then analysing the source code files. Contrary to some static analysis tools, it does not build a model of the program, or an abstract syntax tree. Instead, it analyses files line by line, sometimes looking ahead or looking back a few lines, in order to verify some rules that apply to structures that may span across multiple lines of code. It keeps and updates some objects representing various parsing states, so that it knows, for instance, if a certain line is inside a function or a class definition. It also keeps some additional application state, such as the tool's verbosity level, and output format.

The first step towards adapting Cpplint to a plug-in is discarding all code related to parsing user arguments and configuration files. These provide no utility here, since the plug-in interface covers everything these aspects provide, such as source file lists and locations. Then, its main function, its entry point, is slightly modified to comply with the structure required by the plug-in interface (see section 3.1.3 for a guide on writing plug-ins). Finally, the last modification required is also the most laborious. Cpplint defines an error function that is called whenever it encounters a violation during its verification process. This function writes formatted messages to the standard output, based on its arguments that act as error details. The function has been modified, so that it now receives the rule identifiers associated with the detected violations (Cpplint has no explicit concept of coding rule). As a consequence, every call to this function – about 130 calls, in the current version – also had to be modified, to include the specific rule that was violated. Additionally, most of these rules had to be inserted in the rules file, described in previous sections, since they are specific to this standard and tool, and, thus, were not covered yet.

The plug-in to leverage Cppcheck's capabilities is not as laborious, at least for basic functionality. In fact, its source code follows the same structure as the draft presented in listing 3.6, in the user guide (section 3.1.3). It starts by collecting a list of the packages of interest and then, for each package, calls Cppcheck as a subprocess. Cppcheck allows users to extend its rule set by defining an additional

XML file with extra rules to check, but these rules are limited to regular expressions. This plug-in makes use of this feature, in order to cover some simple rules of interest. After analysis, Cppcheck produces XML output with its error and warning reports, for all analysed files in a package, which is later parsed by the plug-in. Finally, the plug-in maps Cppcheck reports to specific rule violations, and uses the plug-in API to register the occurrences. The current version is able to detect violations of only 24 rules, though, related to unused or uninitialized variables, and other simple code patterns, given its constraint to regular expressions for user-defined rules.

3.2.2 *Analysis of Kobuki*

One of the first steps in this project was to evaluate which type of ROS package the quality analysis should focus on. We focus mainly on applications, as these are more representative of an individual robot, in the sense that by validating the application that controls the robot, one is also validating the robot itself. Thus, with the new plug-ins set in place, the next logical step would be to test them with a relevant ROS application. As a case study, we cover a subset of Kobuki, the mobile robot base discussed previously, in section 2.4. Since Kobuki's source code is maintained as a set of ROS packages, using ROS libraries and features, it is expected that a reasonable number of violations will be detected using the Cpplint plug-in. In particular, a number of code formatting issues should be raised, since Google and ROS differ in their coding style guidelines. The number of reported rule violations is also an estimate of the real number of rule violations, since Cpplint clearly states that some reports may be false positives. Cppcheck can also provide false positives, in particular when inconclusive analysis is enabled. For these tests, inconclusive analysis was not enabled. With this in mind, we now overview the analysis and the results gathered by the plug-ins.

In terms of coding rules, this analysis includes a total of 119 rules that both plug-ins should check. These rules cover most of section 5 and sections 4 and 11 of the ROS C++ Style Guide (*Formatting*, *License statements* and *Namespaces*, respectively). Given that the ROS C++ Style Guide is a work in progress, is based on Google's C++ Style Guide, and its tool, roslint, is mostly a clone of Cpplint, it is safe to assume that Google's rules apply whenever ROS has no explicit rule. As such, if we consider this extended view of the ROS C++ Style Guide, the rule coverage amounts to about 90 rules, given that the plug-in incorporates both Cpplint and the rules added by roslint. Looking at the guide, we can see that some of the rules that were left out are still related to syntax or text formatting, and could potentially be implemented with additional effort, such as naming conventions and the use of documentation throughout the code. Other rules require more specialised verification, beyond the scope or capabilities of these tools. These rules regard inheritance, global variables, and the use of exceptions and error codes.

This analysis is not limited to the ROS C++ Style Guide, however. In particular, when using the Cppcheck plug-in and Cppcheck's XML extensions, some rules from the HIC++ standard are also covered, little more than 20 in the current version. These rules concern, for instance, the use of deprecated features, such as keywords and libraries, or features that are prone to errors, such as *fall through* in *switch* cases. As mentioned in the discussion about HIC++, its most recent version references the MISRA and JSF AV C++ standards, and some of their rules overlap. Some of the implemented rules are part of this overlapping set, and thus a minuscule portion of these two standards (about 10 rules) is also covered in the analysis. Figure 11 puts these numbers side by side, ignoring overlaps (that is, counting them multiple times). Another way to look at rules is by category, instead of looking at them by standard. Close to half of the rules concern formatting issues. Code comments, preprocessing, functions and deprecation are the leading categories afterwards, having about 10 rules each. See Appendix A for the complete rule set file used in the analysis.

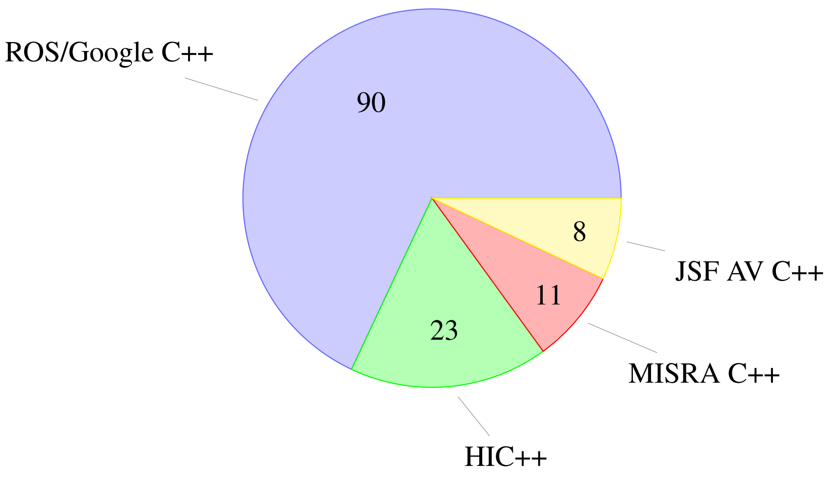


Figure 11.: Implemented rules by coding standard.

Our code base sample consists of 11 packages, three of which are *metapackages* – a package of packages, a logical aggregation of related packages into a single unit – and thus contain no C++ source code. This group of packages represents the source code necessary for Kobuki's drivers, keyboard remote operation, safety controllers, and a *random walker* application that makes the robot walk around, randomly changing its direction from time to time. It is a minimal sample of interest, when considering real, complex applications. Still, this sample contains 68 C++ source code files, amounting to more than 10 000 lines of code, and 2 913 rule violations, with the majority of these violations reported by Cpplint. Figure 12 depicts the distribution of rule violations per analysed package, clearly showing that Kobuki's driver, the package that contains most low-level code, is also the most non-compliant package. Additionally, table 6 shows the main categories of violated rules detected by the plug-ins. Figure 7, shown before, in section 3.1.2, depicts these results, as seen in the graphic component of the tool. Overall, the obtained results clearly show the emphasis of the Cpplint plug-in on coding-style, given that most rule violations fall under code formatting issues.

