Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Ângelo Gomes da Costa

Software Quality for the Robot Operating System

October 2015

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Ângelo Gomes da Costa

Software Quality for the Robot Operating System

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Prof. Manuel Alcino Pereira da Cunha
Nuno Filipe Moreira Macedo

October 2015

Acknowledgements

Primeiramente, gostaria de expressar a minha sincera gratidão aos meus supervisores, Professor Alcino Cunha e Nuno Macedo, pelo seu constante apoio e pesquisa relacionada com o projeto, pela sua disponibilidade, paciência e motivação dada. A sua orientação foi essencial e indispensável na realização da aplicação para este projeto, assim como no desenvolvimento do presente relatório.

Pretendo também agradecer aos meus colegas e membros do HASLab, embora o tempo e convivência não tenha sido muito, para mim foi o suficiente para conhecer novas pessoas, aprender, trabalhar, relaxar e debater sobre uma diversa variedade de temas.

Por último, mas não menos importante, gostaria de agradecer aos meus pais, irmão e a Natália Costa pela constante presença, motivação e apoio dado, mesmo percebendo pouco do assunto.

Abstract

The huge development in the fields of robotics spurred a greater interaction between human beings and robots. It is common knowledge that the malfunctioning of robots may cause not only material damage, but also physical damages to its users. Hence, it is necessary to ensure that robots have proper operation, and one way to ensure this is through the use of high quality software.

In software engineering context there is the ability to assess the quality of the developed software. This quality will differ depending on whether the code meets all the requirements and specifications proposed by a certain quality model. The code quality can be measured through the study of internal quality components, more particularly the Maintainability.

The methods used for measuring the code quality are complex and can include compliance with standards of different quality models, such as *ISO 9000*[1].

This research project has as main objective the assessment of the code quality of several robots, developed from ROS (Robot Operating System - Quigley et al. (2009)). The assessment of this kind of information, about the code quality, is achieved through the use of static analysis tools. These tools, in turn, are responsible for conducting an analysis to source code, measuring its internal quality metrics without having to execute it.

Keywords: Robot Operating System, Software Engineering, Software Quality, Code Quality, Internal and External Quality Metrics, Maintainability.

[1] http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm

Resumo

O grande desenvolvimento na área da robótica impulsionou uma maior interação entre Seres Humanos e robôs. É de conhecimento geral que o mau funcionamento dos robôs pode causar não só danos materiais como também danos físicos aos seus utilizadores. Desta forma, é necessário assegurar que os robôs tenham um funcionamento correto, e uma das formas de o garantir é através da utilização de software de alta qualidade.

Num contexto de engenharia de software existe a capacidade de avaliar a qualidade do software desenvolvido. Esta qualidade vai variar dependendo se o código atinge todos os requisitos e especificações propostos por um determinado modelo de qualidade. A qualidade do código pode ser medida através do estudo das componentes de qualidade interna, mais particularmente a Maintainability.

Os métodos utilizados para aferir a qualidade de código são complexos e podem incluir o cumprimento de normas de diferentes modelos de qualidade, tal como o *ISO 9000*[2].

Este trabalho de investigação tem como principal objetivo a aferição da qualidade de código de varios robôs, desenvolvidos a partir do ROS (Robot Operating System - Quigley et al. (2009)). A aferição deste tipo de informação, sobre a qualidade do código, é obtida através do uso de ferramentas de analise estática. Estas ferramentas, por sua vez, são então responsávies por realizar uma análise ao código, medindo as suas métricas de qualidade interna sem nunca o ter de executar.

Palavras-chave: *Robot Operating System*, Engenharia de Software, Qualidade de Software, Qualidade de código, Métricas de qualidade interna e externa, *Maintainability*.

[2] http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CBO Coupling Between Objects. 14, 19, 20, 36, 39

CC Cyclomatic Complexity. 14, 15, 18, 19, 23, 29, 36, 39, 58

COM Lines of Comment. 14, 23

COMR Comment Code Ratio. 14, 15, 36, 39

DIT Deepest Level in Inheritance. 14, 20, 36, 39

ESPC Estimated Static Path Count. 14, 18, 36

GCC GNU Compiler Collection. 36

HV Halstead Volume. 23

LOC Lines of Code. 14, 23, 58

MI Maintainability Index. 12, 23, 29

MNC Maximum Nesting Control. 14, 17, 18, 36, 38, 39

NEL Number of Executable Lines. 14, 16, 36

NFC Number of Function Calls. 14, 17, 36

NOC Number of Immediate Children. 14, 20, 36, 39

NOM Number of Methods Available. 14, 19, 20, 36, 39

WMC Weighted Methods per Class. 14, 20, 36, 39

1. Introduction

In the 20th century there was a great development of robots in all sectors, particularly in the industrial sector. Throughout the remainder century robots changed the structure of society and allowed better working conditions. In addition, the implementation of advanced robotics in the military sector and NASA modified the panorama of national defense and space exploration.

Software is becoming increasingly important when it comes to controlling the robots, rather than relying solely on hardware. Using software to control robots promotes flexibility, interaction and abstraction, but its use also has negative aspects because the software is much more difficult to control and maintain, in comparison to hardware.

Given this difficulty, it is necessary to apply development methods to promote the quality of the final product. The use of quality models is one of the possible methods to ensure the quality of the software.

Quality models are constituted by a set of rules and practices that must be satisfied while developing software. Each quality model may have a different set of rules and they vary depending on the thresholds and quality metrics used. These quality metrics are measured, independently, through the use of static analysis tools, and it can be also used to predict failures in the software (Danijel Radjenović, 2013). In a software engineering concept, the quality of a software product is gauged as good when it meets all the rules proposed from a quality model. As previously mentioned, quality models are used to prevent the existence of low quality software on robots, promoting in turn their safety. Thus, from this quality models it is possible to obtain information about the software quality. Quality itself is actually quite difficult to define because it requires measuring subjective characteristics and there is no way to rate it, so every person will see it differently, that is why there are many different thresholds for the same programming/coding language. There are also different quality models, each one created with its own purpose, in particularly the standard ISO/IEC 9126 (ISO/IEC, 2001) describes a software quality model that is divided into three quality types, such as external quality, internal quality and quality in use. This three quality types are composed by six characteristics that are also subdivided into twenty-seven sub-characteristics (Heitlager et al., 2007).