The *Formatting* column of table 6 refers both to code formatting issues, and issues related to license statements. The second column, *Function Parameters*, includes violations related only to function

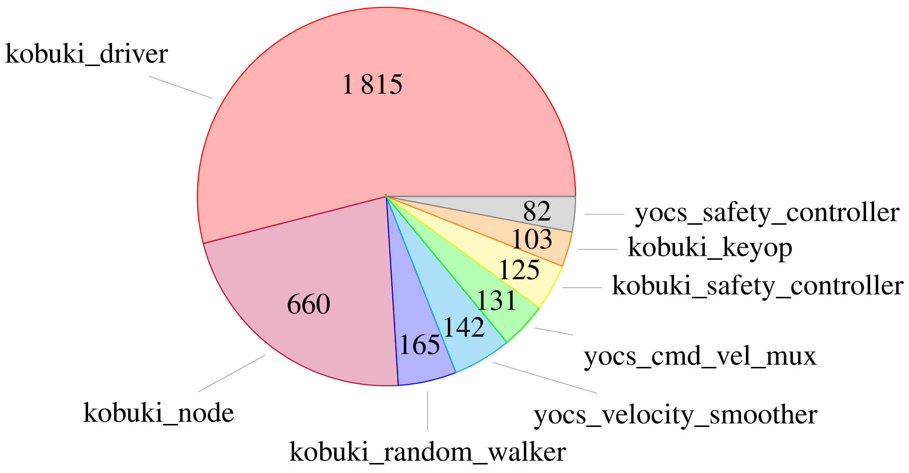


Figure 12.: Non-compliance results, per package, detected using the plug-ins.

Package	Formatting	Function Parameters	Possible Errors	Deprecated Features
kobuki_driver	1367	66	223	43
kobuki_node	574	18	33	8
kobuki_random_walker	146	0	6	2
yocs_velocity_smoother	118	2	13	2
kobuki_safety_controller	113	2	6	0
yocs_cmd_vel_mux	96	1	18	0
kobuki_keyop	94	0	4	0
yocs_safety_controller	66	2	6	0

Table 6.: Main categories of non-compliance detected by the plug-ins.

parameters, such as declaring too many parameters, or declaring parameters as mutable, when they could be declared as constant. The final column, *Deprecated Features*, is self-explanatory. It refers to the use of features that are no longer supported or recommended. The third column, *Possible Errors*, is composed of mostly every other violation that does not fall in the previous columns. Examples of violations that fall under this column include unused variables, uninitialized class members, code patterns that could be misinterpreted, or the use of features that may not be portable across different environments. While, in general, these rule violations may not represent threatening issues, in terms of software safety, they may have some impact in collaborative development. For instance, a contributor to the source code might misread badly formatted code, or have difficulty reading and using a function with too many parameters, especially when multiple parameters are of the same type. With that said, the formatting rules with most violations concern the maximum line length and the placement of curly braces (placing them on their own line versus placing them on the same line as the previous statement).

Table 7 shows yet another perspective on the analysis results. It contrasts the number of lines of code, per package, with the number of rule violations detected. With these numbers it is possible to calculate a ratio of rule violations per line of code, which helps visualise the dimension of non-compliant source code. On average, this ratio is about 0.28, that is, it is about one rule violation every four lines of code. While this ratio may be acceptable, especially considering the existence of false positives and knowing that most of these violations concern formatting, it is still a considerable number.

Package	Lines of Code	Rule Violations	Violations per Line of Code
kobuki_driver	5500	1815	0.33
kobuki_node	1940	660	0.34
kobuki_random_walker	530	165	0.31
yocs_velocity_smoother	430	142	0.33
kobuki_safety_controller	460	125	0.27
yocs_cmd_vel_mux	510	131	0.25
kobuki_keyop	680	103	0.15
yocs_safety_controller	310	82	0.26

Table 7.: Number of lines of code and rule violations per package.

3.3 SUMMARY

Having a look at the analysis tools studied in the previous chapter, there are two impediments to the adoption of an existing tool: free tools for C++ static analysis are rather limited *out-of-the-box*, while commercial tools, although very capable in terms of analysis, are not accessible to the general community and are not specifically suited for a ROS environment. Thus, the development of a new generic tool ensued, taking place as one of the main contributions of the project. In short, it is a tool for automatic source code fetching, analysis and quality measurements, and interactive visualisation of the analysis results using graphic models, such as diagrams.

The possibility of implementing this tool as a set of plug-ins for the SonarQube platform was considered and put aside, given the time frame of the project, the requirements of the ROS environment and the existence of RosEco, an open source tool that was in line with the main objectives pursued here, although to a limited extent. RosEco does not cover static analysis, and its analysis results and visualisation of them are not extensive enough. Nevertheless, it provided a solid foundation for the development of a new tool, and a much quicker start, when compared to developing a tool from scratch.

This new tool kept many ideas from RosEco, namely in terms of architecture. It is divided in two almost independent tools, or components, one responsible for source code management and analysis, while the other is responsible for the interactive reports and visualisation. The analysis component by itself does not perform any analysis, because different users might be interested in different rules and quality standards. As such, the tool provides an extensible interface for plug-ins, and delegates the analysis responsibility to the plug-ins. This chapter also provides instructions to use and extend the tool, with a description of its configuration parameters and a brief overview of its plug-in interface.

Finally, the chapter wraps up with a case study of the developed tool. We go through the adaptation process of Cpplint and Cppcheck, two free C++ analysis tools, into plug-ins for this tool. While the former tool is adapted by modifying its source code, the latter is used in its binary form, showing two alternatives for reusing existing tools. This case study applies the plug-ins to Kobuki, a popular robot using ROS, and we then review the analysis results. Even though these plug-ins have plenty of room for improvement and completeness, they were able to pick up about 3 000 rule violations, in a code base of more than 10 000 lines of code. Most rule violations concern formatting issues, and some of the violations are false positives, but there are also legitimate results concerning code quality. All in all, this shows the potential of this tool, and how straightforward it is to integrate existing tools.

4 CONCLUSIONS AND FUTURE WORK

The field of Robotics is certainly a center of much attention and effort, and more so as it becomes available in education, industry, research and for the general public. This is not without merit, as robots can incorporate ever larger and more complex software controllers, allowing them, in turn, to perform more complex tasks. One of the impediments to this rapid growth was that the software was often proprietary, closing doors to potential new researchers, but that is no longer the case. Open source systems, such as the Robot Operating System, emerged and grew sizeable and active communities, effectively pushing the limits of robotics. As a direct consequence, the safety and reliability of the software becomes a concern, and assuring high quality robotics software becomes a necessity, instead of a benefit. And, as it turns out, the field of robotics is still lagging behind, in terms of integrating high quality assurance techniques in its development process. This research project contributes to this integration, with a focus on ROS systems and the application of coding standards. In this final chapter we go through some concluding remarks, but, since the project cannot possibly cover this topic in full extent, there are also open possibilities of improvement and future work.

In chapter 2, we were able to harness an understanding of the fundamentals of coding rules and standards, the ROS framework, as well as some of the most widely adopted coding standards in industry for the C++ language. Coding standards are regarded in the industry as a logical and necessary component of software development, more so when large teams are involved, or when there is little margin for software errors. Verifying compliance with coding standards, however, can be a daunting task, if performed manually by an individual, given the amount and complexity of the rules a standard can have. Besides, many small details could pass unnoticed by the human eye. Thus, it is often the case that compliance verification is automated and delegated to tools. In the particular case of C++, the most capable tools are commercial. The free tools available often do not commit to specific standards, but instead rely on some general rules of thumb, and open up opportunities for extensibility, leaving most rules to be implemented by the users.

Regarding specific standards, the ROS C++ style guide is, without a doubt, one of the most important standards here considered, since the project's focus is on ROS software. However, it is mostly focused on stylistic issues, and very much based on Google's C++ style guide. These standards deliver more in terms of code readability and consistency across a project, than in terms of code quality, from a software engineering perspective.

Readability and syntactic consistency are complementary goals of attaining high reliability and correctness. Reliable and correct software is not necessarily made up of readable or consistent code, nor vice versa, but high quality software commonly exhibits both traits. Knowing that maintenance and software failures are often the overwhelming source of costs in the software industry, and knowing that both software and robots are increasingly applied in high reliability applications each day, such as medical devices and transportation, it stands to reason that software quality should not be disregarded when programming robots. This, then, presents a great research opportunity in the field of coding standards and code quality: to establish a set of conventions for ROS, one of the most used open source robotics frameworks, highly focused on the production and maintainability of reliable code.

The remaining considered standards, HIC++, MISRA C++, JSF AV C++, CERT C++ and JPL C, are more organized in their structure, more restrictive in their rules, and much more focused in delivering high quality software than the ROS and Google style guides. This is, in part, due to applying rules on language constructs and the way they are used, instead of defining stylistic guidelines. These quality centric coding standards are more appealing in the context of this project, as they are more in line with the goals we try to achieve. They are, in most aspects, independent of context, meaning that they do not encompass in any way the specifics of programming robots, much less when using a specific framework such as ROS, but most of their rules should be applicable regardless, given their broad scope, reasoning and general utility. Some of these standards even refer to each other, leading to a significant rule overlap between them. Given their high adoption and success in the industry, it is not implausible to think that these standards, or some of their rules, should also be a part of a set of guidelines when programming ROS applications, thus improving the current ROS coding standard.

As said before, our first contribution was to put into perspective some important coding standards for the C++ programming language, and for programming ROS applications. Still, as also pointed out before, there is a strong opportunity for future research work in this area, by defining a new document, a set of coding conventions for high quality ROS software, from scratch. From the knowledge gathered in this study, such a standard would incorporate three types of rules:

- style and formatting guidelines,
- general quality, safety and feature centric rules, and
- context specific rules.

The first of these types of rules would come essentially from the current ROS style guide, in an attempt to preserve as many conventions as possible, thus invalidating little to no existing code in this matter. These style guidelines would be complemented with quality and feature centric rules, based on HIC++, MISRA C++ and JSF AV C++. These rules are of general utility in programming, however, and so this opens up some room for the third category, specific rules related to the ROS framework and the best use of its libraries. Such rules could filter out functions that tend to be error prone, or highlight scenarios in which certain features of the framework should not be used, for instance. This, however, requires a deep understanding of the ROS framework and programming robot systems. Finally, in terms of document and rule structure, the resulting document should follow a format similar to those of HIC++, MISRA C++ and JSF AV C++, with rule categorization, the reasoning behind every rule, and examples and exceptions. However, the rules themselves should follow the format of CERT C++, with detailed levels of severity, risk and compliance. This way, the resulting document would take the best from each of the existing standards.

Moving on to the tool department, after identifying a certain lack of free verification tools for compliance with C++ coding standards, this project aimed to fill in that role, by developing an extensible and generic tool for quality assurance of ROS software. Recalling the previous discussion on SonarQube, it seemed to be a promising platform to explore and extend, and so it is plausible to conceive this tool as a SonarQube plug-in, or a series of plug-ins and utility tools. However, given the limited time frame of the project, the challenges of incorporating ROS characteristics in SonarQube and the existence of the RosEco project, starting an independent tool with basis on RosEco was deemed to be the more advantageous route.

This new tool, described in chapter 3 and available on GitHub[1], analyses ROS software following a compliance model, meaning that users define rules and the tool identifies rule violations. It is extensible through delegation of the actual analysis to a plug-in system, that users can use and extend themselves. It is generic in the sense that it should be able to accommodate as many and as diverse coding and quality standards as deemed logical, and its plug-in system presents a bridge between this tool and existing, reusable tools. While the tool itself is little more than a database manager and a web application that renders its data, there is much potential in its plug-in system, given that its plug-ins have few restrictions imposed.

It is a fact that this tool is yet in a very early stage, and so there are many ways to improve it. Some of these are:

- improving the tool's performance and scalability to larger systems, by optimising the use of its database connections;
- improving the tool's interface to plug-ins, by extending its capabilities, thus making plug-ins easier to write, and more versatile;
- improving the tool's user interface, in the visualisation component, so that it handles large amounts of data more efficiently;
- improving the tool's user interface, by rendering similar diagrams for the files inside a package;
- extending the plug-in system so that plug-ins may be integrated in other stages of execution, besides the analysis stage;

1 https://github.com/git-afsantos/haros

- improving the analysis system so that it detects whether two plug-ins disagree about a rule violation (that is, one detects a violation while the other does not detect a violation on the same rule);
- extending the analysis capabilities to identify ROS nodes and messages, possibly rendering them in a diagram.

By further developing this tool, and equipping it with a set of plug-ins that harness existing free tools and cover a relevant part of reputable coding and quality standards, this tool appears to be a relevant and needed contribution to the ROS community. This community already has some efforts in developing a static analysis tool, *roslint*, which focuses mainly on style issues. Roslint can be fully integrated in this tool as a plug-in, as was demonstrated in the case study, keeping a sense of *backwards compatibility* and further increasing the potential value of this contribution.

Summing up, this project delivers an overview of the ROS framework and some of the most relevant coding standards for the C++ language, one of the main programming languages in ROS. It also delivers a summary of the available tools to automate compliance verification, showing some lack of free and open source tools. This served as a motivation for the remainder of the project and its contributions, which delve into building such a tool and leveraging its capabilities with a case study. Finally, it presents opportunities for future work, by defining a quality centric coding standard for ROS, given that the community is currently lacking one, and by refining the implemented tool. As the software industry tends to agree, and as this project hopes to show, coding standards are a crucial part in building high quality software, and there is still plenty of room for improvement regarding robotics, and ROS in particular.

BIBLIOGRAPHY

Scott W Ambler. Writing Robust Java Code. *The AmbySoft Inc Coding Standards for Java v17.01d*, January 2000.

Paul Anderson. Coding standards for high-confidence embedded systems. In *2008 IEEE Military Communications Conference (MILCOM 2008)*, 2008. ISBN 9781424426768.

Wojciech Basalaj. HICPP, JSF++ and MISRA C++: a study of rule overlaps and effective compliance. 2011.

Wojciech Basalaj and Richard Corden. High Integrity C++ Coding Standard V4.0 - an overview. 2013.

David W Binkley. C++ in safety critical systems. *Annals of Software Engineering*, 4(1):223–234, 1997.

Joshua Bloch. *Effective Java*. 2008. ISBN 9780321356680. doi: 10.1016/B978-075067929-9/50038-5.

Andrea Capiluppi, Cornelia Boldyreff, Karl Beecher, and Paul J. Adams. Quality Factors and Coding Standards - a Comparison Between Open Source Forges. *Electronic Notes in Theoretical Computer Science*, 233:89–103, 2009.

CERT Division of the Software Engineering Institute - Carnegie Mellon University. CERT C++ Coding Standard. `https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637`. [Online; accessed 04-January-2015].

Richard Corden. Freeing C++ Developers with a Coding Standard. In *using std::cpp*, Madrid, Spain, 2013. URL `http://www.programmingresearch.com/resources/seminars/4021-2/`.

Richard Corden. V4.0 High Integrity C++ Coding Standard (HIC++), One Year On. 2014.

Ayssam Elkady and Tarek Sobh. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012, 2012. doi: 10.1155/2012/959013.