For this research project it was chosen to use techniques to statically analyse source code, so the main focus is on measuring the internal quality. The internal quality is focused on measuring the quality of the developed software, through the analysis of its internal attributes. Furthermore, it is known that the internal quality also affects the external quality. To gauge something about the quality of the source code is necessary to focus on a particular characteristic of the internal quality, the *maintainability*. In software engineering, the maintainability is defined by the simplicity and the ease how changes can be made in a software system (IEEE, 1990). Some companies have defined a formula called maintainability index which uses internal quality metric values as variables. This formula may vary from company to company depending on the product or the aspects that the company is focusing.

Considering the increase in the use of robots, it is necessary to ensure that the use of the same is safe. Software errors can be one of the reasons to the safety breaches, particularly in highly complex robots.

The lack of safety is obviously a problem because it can lead to a unsafe operational functioning or harmful behavior of a robot. Despite of all the positive aspects of the robot usage, their malfunction may lead to many negative events, such as: great economical losses, failure of a mission and in a worst case scenario it can put people's lives at risk (Barchanski, 2011). For instance, the industrial robots are large and in turn also quite powerful, which means they are capable of make severe injuries to the persons that interact with them if the software used does not provide the necessary safety.

As indicated by Barchanski (2011), the software of robot control "allows unprecedented complexity which goes beyond the ability of current engineering techniques for assuring acceptable risk. Most of the publications on safety have a form of recommendations on providing safe environment for robot operators, like the Occupational Safety and Health Administration regulations or the more recent NASA recommendations for space robots. This approach is effective when accidents are primarily caused by hardware components failures."The main role of the robot control software is the interaction of its components (Barchanski, 2011). For instance, if we think of a robot that simulates the walking of the human being we have the component "*Brain*" and components "*Legs*", where the "*Brain*" has to interact with the "*Legs*", in order to make them move. Accidents occur mainly in the component interaction rather on the components themselves. For instance, with the previous example, an accident could happen if the *Brain* component fail to send information to the *Legs* component, which would make the robot to stop. In a industrial scenario, where the robots are utilized in a production process, if a robot stops working prematurely it might neglect the whole production process. With this knowledge, many middlewares, to robotic systems, were developed with the objective of improving the interaction process between components.

Although many middleware have been proposed to robotics (Elkady and Sobh, 2012), the chosen study middleware was the Robot Operating System (ROS - Quigley et al. (2009)). The reason that led to the choice of ROS and none of the other was due to many aspects, such as: the big community, the huge number of available packages, a very permissive open license, the fact that robots with ROS software are already used in industry and many others that can be find the ROS webpage[1]. The fact that the packages are all public allows us to explore and study the static analysis techniques on source code. ROS is an open-source, collection of software structures for a robot. It provides the expected standard services from an operation system such as hardware abstraction, implementation of commonly-used functionality, low-level device control, package management and message-passing between processes. ROS provides libraries and tools for obtaining, running, writing and building code on multiple computers (Thomas, 2014), and it also proposes a quality model built from other institution's quality models (Kuehn, 2013).

1.1. Main Goals

This research project is focused on determining whether the existing quality models fit the robotic software, in particular ROS, and in the process, study and evaluate the code quality of the ROS packages. The process previously referred is separated into two distinct stages, in the first stage a measurement of a number of quality metrics is executed, through the use of static analysis tools, while in the second stage these measurements are used in order to obtain some information about the code quality. More specifically, the objectives are:

- Explore the quality models that best fit the ROS system;
- Explore quality metrics and the static analysis tools able to measure them;
- Given the conclusions drawn in the previous assignments, develop a system for calculating quality models that takes into account the particularities of ROS;
- Measuring the quality of a large number of ROS packages, allowing the validation of the developed tool and, at the same time, present the quality of the current state of the ROS corpus.

[1] http://www.ros.org/is-ros-for-me/

Given the identification of the objectives for this research project, Chapter 2 is aimed at presenting the state of the art of this thematic, whereas Chapter 3 presents the contribution of this research project, which includes a developed system for calculating quality models and a study conducted on several ROS packages, and finally, the Chapter 4 is intended for the conclusions drawn from the study realized in Chapter 3.

2. State of the Art

Code quality is something unavoidable if we want to study the quality of the software. A way to gauge this quality is through the use of automated tools (independent of human interaction) that are responsible for conducting a static analysis of the source code without even executing it. Right after completing the code static analysis it is then necessary to choose the quality model that is intended to use, in order to verify if the developed software have a *good* or *bad* quality. The quality is assigned in accordance with the selected quality model. For instance, the code quality is considered *good* when the values of the quality metrics are within the values proposed by each quality model. On other hand, the existence of violated rules, i.e. values of quality metrics that are outside of the proposed threshold values, influences the reduction of code quality.

This chapter is dedicated to some of the research and knowledge acquired about quality metrics, code quality, quality models, maintainability characteristic, ROS and the static analysis tools. In order to simplify reading, henceforward, quality metrics will be referred solely as metrics.

2.1. Code Quality

Quality itself is actually quite difficult to define because all the qualities are subjective and there is no way to rate it, so every person will see it differently, that is why there are so many different thresholds for the same programming/coding language. Some define quality as a "systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures" (NASA, 1989). Others, give multiple meanings to quality, for instance, Juran (1974) gives two meanings to the word quality, which are:

1. "Quality means those features of products which meet customer needs and thereby provide customer satisfaction."

2. "Quality means *freedom* from *deficiencies*"

One of the first definitions of quality was given by Shewart (1931): “There are two common aspects of quality: one of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality.”