Andre Goforth. The Role and Impact of Software Coding Standards On System Integrity. In *AIAA Infotech@Aerospace (I@A) Conference*, Guidance, Navigation, and Control and Co-located Conferences. American Institute of Aeronautics and Astronautics, August 2013. doi: 10.2514/6.2013-5222. URL `http://dx.doi.org/10.2514/6.2013-5222`.

Google. Google C++ Style Guide. http://google-styleguide.googlecode.com/svn/trunk/cppguide.html, 2014. [Online; accessed 04-January-2015].

Blake Hannaford, Jacob Rosen, Diana W. Friedman, Hawkeye King, Phillip Roan, Lei Cheng, Daniel Glozman, Ji Ma, Sina Nia Kosari, and Lee White. Raven-II: An open platform for surgical robotics research. *IEEE Transactions on Biomedical Engineering*, 60:954–959, 2013. ISSN 00189294. doi: 10.1109/TBME.2012.2228858.

Les Hatton. Safer language subsets: An overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.

Les Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49(5):475–482, 2007.

Gerard J Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6): 95–99, 2006.

Gerard J Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, 2014.

Pablo Iñigo Blasco, Fernando Diaz-del Rio, M^{a} Carmen Romero-Ternero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, 60(6):803–821, 2012. ISSN 09218890. doi: 10.1016/j.robot.2012.02.004.

Daniel Jackson. Dependable Software by Design. *Scientific American*, 294(6):68–75, 2006.

Jet Propulsion Laboratory. *JPL Institutional Coding Standard for the C Programming Language*, March 2009.

Ajith K. John, Babita Sharma, A. K. Bhattacharjee, S. D. Dhodapkar, and S. Ramesh. Detection of Runtime Errors in MISRA C Programs: A Deductive Approach. In *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'07, pages 491–504, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75100-9, 978-3-540-75100-7. URL http://dl.acm.org/citation.cfm?id=2392550.2392609.

Xiaosong Li and Christine Prasad. Effectively teaching coding standards in programming. In *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, pages 239–244, New York, NY, USA, 2005. ACM. ISBN 1-59593-252-6. doi: 10.1145/1095714.1095770. URL http://doi.acm.org/10.1145/1095714.1095770.

Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle C++ Coding Standard for the System Development and Demonstration Program*, 2005.

Bruce MacDonald, David Yuen, Sylvia Wong, Evan Woo, Rowan Gronlund, Toby Collett, Félix-Étienne Trépanier, and Geoff Biggs. Robot programming environments. In *ENZCon'03: Proceedings of the Tenth Electronics New Zealand Conference : 1-2 September, 2003, University of Waikato, Hamilton, New Zealand*, University of Waikato, Hamilton, 2003.

MISRA. *Development Guidelines for Vehicle Based Software*. MIRA Limited, November 1994. ISBN 0952415607.

MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, 1998. ISBN 9780952415695. URL `https://books.google.pt/books?id=XomKAQAACAAJ`.

MISRA. *MISRA-C:2004: Guidelines for the Use of the C Language in Critical Systems*. MIRA, 2004. ISBN 9780952415626. URL `https://books.google.pt/books?id=j6oXAAAACAAJ`.

MISRA. *MISRA-C++:2008: Guidelines for the Use of the C++ Language in Critical Systems*. MIRA Limited, 2008. ISBN 9781906400040. URL `http://books.google.pt/books?id=q5p6MwEACAAJ`.

Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *2008 IEEE Conference on Robotics, Automation, and Mechatronics*, pages 736–742, 2008. ISBN 9781424416769. doi: 10.1109/RAMECH.2008.4681485.

James W Moore and Robert C Seacord. Secure Coding Standards. *CrossTalk*, (March):9–12, 2007.

Molaletsa Namoshe, N. S. Tlale, C. M. Kumile, and G. Bright. Open middleware for robotics. In *2008 15th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 189–194, December 2008. ISBN 9781424437795. doi: 10.1109/MMVIP.2008.4749531.

Günter Obiltschnig. C++ for Safety-Critical Systems. URL `http://www.appinf.com/download/SafetyCriticalC++.pdf`.

Takao Okubo and Hidehiko Tanaka. Secure software development through coding conventions and frameworks. In *2007 2nd International Conference on Availability, Reliability and Security (ARES)*, pages 1042–1051. IEEE, 2007.

Open Source Robotics Foundation. ROS C++ Style Guide. `http://wiki.ros.org/CppStyleGuide`, 2014a. [Online; accessed 04-January-2015].

Open Source Robotics Foundation. ROS Developer's Guide. `http://wiki.ros.org/DevelopersGuide`, 2014b. [Online; accessed 04-January-2015].

Programming Research Ltd. *High-Integrity C++ Coding Standard Manual*, 2013.

Programming Research Ltd. MISRA C and MISRA C++ Compliance. http://www.programmingresearch.com/solutions/coding-standards/misra/, 2014. [Online; accessed 04-January-2015].

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. In *ICRA Workshop on Open Source Software*, 2009. URL https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf.

Jeremy Rifkin and Ellen Kruger. *The end of work*. Social Planning Council of Winnipeg, 1996.

Robert C. Seacord. *Secure Coding in C and C++*. Pearson Education, 2005. ISBN 9780768685138. URL https://books.google.com/books?id=jfn1IAN3dvwC.

Robert C. Seacord. Secure coding standards. In *Proceedings of the Static Analysis Summit*, volume 500-262 of *NIST Special Publication*, pages 14–16, 2006. URL http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf.

William D. Smart. Is a common middleware for robotics possible? *Proceedings of the IROS 2007 workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*, 2007. URL http://www.cs.wustl.edu/~wds/library/papers/2007/iros-ws2007.pdf.

Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010283.

Maj Stenmark, Jacek Malec, and Andreas Stolt. From High-Level Task Descriptions to Executable Robot Code. In *Intelligent Systems'2014*, volume 323 of *Advances in Intelligent Systems and Computing*, pages 189–202. Springer International Publishing, 2015. ISBN 978-3-319-11309-8. doi: 10.1007/978-3-319-11310-4_17. URL http://dx.doi.org/10.1007/978-3-319-11310-4_17.

Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2421–2427. IEEE, October 2003. ISBN 0-7803-7860-1. doi: 10.1109/IROS.2003.1249233.

Didier Verna. Towards LaTeX coding standards. In *TUGboat*, volume 32, pages 309–328, 2011. URL http://tug.org/TUGboat/tb32-3/tb102verna.pdf.

A

ANALYSIS RULES

The following listing shows the rules file used during analysis, as mentioned in section 3.2.2. These rule declarations use the YAML syntax, as also mentioned before. Each rule has a description of what it is, and a set of tags for categorisation and filtering. Whenever a rule belongs to a specific standard, the standard is featured in its tags. The coding standard tags are assigned as follows:

ROS-CPP ROS C++ Style Guide

GOOGLE-CPP Google C++ Style Guide

HICPP High Integrity C++ Coding Standard

MISRA-CPP MISRA C++ Coding Standard

JSF-AV-CPP Joint Strike Fighter Air Vehicle C++ Coding Standard

```
%YAML 1.1
# Rules file.
---
-
    id:             1000
    name:           MAX_FUNCTION_PARAMETERS
    scope:          function
    description:    "Maximum number of function parameters: 6"
    tags:
        - code-standards
        - functions
        - parameters
        - hicpp
-
    id:             10000
    name:           MIXED_LINE_ENDINGS
    scope:          file
    description:    No file should mix LF and CRLF line endings.
    tags:
        - code-standards
        - formatting
        - whitespace
        - newline
-
    id:             10001
    name:           MAX_LINE_LENGTH
    scope:          file
    description:    No line should exceed 80 characters in length.
    tags:
        - code-standards
        - formatting
        - line-length
        - google-cpp
-
    id:             10002
    name:           MAX_LINE_LENGTH
    scope:          file
    description:    No line should exceed 120 characters in length.
    tags:
        - code-standards
        - formatting
        - line-length
        - ros
        - ros-cpp
-
    id:             10003
    name:           SPACES_OVER_TABS
    scope:          file
    description:    Indent each block by 2 spaces. Never insert literal tab
        characters.
    tags:
        - code-standards
        - formatting
        - whitespace
        - indentation
        - ros
        - ros-cpp
        - google-cpp
-
    id:             10004
    name:           OPEN_CURLY_BRACE
    scope:          file
    description:    The open curly brace should almost never be on its own line.
    tags:
        - code-standards
        - formatting
        - curly-braces
        - google-cpp
-
    id:             10005
    name:           OPEN_CURLY_BRACE
    scope:          file
    description:    The open curly brace is always on its own line.
    tags:
        - code-standards
        - formatting
        - curly-braces
        - ros
        - ros-cpp
-
    id:             10006
    name:           SHORT_FUNCTIONS
    scope:          function
    description:    If a function exceeds about 40 lines, think about whether it
        can be broken up without harming the structure of the program.
    tags:
        - code-standards
        - formatting
        - function-size
        - google-cpp
-
    id:             10007
    name:           BAD_CHARACTERS
    scope:          file
    description:    Non-ASCII characters should be rare, and must use UTF-8
        formatting.
    tags:
        - code-standards
        - formatting
        - character-encoding
        - google-cpp
-
    id:             10008
    name:           NEWLINE_AT_END_OF_FILE
    scope:          file
    description:    Files should end with a newline character.
    tags:
        - code-standards
        - formatting
        - newline
        - end-of-file
        - google-cpp
        - ros
        - ros-cpp
```

```
-
    id:             10009
    name:           ALIGNED_CLOSING_CURLY_BRACE
    scope:          file
    description:    Closing brace should be aligned with the beginning of class
        or structure.
    tags:
        - code-standards
        - formatting
        - curly-braces
        - indentation
        - google-cpp
-
    id:             10010
    name:           INDENT_ACCESS_MODIFIERS
    scope:          class
    description:    Access modifiers should be indented by one space.
    tags:
        - code-standards
        - formatting
        - indentation
        - access-modifiers
        - google-cpp
-
    id:             10011
    name:           BLANK_LINES_IN_CODE_BLOCKS
    scope:          function
    description:    Leave no redundant blank lines in code blocks.
    tags:
        - code-standards
        - formatting
        - whitespace
        - functions
        - code-blocks
        - google-cpp
-
    id:             10012
    name:           WHITESPACE_BEFORE_BRACKETS
    scope:          function
    description:    You shouldn’t have spaces before your brackets, except maybe
        after ’delete []’ or ’return []() {};’
    tags:
        - code-standards
        - formatting
        - whitespace
        - brackets
        - google-cpp
```

```
-
    id:             10013
    name:           WHITESPACE_AROUND_COLON
    scope:          function
    description:    Colons in range-based for loops should be preceeded and
       followed by a whitespace.
    tags:
        - code-standards
        - formatting
        - whitespace
        - colon
        - for-loop
        - range-based-for
        - google-cpp
-
    id:             10014
    name:           WHITESPACE_AROUND_ASSIGNMENT
    scope:          function
    description:    Assignment operators should be preceeded and followed by
       whitespace.
    tags:
        - code-standards
        - formatting
        - whitespace
        - assignment
        - google-cpp
-
    id:             10015
    name:           WHITESPACE_AROUND_BINARY_OPERATOR
    scope:          function
    description:    Binary operators should be preceeded and followed by
       whitespace.
    tags:
        - code-standards
        - formatting
        - whitespace
        - binary-operator
        - google-cpp
-
    id:             10016
    name:           WHITESPACE_AROUND_UNARY_OPERATOR
    scope:          function
    description:    Unary operators should have no whitespace around them.
    tags:
        - code-standards
        - formatting
        - whitespace
        - unary-operator
        - google-cpp
-
    id:             10017
    name:           WHITESPACE_BEFORE_PARENTHESIS
    scope:          function
    description:    There should be a space before a ( when it is preceeded by an
         if, switch, for or while.
    tags:
        - code-standards
        - formatting
        - whitespace
        - parenthesis
        - google-cpp
-
    id:             10018
    name:           WHITESPACE_INSIDE_PARENTHESIS
    scope:          function
    description:    Whitespace inside parenthesis should be consistent, and
        should consist of either zero or one space.
    tags:
        - code-standards
        - formatting
        - whitespace
        - parenthesis
        - google-cpp
-
    id:             10019
    name:           WHITESPACE_AFTER_COMMA
    scope:          function
    description:    There should be a space after a comma.
    tags:
        - code-standards
        - formatting
        - whitespace
        - comma
        - google-cpp
-
    id:             10020
    name:           WHITESPACE_AFTER_SEMICOLON
    scope:          function
    description:    There should be a space after a semicolon, if it does not end
         the line.
    tags:
        - code-standards
        - formatting
        - whitespace
        - semicolon
        - google-cpp
-
    id:             10021
    name:           WHITESPACE_BEFORE_OPENING_BRACE
    scope:          file
    description:    Except after an opening parenthesis, or after another opening
         brace (in case of an initializer list, for instance), you should have
        spaces before your braces.
    tags:
        - code-standards
        - formatting
        - whitespace
        - curly-braces
        - google-cpp
-
    id:             10022
    name:           WHITESPACE_AFTER_CLOSING_BRACE
    scope:          file
    description:    There should be a space after a closing brace, if it does not
         end the line (for instance, '} else {').
    tags:
        - code-standards
        - formatting
        - whitespace
        - curly-braces
        - google-cpp
-
    id:             10023
    name:           WHITESPACE_BEFORE_SEMICOLON
    scope:          function
    description:    Don't leave spaces before a semicolon at the end of a line.
    tags:
        - code-standards
        - formatting
        - whitespace
        - semicolon
        - google-cpp
-
    id:             10024
    name:           BLANK_LINE_BEFORE_SECTION
    scope:          class
    description:    The line before an access modifier (public, private,
        protected) should be blank, unless it is the beginning of the class.
    tags:
        - code-standards
        - formatting
        - whitespace
        - access-modifiers
        - google-cpp
-
    id:             10025
    name:           ELSE_STATEMENT_ON_ITS_LINE
    scope:          function
    description:    The else statement should be on the same line as the
       preceeding closing brace.
    tags:
        - code-standards
        - formatting
        - whitespace
        - if-else
        - curly-braces
        - google-cpp
-
    id:             10026
    name:           ELSE_BRACES
    scope:          function
    description:    If an else has a brace on one side, it should have it on both
        .
    tags:
        - code-standards
        - formatting
        - if-else
        - curly-braces
        - google-cpp
-
    id:             10027
    name:           SINGLE_LINE_ELSE_IF_ELSE
    scope:          function
    description:    The else clause of an else-if clause should be on its own
       line.
    tags:
        - code-standards
        - formatting
        - if-else
        - google-cpp
-
    id:             10028
    name:           SINGLE_LINE_DO_WHILE
    scope:          function
    description:    do/while clauses should not be on a single line.
    tags:
        - code-standards
        - formatting
        - do-while
        - google-cpp
-
    id:             10029
    name:           SINGLE_STATEMENT_IF_ELSE
    scope:          function
    description:    There should not be more than one semicolon statement in a
        single-line if or else statement, unless it is part of a lambda expression
        , or the if/else body is enclosed in curly braces.
    tags:
        - code-standards
        - formatting
        - if-else
        - single-line-statements
        - google-cpp
-
    id:             10030
    name:           AMBIGUOUS_IF_ELSE_INDENTATION
    scope:          function
    description:    Statements that are not part of an if body should have an
        indentation level equal to or less than the if statement.
    tags:
        - code-standards
        - formatting
        - whitespace
        - indentation
        - if-else
        - ambiguous-indentation
        - google-cpp
-
    id:             10031
    name:           WHITESPACE_AT_END_OF_LINE
    scope:          function
    description:    Do not leave whitespace at the end of a line.
    tags:
        - code-standards
        - formatting
        - whitespace
        - google-cpp
-
    id:             10032
    name:           SPACE_INDENTATION
    scope:          file
    description:    Use 2 spaces for indentation.
    tags:
        - code-standards
        - formatting
        - whitespace
        - indentation
        - google-cpp
-
    id:             10033
    name:           ONE_COMMAND_PER_LINE
    scope:          function
    description:    Avoid more than one command per line.
    tags:
        - code-standards
        - formatting
        - google-cpp
-
    id:             10034
    name:           IF_ON_ITS_OWN_LINE
    scope:          function
    description:    An if should start on its own line. Avoid constructs such as
        '} if'.
    tags:
        - code-standards
        - formatting
        - if-else
        - google-cpp
-
    id:             10035
    name:           NAMESPACE_INDENTATION
    scope:          namespace
    description:    Do not indent within a namespace.
    tags:
        - code-standards
        - formatting
        - indentation
        - namespace
        - google-cpp
-
    id:             10036
    name:           CLOSE_CURLY_BRACE
    scope:          file
    description:    The closing curly brace is always on its own line.
    tags:
        - code-standards
        - formatting
        - curly-braces
        - ros
        - ros-cpp
-
    id:             10100
    name:            LICENSE_STATEMENT
    scope:           file
    description:     Every source and header file must contain a license and
       copyright statement at the beginning of the file.
    tags:
        - code-standards
        - comments
        - license
        - copyright
        - google-cpp
        - ros
-
    id:              10101
    name:            MULTILINE_COMMENTS_END
    scope:           file
    description:     Multi-line comments must have an end.
    tags:
        - code-standards
        - comments
        - multiline-comments
        - invalid-code
        - google-cpp
-
    id:              10102
    name:            COMPLEX_MULTILINE_COMMENTS_AND_STRINGS
    scope:           file
    description:     Strings and /**/-comments should not extend beyond one line.
    tags:
        - code-standards
        - comments
        - strings
        - multiline-comments
        - multiline-strings
        - google-cpp
-
    id:              10103
    name:            END_OF_NAMESPACE_COMMENT
    scope:           namespace
    description:     Namespaces should have a comment at the end.
    tags:
        - code-standards
        - formatting
        - comments
        - namespace
        - google-cpp
        - ros
        - ros-cpp
```

```
-
    id:             10104
    name:           WHITESPACE_AFTER_PARENTHESIS
    scope:          file
    description:    Except in if/for/while/switch, there should never be space
        immediately inside parens (eg "f( 3, 4 )"). We make an exception for
        nested parens ( (a+b) + c ).
    tags:
        - code-standards
        - formatting
        - whitespace
        - parenthesis
        - google-cpp
-
    id:             10105
    name:           WHITESPACE_BEFORE_PARENTHESIS
    scope:          file
    description:    There should never be a space before a ( when it is a
        function argument. Closing parenthesis should not be preceded only by
        whitespaces.
    tags:
        - code-standards
        - formatting
        - whitespace
        - parenthesis
        - google-cpp
-
    id:             10106
    name:           WHITESPACE_BEFORE_COMMENTS
    scope:          file
    description:    At least two spaces is best between code and comments.
    tags:
        - code-standards
        - formatting
        - whitespace
        - comments
        - google-cpp
-
    id:             10107
    name:           WHITESPACE_BEFORE_COMMENT_TEXT
    scope:          file
    description:    Use one space before comment text.
    tags:
        - code-standards
        - formatting
        - whitespace
        - comments
        - google-cpp
-
    id:             10108
    name:           TODO_COMMENT_FORMAT
    scope:          file
    description:    "To-do comments should look like '// TODO(my_username): Stuff
        .'."
    tags:
        - code-standards
        - formatting
        - comments
        - todo-comment
        - google-cpp
-
    id:             10109
    name:           INCLUDE_DIRECTORY_IN_HEADER
    scope:          file
    description:    Include the directory when naming header files.
    tags:
        - code-standards
        - naming
        - headers
        - include
        - google-cpp
-
    id:             10200
    name:           HEADER_GUARD
    scope:          file
    description:    All headers must be protected against multiple inclusion by #
        ifndef guards.
    tags:
        - code-standards
        - header
        - header-guard
        - preprocessor
        - google-cpp
        - ros
        - ros-cpp
-
    id:             10201
    name:           HEADER_GUARD_FORMAT
    scope:          file
    description:    The format of the symbol name should be <PROJECT>_<PATH>_<
        FILE>_H_.
    tags:
        - code-standards
        - header
        - header-guard
        - preprocessor
        - google-cpp
-
    id:             10202
    name:           HEADER_GUARD_FORMAT
    scope:          file
    description:    The format of the symbol name should be <PACKAGE>_<PATH>_<
       FILE>_H.
    tags:
        - code-standards
        - header
        - header-guard
        - preprocessor
        - ros
        - ros-cpp
-
    id:             10203
    name:           HEADER_GUARD_CLOSE
    scope:          file
    description:    The format of the guard's end should be "#endif  // <PROJECT>
       _<PATH>_<FILE>_H_" or "#endif  /* <PROJECT>_<PATH>_<FILE>_H_ */".
    tags:
        - code-standards
        - header
        - header-guard
        - preprocessor
        - google-cpp
-
    id:             10204
    name:           INCLUDE_OWN_HEADER
    scope:          file
    description:    In general, every .cc file should have an associated .h file.
    tags:
        - code-standards
        - header
        - include
        - preprocessor
        - google-cpp
-
    id:             10205
    name:           TEXT_AFTER_ENDIF
    scope:          file
    description:    Uncommented text after #endif is non-standard. Use a comment
       instead.
    tags:
        - code-standards
        - preprocessor
        - invalid-code
        - google-cpp
-
    id:             10206
    name:           INCLUDE_ORDER
    scope:          file
    description:    Include files in alphabetical order, and in the following
       order. 1. preferred location 2. c system files 3. cpp system files 4.
       deprecated location 5. other headers
    tags:
        - code-standards
        - include
        - preprocessor
        - formatting
        - google-cpp
-
    id:             10207
    name:           UNNAMED_PARAMETERS
    scope:          function
    description:    All parameters should be named in a function.
    tags:
        - code-standards
        - parameters
        - functions
        - naming
        - google-cpp
-
    id:             10208
    name:           INCLUDE_WHAT_YOU_USE
    scope:          file
    description:    Include all required headers for what you use.
    tags:
        - code-standards
        - include
        - headers
        - preprocessor
        - google-cpp
-
    id:             10209
    name:           UNAPPROVED_HEADERS
    scope:          file