There is also some people that see it in some different perspectives like Kitchenham and Pfleeger (1996), together with the teachings of Garvin (1988) identified five different perspectives on quality:[1]

1. The **transcendental** perspective - deals with the metaphysical aspect of quality. In this view of quality, it is “something toward which we strive as an ideal, but may never implement completely” (Kitchenham and Pfleeger, 1996).

2. The **user** perspective - is focused in the adequacy of the product in a certain context. The user view is more concrete, based in the product characteristics that meet the user’s needs, however the transcendental view is more aerial. (Kitchenham and Pfleeger, 1996)

3. The **manufacturing** perspective - represents quality as conformance to requirements.

4. The **product** perspective - the quality can be evaluated by measuring the inherent characteristics of the product.

5. The **value-based** perspective - acknowledge that different perspectives can have different degrees of importance, or value to the various stakeholders.

The perspective addressed in this research project is the manufacturing perspective, because the software quality is directly linked with the compliance of the quality model requirements. A further explanation about the quality model requirements is given on Section 2.1.3.

As stated before, there are different quality models each one created with its own purpose. Let’s focus, for instance, on a standard model like ISO/IEC 9126 (Quigley et al., 2009). This standard describes a software quality model that is divided into six characteristics:

- Functionality
- Reliability
- Usability

[1] Source: http://en.wikipedia.org/wiki/Software_quality

- Efficiency
- Maintainability
- Portability

These characteristics, as presented in Figure 2.1, are also subdivided into twenty-seven sub-characteristics (Heitlager et al., 2007).

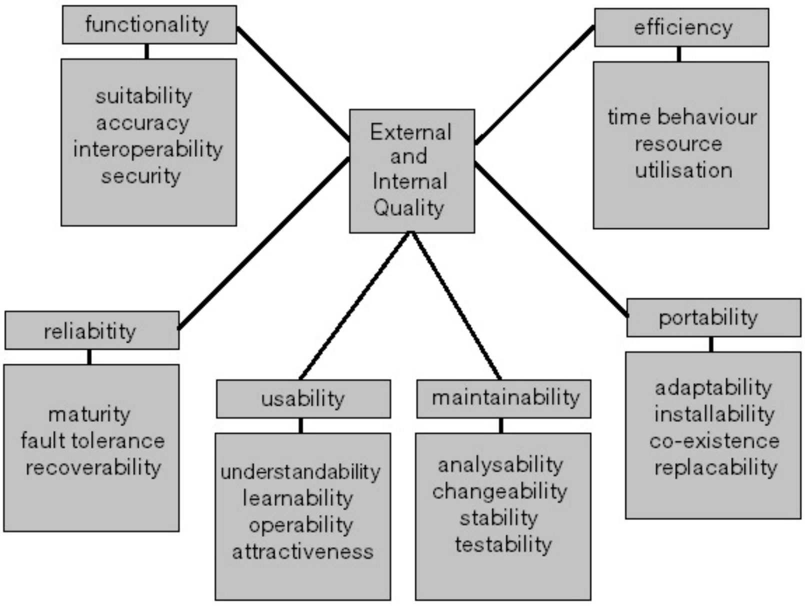


Figure 2.1.: The external and internal characteristics - ISO 9126-1 (ISO/IEC, 2001).

The standard ISO/IEC 9126 provides a framework that allows every organization to define his own quality model. With that possibility, each organization has the chance to specify accurately his own target values for quality metrics (Heitlager et al., 2007).

To get a better understanding of code quality it is convenient to distinguish the different quality types, such as:

- External quality,
- Internal quality,
- Quality in use.

The **Internal quality** concerns with the static properties of the code that can be measured without executing it. The **External quality** concerns with the behavior of the software when it is executed. The **Quality in use** according to Bevan and Azuma (1997) “is the user’s view of the quality of a system containing software, and is measured in terms of the result of using the software, rather than properties of the software itself. Quality in use is the combined effect of the software quality characteristics for the end user.”

As mentioned by Heitlager et al. (2007), it is believed that internal quality has an impact in the external quality, which in turn impacts with the quality in use (Figure 2.2). These three types of qualities are directly linked to code quality, but not all can be measured through the use of autonomous static analysis tools. Thus, the only type of quality addressed in this project is the **internal quality**, because it is easier to gauge and, as previously mentioned, the internal quality has an impact with the other two types of quality. To gauge something about the quality of the source code it is necessary to focus on a particular characteristic of the internal quality, the *maintainability* (Figure 2.3). The chosen characteristic is the maintainability because it can be measured by automatic tools without having the need to execute the code.

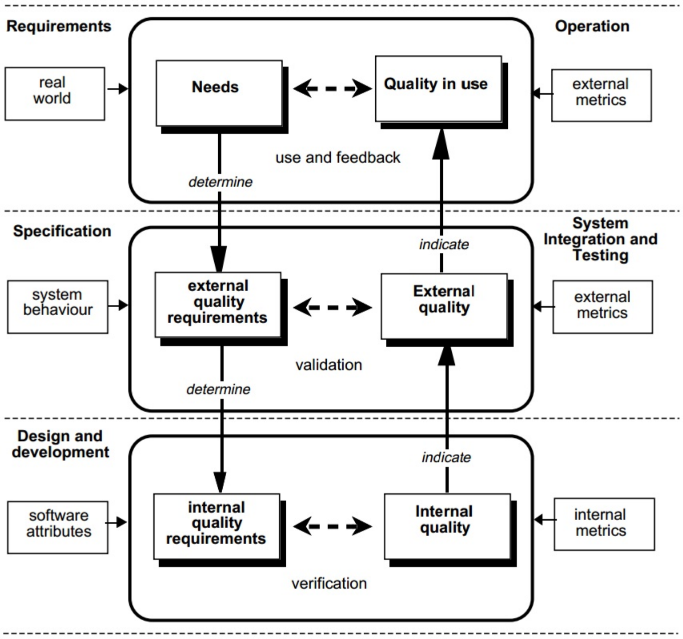


Figure 2.2.: Software quality - life cycle - ISO 9126 (Bevan and Azuma, 1997; ISO/IEC, 2001).

2.1.1. What is Maintainability?

In software engineering, the maintainability is defined by the simplicity and the ease how changes can be made in a software system in order to:

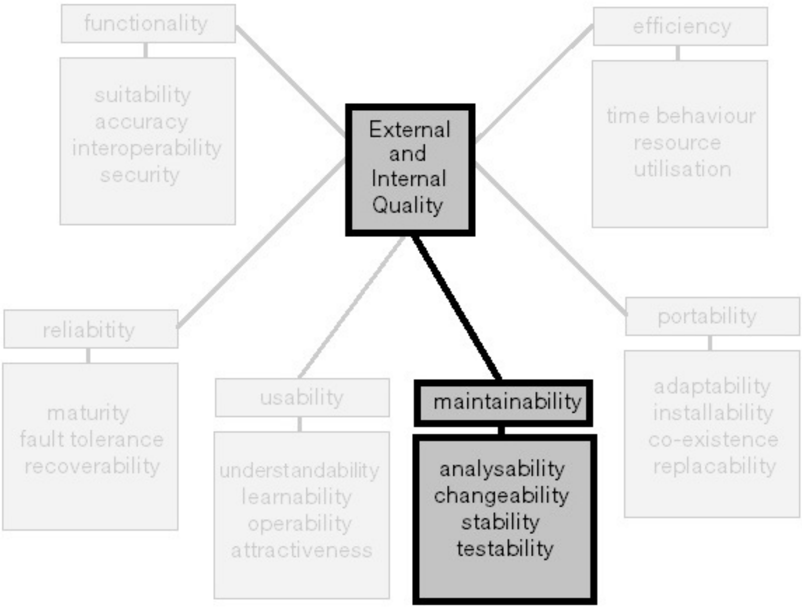


Figure 2.3.: Internal quality - Highlighted maintainability characteristic (ISO/IEC, 2001).

- correct faults,
- adapt the system to meet new requirements,
- prevent unexpected breakdowns,
- add, remove or update functionalities,
- correct errors or deficiencies when they occur,
- adapt the software or take actions to reduce further maintenance costs.

More formally, the IEEE Standard Glossary of Software Engineering Terminology (IEE) defines maintainability as: “The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.”

In a more general point of view there are a few factors that ensures the maintainability of the software, such as the ease of, understanding how the software works (Why does it works that way? What does it do?), find what needs to be changed, and making changes without introducing bugs.

The Importance of Maintainability

A maintainable software ensures that the system downtime is reduced as well as the updates done in the software are much faster and cheaper, thus, guaranteeing the possibility to reuse code due to low cost and update time.

Let's focus on a more practical example, imagine a team of software developers that are working together in a project and they split the work between each other. Now, consider one of the following events: a developer leaves the team and the others need to use his code, the need of hiring someone new that have never seen the code that the team has been developing or, for some reason, the team has to suspend the project for a certain period of time and at the end of a few months/years they have to re-use the code that was developed. Regardless the case, the developers will lose a huge amount of time trying to understand the code and how it fits together. This is a really negative aspect because the time they are losing trying to understand the code could be invested in improving it or creating new features. It is then possible to state that, a maintainable software is easier to understand for someone that have not seen it before or someone that stopped working with it for a long period of time.

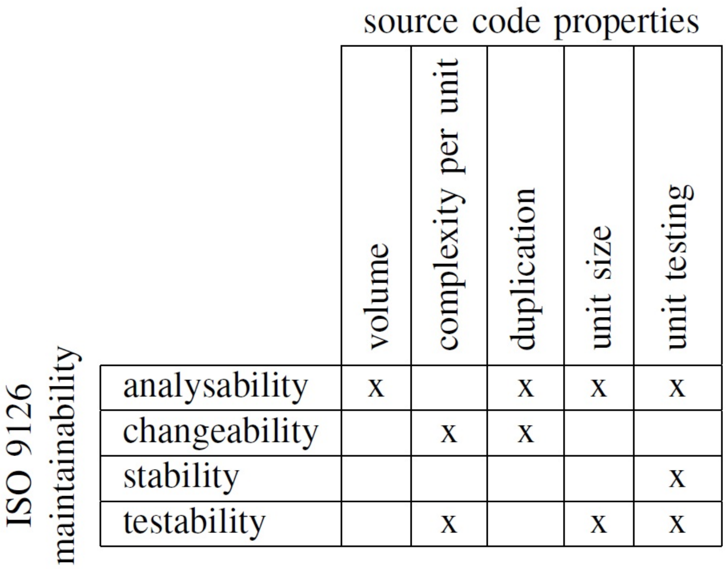
Measuring the Maintainability

As stated before, in this report, the maintainability is one characteristic of the internal quality and a way to gauge it is through the study and measurement of metrics. However, it is important to notice that not all the companies use only the metrics to measure the software quality.

On a attempt to measure the maintainability of software systems, based on the source code, Omar and Hagemeister; Coleman et al. proposed the Maintainability Index (MI) (Heitlager et al., 2007). MI is a mathematical formula which uses source code metrics as variables and the results of this formula illustrates the maintainability of the software. There is no formula considered the best to measure the MI, because it varies according to each company's focus and needs. To enable a software analysis through maintainability it is also necessary to map the system characteristics onto code properties, as shown in Figure 2.4 (Heitlager et al., 2007).

2.1.2. Metrics

Metrics are the measurement of a particular attribute of the program software. Many of this attributes point to the complexity of the program, which makes them good indicators of quality. However all of this is subjective, hence the need for quality models to define thresholds for the different metrics.



		source code properties				
		volume	complexity per unit	duplication	unit size	unit testing
ISO 9126 maintainability	analysability	x		x	x	x
	changeability		x	x		
	stability					x
	testability		x		x	x

Figure 2.4.: Maintainability sub-characteristics and the source code properties proposed by ISO/IEC (2001).