    description:    Do not include unapproved C++11 headers.
    tags:
        - code-standards
        - include
        - headers
        - preprocessor
        - invalid-code
        - cpp11
        - google-cpp
-
    id:             10210
    name:           UNAPPROVED_CLASSES_AND_FUNCTIONS
    scope:          file
    description:    Do not use unapproved C++11 classes and functions.
    tags:
        - code-standards
        - class
        - functions
        - invalid-code
        - cpp11
        - google-cpp
-
    id:             10211
    name:           C_SYSTEM_HEADERS
    scope:          file
    description:    Do not include the C standard headers. Use the C++ headers
       instead.
    tags:
        - code-standards
        - include
        - headers
        - preprocessor
        - deprecation
        - c
        - hicpp
-
    id:             10212
    name:           DEPRECATED_FUNCTIONS
    scope:          file
    description:    Do not use deprecated STL library features.
    tags:
        - code-standards
        - deprecation
        - functions
        - hicpp
-
    id:             10213
    name:           REGISTER_KEYWORD
    scope:          file
    description:    Do not use the deprecated register keyword.
    tags:
        - code-standards
        - deprecation
        - register
        - cpp11
        - hicpp
-
    id:             10214
    name:           THROW_SPECIFICATION
    scope:          function
    description:    Do not use throw exception specifications. Use noexcept
       instead.
    tags:
        - code-standards
        - deprecation
        - cpp11
        - exceptions
        - throw
        - hicpp
-
    id:             10215
    name:           C_STANDARD_LIBRARY
    scope:          file
    description:    Wrap use of the C Standard Library.
    tags:
        - code-standards
        - c
        - libraries
        - c-standard-library
        - hicpp
        - misra-cpp
-
    id:             10300
    name:           INVALID_INCREMENT
    scope:          file
    description:    Do not use the invalid increment form *count++.
    tags:
        - code-standards
        - increment
        - pointer
        - invalid-code
        - google-cpp
-
    id:             10301
    name:           STORAGE_CLASS_BEFORE_TYPE
    scope:          file
    description:    Storage class should come before the type.
    tags:
        - code-standards
        - invalid-code
        - storage-class
        - google-cpp
-
    id:             10302
    name:           INVALID_FORWARD_DECLARATION
    scope:          file
    description:    Inner-style forward declarations are invalid.
    tags:
        - code-standards
        - invalid-code
        - forward-declaration
        - google-cpp
-
    id:             10303
    name:           DEPRECATED_OPERATORS
    scope:          file
    description:    Do not use non-standard or deprecated operators (e.g. >? and
        <?).
    tags:
        - code-standards
        - invalid-code
        - deprecation
        - operators
        - google-cpp
-
    id:             10304
    name:           EMPTY_STATEMENT
    scope:          function
    description:    Don't use a semicolon to denote an empty statement. Use {}
        instead.
    tags:
        - code-standards
        - formatting
        - empty-statement
        - semicolon
        - empty-block
        - google-cpp
-
    id:             10305
    name:           RVALUE_REFERENCE
    scope:          function
    description:    Do not use RValue references.
    tags:
        - code-standards
        - invalid-code
        - rvalue-reference
        - google-cpp
-
    id:             10306
    name:           POINTLESS_EMPTY_STATEMENT
    scope:          function
    description:    Do not use meaningless empty statements.
    tags:
        - code-standards
        - empty-statement
        - ambiguous-code
        - google-cpp
-
    id:             10307
    name:           CHECK_EQ_INSTEAD_OF_CHECK
    scope:          function
    description:    To check for equality, use CHECK_EQ(a, b) instead of CHECK(a
        == b).
    tags:
        - code-standards
        - macros
        - equality
        - google-cpp
-
    id:             10308
    name:           ALTERNATIVE_TOKENS
    scope:          function
    description:    Do not use alternative tokens instead of operators (e.g. ’and
        ’, ’or’).
    tags:
        - code-standards
        - tokens
        - alternative-tokens
        - operators
        - google-cpp
-
    id:             10309
    name:           INCLUDE_TWICE
    scope:          file
    description:    Do not include the same file twice.
    tags:
        - code-standards
        - include
        - headers
        - preprocessor
        - google-cpp
-
    id:             10310
    name:           INCLUDE_CPP_FILES
    scope:          file
    description:    Do not include non-header files from other packages.
    tags:
        - code-standards
        - include
        - preprocessor
        - implementation-files
        - google-cpp
-
    id:             10311
    name:           C_TYPES
    scope:          function
    description:    Do not use the verboten C basic types.
    tags:
        - code-standards
        - types
        - deprecation
        - google-cpp
-
    id:             10312
    name:           OPERATOR&_OVERLOAD
    scope:          file
    description:    Do not use the unary operator&.
    tags:
        - code-standards
        - operators
        - overload
        - google-cpp
-
    id:             10313
    name:           FORMAT_STRING_VARIABLES
    scope:          function
    description:    Avoid using variables as format string arguments. Use 'printf
        ("%s", var)' instead.
    tags:
        - code-standards
        - formatting
        - strings
        - potential-bugs
        - google-cpp
-
    id:             10314
    name:           NAMESPACE_USING_DIRECTIVES
    scope:          file
    description:    Do not use namespace using-directives. Use using-declarations
         instead.
    tags:
        - code-standards
        - namespace
        - using-directives
        - using-declarations
        - google-cpp
        - hicpp
        - misra-cpp
-
    id:             10315
    name:           VARIABLE_LENGTH_ARRAYS
    scope:          file
    description:    Do not use variable-length arrays.
    tags:
        - code-standards
        - arrays
        - variables
        - constants
        - google-cpp
-
    id:             10316
    name:           UNNAMED_NAMESPACES
    scope:          file
    description:    Do not use unnamed namespaces in header files.
    tags:
        - code-standards
        - namespace
        - unnamed-namespace
        - headers
        - google-cpp
-
    id:             10317
    name:           STRING_CONSTANTS
    scope:          file
    description:    Use C-style strings for static and global string constants.
    tags:
        - code-standards
        - strings
        - constants
        - google-cpp
-
    id:             10318
    name:           SNPRINTF_ARGUMENTS
    scope:          function
    description:    Avoid using literals as the second argument for snprintf.
    tags:
        - code-standards
        - literals
        - potential-bugs
        - google-cpp
-
    id:             10319
    name:           STRING_PRINT_C_FUNCTIONS
    scope:          function
    description:    Avoid string printing C functions (sprintf, strcpy, strcat).
       Use snprintf instead.
    tags:
        - code-standards
        - deprecation
        - library
        - strings
        - google-cpp
-
    id:             10320
    name:           NON_CONST_REFERENCE_PARAMETERS
    scope:          function
    description:    Avoid non-const reference parameters. Use const or pointers.
    tags:
        - code-standards
        - parameters
        - const
        - pointers
        - references
        - google-cpp
-
    id:             10321
    name:           DEPRECATED_CASTING
    scope:          function
    description:    Don't use deprecated casting styles.
    tags:
        - code-standards
        - casting
        - deprecation
        - google-cpp
-
    id:             10322
    name:           DANGEROUS_ADDRESSES
    scope:          function
    description:    Avoid using dangerous addresses, such as addresses from casts
       , or addresses dereferenced from casts.
    tags:
        - code-standards
        - potential-bugs
        - addresses
        - casting
        - google-cpp
-
    id:             10323
    name:           MAKE_PAIR_TEMPLATE
    scope:          file
    description:    Either omit template arguments from make_pair, or use pair
        directly, or construct a pair directly.
    tags:
        - code-standards
        - cpp11
        - compatibility
        - templates
        - pair
        - google-cpp
-
    id:             10324
    name:           DEFAULT_LAMBDA_CAPTURES
    scope:          function
    description:    Do not use default lambda captures.
    tags:
        - code-standards
        - invalid-code
        - lambda
        - default-lambda-capture