Software industry uses metrics with the objective of verifying the efficiency, progress, quality and performance of the software. They provide information about the software status, which is helpful to obtain an objective assessment of the software.

Companies may be focused on different objectives or purposes, such as: Software quality, Programming complexity, Software sizing and scheduling. It is hard to determine which metrics matter, and their meaning (Binstock; Kolawa) so the set of metrics used to measure the internal quality of the software are not yet universally defined by the Software Engineering community. However, many people have proposed different ways to measure the software attributes, but just a few metrics have prevailed. These metrics, among others, are analysed by static analysis tools and they can be sorted into:

- File-metrics,
- Function-metrics,
- Class-metrics.

This kind of sorting is the most common in software engineering, however, the ensemble of metrics may vary. Table 2.1 illustrates the set of metrics defined by ROS (Kuehn, 2013), which in turn, are the ones studied and measured in this research project.

Table 2.1.: Set of metrics used on ROS threshold (Kuehn, 2013).

File-Based	Function-Based	Class-Based
Comment to code ratio	Cyclomatic Complexity Number of Executable Lines Number of Function Calls Maximum Nesting of Control Estimated Static Path Count	Coupling Between Objects Number of Immediate Children Weighted Methods per Class Deepest Level in Inheritance Number of Methods Available

It will now be presented a more thorough description of the metrics presented in Table 2.1. Furthermore, in some metric descriptions, it is also given an example of the metric application in a function or program, in order to help understanding the operation of each one. In addition, it is important to notice that many of these definitions were based on the document written by Krusko (2004).

Comment Code Ratio

The Comment Code Ratio (COMR) metric represents the percentage (%) of the comment lines (COM) existing in a C program. Blank lines and lines of code do not count as lines of comment. The COMR value can be obtained by the following equation:

$$COMR = \frac{100(\%) * COM}{(COM + LOC)}$$

```
#include <stdlib.h>
/* This is one more example */
int main( void ){
  // This will print a pretty string on your screen
  printf{"This is a just an example"};
    return 1;
}
/*
*  You can
*  also have
*  loads of
*  comments
```

```
*/
/* This program has 9 lines of comment and 4 lines of code */
```

Listing 2.1: A simple program written in C.

According to the previous equation, the program presented in Listing 2.1 has a comment code ratio value of 69.23%.

$$COMR = \frac{100(\%) * 9}{(9+4)} = \frac{900}{13} = 69.23\%.$$

Cyclomatic Complexity

In 1976, Thomas McCabe (McCabe, 1976) introduced the Cyclomatic Complexity (CC). CC is a software metric used to measure the number of linearly-independent paths of a program or even individually to functions, modules, methods or classes. Functions with high CC are more difficult to understand and it also indicates inadequate modularization. This metric can be counted as the number of logical operators (do-while, for, if, switch and while) plus one. For instance, an empty function has a CC value of 1.

Figure 2.5 presents the flow graph of the function defined in Listing 2.2, showing the behavior of the program/function after encountering any decision and how it reacts to them, leading always to a *Final State*, in this particular case, end of function. The following function example, shown on the left side (Listing 2.2), has a CC value of **3**.

```
int function_1(int arg1, int arg2)
{
  if(arg1==1)           /* 1 */
    {
      return arg1;
    }
    else if(arg2==1)   /* 2 */
        {
          return arg2;
        }
        else
        {
          printf(...);
        }
    return 0;
}
/* CC: 2 + 1 = 3 */
```

Listing 2.2: Cyclomatic complexity of FUNCTION_1.

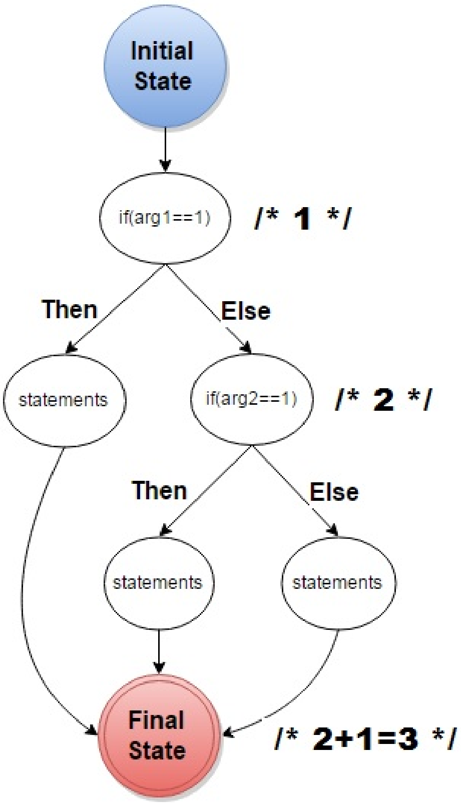


Figure 2.5.: Control flow graph of the function defined in Listing 2.2.

Number of Executable Lines

The Number of Executable Lines (NEL) metric counts all the lines of code in a function, except for comments, braces and declarations (Krusko, 2004). The following function example (Listing 2.3) has a NEL value of **7**.

```
int main(){
   int marks[10],i,n,sum=0;
   printf("Nr of students:");           /* 1 */
   scanf("%d",&n);                      /* 2 */
   for(i=0;i<n;++i){                    /* 3 */
     printf("Enter mark for %d:",i+1);  /* 4 */
     scanf("%d",&marks[i]);             /* 5 */
     sum+=marks[i];                     /* 6 */
   }
   printf("Sum= %d",sum);               /* 7 */
   return 0;
}
/* NEL: 7 */
```

Listing 2.3: Number of Executable Lines of MAIN.

Number of Function Calls

The Number of Function Calls (NFC) metric represents the number of functions called within a function. Is important to notice that the NFC is the sum of every function call and not the number of the distinct function calls. The following function example has (Listing 2.4) a NFC value of **5**.

```
void function_2(){
  int a=1,b=1,c=2;

  switch ( a ) {
    case b:
      do_something(b);       /* 1 */
      do_something_more(a); /* 2 */
      break;
    case c:
      do_something(c);       /* 3 */
      do_something_more(a); /* 4 */
      break;
    default:
      do_nothing(b,c);       /* 5 */
      break;
  }
}
/* NFC: 5 */
```

Listing 2.4: Number of Function Calls of FUNCTION_2.

Maximum Nesting of Control

The Maximum Nesting Control (MNC) is the maximum depth of nested branches and loops (If, Switch, Do While, LOOP statements, etc) in a function. The value of this metric can be reduced by simply separating this nested branches and loops into separated functions. The reduction of the MNC value provides an easier read/trace of the source code and it also reduces the CC value of the function (Krusko, 2004). The following function examples have a MNC value of **3** (Listing 2.5) and **2** (Listing 2.6).

```
void function_3(int x){
  int i=0;

  while( i < x ){       /* 1 */
    if( x == 1){        /* 2 */
      /* statements */
    }
    else if ( x == 2 ){ /* 3 */
      /* statements */
    }
    else{
      /* statements */
    }
    i++;
  }
}
/* MNC: 3 */
```

Listing 2.5: Maximum Nesting Control of FUNCTION_3.

```
void function_4(int max, int min)
{
  if(m == n){       /* 1 */
    printf("Max = Min");
  }
  else
  {
    int i = min;
    int dif = 0;
  while(i < max){ /* 2 */
    i++;
        dif++;
    }
    print("max-min=%d",dif);
  }
}
/* MNC: 2 */
```

Listing 2.6: Maximum Nesting Control of FUNCTION_4.

Estimated Static Path Count

The Estimated Static Path Count (ESPC) metric is similar to the Nejmeh's NPATH. As Nejmeh (1988) stated "NPATH, counts the acyclic execution paths through a function". The value of this metrics can be obtained from the product of each statement and their nested structures.

More specifically, ESPC (or NPath) value is obtained by the product of the statements present in Table A.14, Appendix A.2. The following function example (Listing 2.7) has a ESPC value of **8**.

```
void function_5(int arg1, int arg2)
{
  int res;
  if ( arg1 == arg2 ){
    res = 1;
    do_something(arg);
  }   /* block 1, 2 paths */
  if ( arg1 >= arg2 ){
    res = 1
    do_something(arg1);
  }   /* block 2, 2 paths */
  if ( arg1 < arg2 ){
    res = -1;
    do_something(arg2);
  }   /* block 3, 2 paths */
  /* block 4 = block 1 * block 2 * block 3 = 8 */
}
/* NPath: 8 */
```

Listing 2.7: Static Path Count of FUNCTION_5.

The result for Listing 2.7 can be obtained from the calculation of the expression:

$$NPATH(\text{FUNCTION}) = \prod_{i=1}^{\#block} block\_i = block1 * block2 * block3 = 2 * 2 * 2 = 8.$$

Number of Methods Available

The Number of Methods Available (NOM) metric basically is the sum of all the function CC values divided by the number of methods found in a file. This can be expressed by the following formula, where C_i represents the CC value of a function and *n* represents the number of functions in that class.

$$NOM = \frac{\sum_{i=1}^{n} C_i}{\#Methods}$$

Coupling Between Objects

The Coupling Between Objects (CBO) metric indicates the existing relationships between classes in the package. If a class A is coupled/related with a class B, so both of the classes have at least one relation (A ↔ B), this means that the relations between classes are bidirectional.

Weighted Methods per Class

The Weighted Methods per Class (WMC) metric simply indicates the number of methods that are defined in a class.

Number of Immediate Children

The Number of Immediate Children (NOC) metric represents the number of sub-classes that will inherit the methods from the ancestor class.

Deepest Level in Inheritance

The Deepest Level in Inheritance (DIT) metric indicates the number of antecedent classes that can affect the current class. As stated by (Chidamber and Kemerer, 1991), "The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex".

The previous five class-metrics, respectively, CBO, WMC, NOC, DIT and NOM, were proposed by Chidamber and Kemerer (1994, 1991).

2.1.3. Quality Models

Since a few decades until present, the quality models have been a well researched subject and a variety of different models have been proposed. At first, such quality models did not seem to be fully related with each other, however they all deal with software quality. Quality models are constituted by a set of rules and practices that must be satisfied while developing software. The set of rules, which encompasses the metrics and a metric value may vary depending on the companies main focus and objectives. Rules are nothing more than a metric marked with a threshold value (e.g. Maximum Cyclomatic Complexity of 10). To support this perspective Wagner (2013) also claimed that the different between quality models "is caused by the different purposes the models pursue".

As known, these quality models are defined for different contexts and purposes, although some thresholds are more restrictive that others due to the different importance of satisfying the requirements. The requirements, of each quality model, can be qualified from *Low* to *High* and this qualification is dependent of the importance of the metrics compliance.

It is also important to point out that quality models can be separated into two different types, such as:

- **Quality models oriented to violations** - to each metric evaluated, if its value is out of the scale (*MIN* and *MAX*), proposed by each quality model's threshold, then the metric is reported as a violation.

- **Quality models oriented to grades** - to each metric measured is assigned a note, depending of its value. For instance, a metric can be evaluated with a grade from 1 to 5, where 1 is considered *Bad* or *not recommendable* and 5 is considered *excellent* or *recommendable*.

The following sections presents a brief demonstration of the quality models proposed by some companies.

NASA SATC Quality Model

The main objectives of the SATC NASA's software metrics program encompasses three areas:

- Process Improvement;

- Quality Assessment;

- Process Improvement.

To meet the objectives listed above the SATC has collected measurement metrics that could assist the project managers and developers, from NASA and contractor, to evaluate the quality of their products/software and also to give a knowledge about the risks of their projects. (Software Assurance Technology Center, 2000a)

NASA SATC has *High* requirements for his proposed threshold values and those values can be found in Tables A.1, A.2 and A.3, Appendix A.1 (Software Assurance Technology Center, 2000b; Kuehn, 2013).