        - google-cpp
-
    id:             10325
    name:           REDUNDANT_VIRTUAL_DECLARATION
    scope:          function
    description:    Do not declare a function as both "virtual" and "override" or
         "final".
    tags:
        - code-standards
        - functions
        - virtual
        - override
        - final
        - rendundancy
        - google-cpp
-
    id:             10326
    name:           REDUNDANT_OVERRIDE_DECLARATION
    scope:          function
    description:    Do not declare a function as both "override" and "final".
    tags:
        - code-standards
        - functions
        - override
        - final
        - rendundancy
        - google-cpp
-
    id:             10400
    name:           INITIALIZE_MEMBER_VARIABLES
    scope:          class
    description:    All member variables of a class should be initialized after
        calling the constructor.
    tags:
        - code-standards
        - classes
        - member-variables
        - uninitialized-variables
        - constructors
        - google-cpp
        - jsf-av-cpp
        - misra-cpp
        - hicpp
-
    id:             10401
    name:           UNUSED_VARIABLES
    scope:          file
    description:    There shall be no unused variables.
    tags:
        - code-standards
        - variables
        - unused-variables
        - misra-cpp
-
    id:             10402
    name:           REDUNDANT_EXPRESSIONS
    scope:          function
    description:    Ensure that no expression is redundant.
    tags:
        - code-standards
        - redundancy
        - hicpp
-
    id:             10403
    name:           SMALLEST_SCOPE
    scope:          file
    description:    Declarations should be at the smallest feasible scope.
    tags:
        - code-standards
        - scope
        - jsf-av-cpp
-
    id:             10404
    name:           CASE_FALL_THROUGH
    scope:          function
    description:    Non-empty case blocks must not fall through to the next case.
    tags:
        - code-standards
        - switch
        - case
        - fall-through
        - hicpp
-
    id:             10405
    name:           MIN_TWO_CASES
    scope:          function
    description:    A switch should have at least two cases distinct from the
       default case.
    tags:
        - code-standards
        - switch
        - hicpp
        - misra-cpp
        - jsf-av-cpp
-
    id:             10406
    name:           ENUM_BASE
    scope:          file
    description:    Ensure that an enum has a specified base type able to
       accomodate all its values.
    tags:
        - code-standards
        - enum
        - types
        - hicpp
-
    id:             10407
    name:           ASM_DECLARATIONS
    scope:          file
    description:    Do not use asm declarations.
    tags:
        - code-standards
        - assembly
        - asm
        - hicpp
-
```

```
    id:             10408
    name:           MAX_POINTER_INDIRECTION
    scope:          file
    description:    Use at most one level of pointer indirection.
    tags:
        - code-standards
        - pointers
        - multiple-pointers
        - hicpp
-
    id:             10409
    name:           MAX_POINTER_INDIRECTION
    scope:          file
    description:    Use at most two levels of pointer indirection.
    tags:
        - code-standards
        - pointers
        - multiple-pointers
        - misra-cpp
        - jsf-av-cpp
-
    id:             10410
    name:           UNIQUE_POINTER_CONST_REFERENCE
    scope:          function
    description:    "Do not pass a std::unique_ptr by const reference."
    tags:
        - code-standards
        - pointers
        - std-unique-ptr
        - const
        - references
        - hicpp
-
    id:             10411
    name:           DEFAULT_ARGUMENTS
    scope:          function
    description:    Do not use default arguments.
    tags:
        - code-standards
        - functions
        - default-arguments
        - arguments
        - parameters
        - hicpp
-
    id:             10412
    name:           STD_VECTOR_BOOL
    scope:           file
    description:     "Do not use std::vector<bool>. It does not conform to the
        requirements of a container."
    tags:
        - code-standards
        - std-vector
        - boolean
        - hicpp
        - misra-cpp
-
    id:              10413
    name:            UNIONS
    scope:           file
    description:     Do not use unions. Use a safe polymorphic abstraction,
        instead.
    tags:
        - code-standards
        - unions
        - polymorphism
        - type-safety
        - hicpp
        - misra-cpp
        - jsf-av-cpp
-
    id:              10414
    name:            INTEGER_TYPES
    scope:           file
    description:     Do not use integer types directly. Use size-specific typedefs
        , for instance from <cstdint>.
    tags:
        - code-standards
        - integer-types
        - type-safety
        - portability
        - hicpp
        - misra-cpp
        - jsf-av-cpp
-
    id:              10415
    name:            ORDER_OF_EVALUATION
    scope:           file
    description:     Do not rely on the sequence of evaluation within an
        expression.
    tags:
        - code-standards
        - evaluation-order
        - expressions
        - hicpp
        - misra-cpp
        - jsf-av-cpp
-
    id:             10416
    name:           FLOAT_ACCURACY
    scope:          file
    description:    Do not write code that expects floating point calculations to
         yield exact results.
    tags:
        - code-standards
        - floats
        - floating-point
        - hicpp
-
    id:             10417
    name:           OVERLOAD_SPECIAL_OPERATORS
    scope:          function
    description:    Do not overload operators with special semantics, such as
        '&&', '||', ',' or '&'.
    tags:
        - code-standards
        - overload
        - operators
        - hicpp
        - misra-cpp
        - jsf-av-cpp
-
    id:             10418
    name:           STD_ARRAY_RVALUE
    scope:          function
    description:    "Do not create an rvalue reference of std::array."
    tags:
        - code-standards
        - std-array
        - rvalue-reference
        - hicpp
-
    id:             20000
    name:           THREAD_SAFE_FUNCTIONS
    scope:          file
    description:    Avoid using thread-unsafe functions, when thread-safe
        variants are available.
    tags:
        - code-standards
        - multi-threading
        - thread-safety
        - google-cpp
-
    id:             30000
    name:           LOGGING_LEVELS
    scope:          file
    description:    Use VLOG with a numeric argument.
    tags:
        - code-standards
        - logging
        - library
        - google-cpp
-
    id:             30001
    name:           DISALLOW_MACRO
    scope:          class
    description:    If copying and assignment are disabled with a macro such as
        DISALLOW_COPY_AND_ASSIGN, it should be at the end of the private section,
        and should be the last thing in the class.
    tags:
        - code-standards
        - macros
        - class
        - constructors
        - google-cpp
-
    id:             30002
    name:           STRING_FORMATTING
    scope:          file
    description:    Do not use deprecated or unconventional string formattings.
    tags:
        - code-standards
        - strings
        - string-formatting
        - deprecation
        - invalid-code
        - google-cpp
-
    id:             30003
    name:           INVALID_CHARACTER_ESCAPES
    scope:          file
    description:    Do not use invalid escape sequences.
    tags:
        - code-standards
        - strings
        - escape-sequences
        - invalid-code
        - google-cpp
```

```
-
    id:             30004
    name:           CONST_STRING_REFERENCES
    scope:          file
    description:    Do not use const string& members. Use pointers or simple
        constants instead.
    tags:
        - code-standards
        - strings
        - references
        - constants
        - google-cpp
-
    id:             30005
    name:           EXPLICIT_CONSTRUCTORS
    scope:          class
    description:    Zero-parameter constructors, single-parameter constructors
        and constructors callable with one argument should be marked explicit.
    tags:
        - code-standards
        - explicit
        - constructors
        - google-cpp
-
    id:             30006
    name:           NON_EXPLICIT_CONSTRUCTORS
    scope:          class
    description:    Constructors that require multiple arguments should not be
        marked explicit.
    tags:
        - code-standards
        - explicit
        - constructors
        - google-cpp
-
    id:             30007
    name:           DISALLOW_MACROS_IN_PRIVATE
    scope:          class
    description:    DISALLOW macros must be in the private section.
    tags:
        - code-standards
        - macros
        - class
        - private
        - google-cpp
```

Listing A.1: Rule set used in the case study.