HIS: Hersteller Initiative Software Quality Model

HIS is composed of five groups from the Automotive manufacturer "whose goal is the production of agreed standards within the areas of Standard software modules for networks, Development of process maturity, Software test, Software tools and Programming of ECU's." (Kuder, 2008).

HIS has not proposed any class-metrics, but it has *High* requirements for his remaining proposed threshold values and those values can be found in Tables A.8 and A.9, Appendix A.1 (Kuder, 2008; Kuehn, 2013).

KTH: Royal Institute of Technology Quality Model

The main objective of KTH is to choose the software metrics that are implicitly related with the quality of real time software and to define a threshold for the metrics. The definition of this threshold will provide a better software quality and will also improve the process of software development. (Krusko, 2004)

KTH has *Low* requirements and it has only proposed function-metrics that can be found in Table A.7, Appendix A.1 (Krusko, 2004; Kuehn, 2013).

UNAK: University of Akureyri in Iceland Quality Model

The objective of the University of Akureyri in Iceland was to gather information about the different existing thresholds and create a unique threshold according to the metrics utilized in the RefactorIT[2] tool.

University of Akureyri in Iceland has not proposed any function-metrics, but it has *Low* requirements for his remaining proposed threshold values and those values can be found in Tables A.10 and A.11, Appendix A.1 (Brooks, 2008; Kuehn, 2013).

ROS Quality Model

Even knowing that there is already a threshold defined specifically for ROS, the other thresholds (NASA SATC, HIS, KTH and UNAK) were also used, because, according to Kuehn (2013), the ROS threshold was derived from those four thresholds. The ROS threshold values should be used as guidelines until the ROS community propose an official threshold for ROS. For this reason ROS can be considered to have a *Low* level on his requirements, and its threshold values can be found in Tables A.4, A.5 and A.6, Appendix A.1 (Kuehn, 2013).

SIG Quality Model

SIG created his threshold inspired in the structure proposed by ISO 9126 (Figure 2.4) and, as such, they set different values for each of the code properties (volume, complexity per unit, duplication, unit size and unit testing). The values for these properties are not evaluated for a *MIN* and *MAX* value as those mentioned above, but with ⋆ on a scale of 1 to 4, being **4** ⋆ the best grade and **1** ⋆ the worst grade.

[2] http://sourceforge.net/projects/refactorit/

In 2014, SIG provided a guidance document (Visser, 2014) with the threshold values that are required to achieve a certification of **4** $\star$. The threshold values for SIG can be found in Tables A.12 and A.13, Appendix A.1 (Heitlager et al., 2007).

SIG also uses the following formula, introduced by Omar and Hagemeister (1992), to calculate the MI of his software.

$$\text{MI} = 171 - 5.2\ln(\text{HV}) - 0.23\text{CC} - 16.2\ln(\text{LOC}) + 50.0\sin(\sqrt{2.46 * COM})$$

However, not all the institutions use existing MI formulas, for example, Microsoft (2015) have defined his own MI formula and threshold. The result values in its threshold range between 0 and 100, where the higher values means a better maintainability, while the lower value in turn, means a worse maintainability. More objectively, a result between: 20 and 100 indicates that the code has good maintainability, 10 and 19 indicates that the code is moderately maintainable and the remaining results indicates low maintainability (Microsoft, 2015).

2.2. Robot Operating System

Developing software for robots is not an easy task. Thus, many robotic middleware frameworks have been proposed, with the purpose of easing the development of software for robots. Some of these robotic middleware frameworks are fully open source and they possess an ascendant community. The Robot Operating System[3] (ROS) is the robotic middleware focused in this research project, due to its large community, the huge number of available projects and its growing maturity.

According to Quigley et al. (2009), Willow Garage, a Personal Robots[4] (PR) program, together with the STanford AI Robot[5] (STAIR) project at the Computer Science Department of Stanford University, had the need to create a system that meet a specific set of challenges for developing large-scale service robots. ROS was the system developed by these teams and its architecture is more general than the service-robot and mobile-manipulation domains. This system was designed to be a open source, thin, multilingual and tools-based framework with a peer-to-peer architecture. The development of ROS took place around the 2000s, while in 2007, Willow Garage with the help of a huge number of researchers, provided enough resources to expand the concepts of ROS and create well-tested implementations.

[3] http://www.ros.org/

[4] http://pr.willowgarage.com/

[5] http://stair.stanford.edu/

ROS is an open-source, collection of software structures for a robot. It provides the expected standard services from an operation system such as hardware abstraction, implementation of commonly-used functionality, low-level device control, package management and message-passing between processes. It also provides libraries and tools for obtaining, running, writing and building code on multiple computers. Even knowing the importance of a low latency and reactivity in a robot control, ROS is not a real-time system, but it is possible to integrate ROS in real-time code. ROS main developing languages are C++ and Python, although it offers support for many other programming languages, and a support for C language will be possible with the release of ROS 2.

The ROS system is organized into packages, metapackages and stacks[6]. A package in ROS might be constituted for ROS nodes, a ROS-independent library, a dataset, configuration files and a third-party piece of software. Briefly, the packages contains all the source code that is used by robots. The metapackages, unlike packages, do not contain any tests, code, files, or other items usually found in packages. A metapackage is only responsible to make reference of the related packages. A ROS stack is a simple folder that contains a different set of packages.

A system in ROS is constituted by a set of nodes that communicate with each other following the *publisher-subscriber* model. The nodes have different types of communication such as streaming topics[7], RPC services[8], and the Parameter Server[9] (Quigley et al., 2009; Conley, 2012). Figure 2.6, adapted from Santos (2015), illustrates a diagram with the operation of a ROS system. ROS is composed by different layers, however, this research project is only focused on the operational layer and not on ROS core itself.

In order to validate this project several robots, implemented using ROS, such as Kobuki[10], Turtlebot[11], PR2[12], Motoman[13], and some others were used as case of study. To these robots it was performed a static analysis to their source code in order to access its quality. This analysis is possible due to the ROS policy of keeping all the developed code public on GitHub[14].

Initially, in order to get a better understand of the ROS arquiteture, the robot Kobuki was studied due to his simplicity and low complexity compared to other existent robots. Kobuki is a low-cost mobile robot base that can be used by other robots, such as the Turtlebot. Both Kobuki

[6] ROS meta-packages, packages and stacks: http://www.ros.org/browse/list.php

[7] http://wiki.ros.org/Topics

[8] http://wiki.ros.org/Services

[9] http://wiki.ros.org/Parameter%20Server

[10] http://kobuki.yujinrobot.com/home-en

[11] http://www.turtlebot.com/

[12] https://www.willowgarage.com/pages/pr2/overview

[13] http://www.motoman.com/

[14] https://github.com/

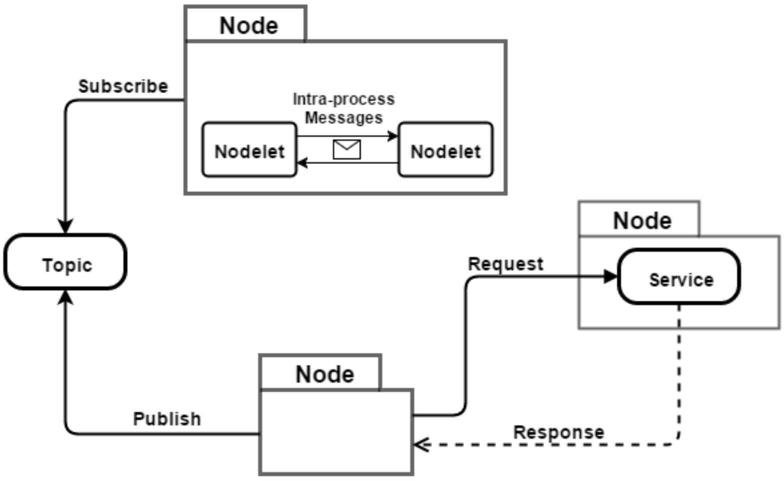


Figure 2.6.: Simplified diagram with the behavior of a ROS system (Santos, 2015).

and Turtlebot, but more specifically the Kobuki, are designed with education and research purposes in mind More particularly, Kobuki itself already provides a set of features, such as keyboard remote operation, autodocking, visualisation and simulation tools, and a safety controller, that uses sensor information to randomly navigate without colliding with any other object.

2.3. Static Analysis Tools

The analysis of the static properties of code is really important if we want to assess the software quality. In the area of robotics, the robots are considered safer if the software developed to them is in compliance with the rules and practices of a quality model (i.e. software with good quality). Nowadays, with the growth of the application security market it became really important to ensure that the software is safe and maintainable. This growth influenced also the creation and development of a variety of static analysis tools, some of them commercial/paid and others open source/free. Both, commercial and open source tools have its own pros and cons, however, the choice of the most viable and appropriate tool is dependent on the company aim, objectives, budget, target programming language, and many others. Table 2.2 contains a set of pros and cons, introduced by Vonnegut (2015) to these two types of tools.

Is important to be aware that some tools are able to analyse more than a single programming language. However, the tools of interest are the ones capable of analysing C++ code, because this is the programming language focused in this project.

This section provides a brief description of the tools studied in this project, as well as the importance in developing or extending a tool that is tailored for ROS.

Table 2.2.: Pros and cons of commercial and open source tools (Vonnegut, 2015).

	Commercial Tools	**Open Source Tools**
Pros	• Guaranteed support • Main focus on their product • Better reporting • More complete vulnerability coverage • Long term viability with road-map	• Free of upfront costs • More eyes on the code • Flexibility • Smart way to ease into Static Code Analysis tools
Cons	• Prohibitive cost • Too big companies lose their passion and support quality • Only the vendor can fix issues in the code	• Limited support and liability • More eyes don't *necessarily* make all bugs shallow • Cost in resources and staff time • Harder to Scale – Especially coding in different languages

Open Source Tools

This subsection presents some of the most relevant open source tools studied during the time frame of this research project.

Lint (Joh) is a static analysis tool which works similarly to a compiler, more particularly it checks the syntactical correctness of C and C++ source code files. More than a tool, Lint is a term given to a program responsible to perform static code analysis and detect suspicious usage in source code files for any programming language.

Lint was not a tool of interest for this project, because it does not perform any analysis to code metrics. This tool is mainly focused on checking the syntactical correctness of C and C++ source code files. However, it was relevant to study this tool because it was one of the first static analysis tools and the term "lint" was stemmed from it.

ROS Static Analysis Tool (Santos, 2015) is a web-based platform to gauge the code quality of ROS projects, offering a visual component which reports the metrics with no compliance and a graph to check the dependencies between packages. Currently, this is the only existent tool directly tailored to ROS. According to Santos (2015), it is a flexible and extendable tool (extendable with plugins), so it should be suitable to many different types of quality analysis. The extensibility of the tool is achievable through the creation of new plugins. Currently this tool only supports analysis for coding standards, but apart from the analysis the tool also has a graphic component, which allows to check the dependencies between packages, as well as more detailed information from each package (e.g. name, description and violated rules). Due to its flexibility and extensibility, the tool can be improved through the creation of plugins or new components. For instance, the addition of a new kind of static analysis is considered an improvement.

SonarQube[15] (previously known as *Sonar*) is a web-based platform to manage code quality, offering visual reporting on and across projects. Just like ROS Static Analysis Tool, SonarQube extensibility is achieved through the creating of new plugins. Currently, the tool covers more than twenty programming languages, due to the large amount of existing plugins. SonarQube is a open source tool, however SonarSource proposed some commercial plugins to cover additional programming languages and to manage projects along with Professional services. As opposed to ROS Static Analysis Tool, SonarQube is not tailored for ROS, however it is capable of measuring the quality of many other types of programming languages.

CCCC[16] is a static analysis tool which analyses C++ source code files and generates a report with the various internal quality metric measurements. According to the others open source tools studied the CCCC is the one that arouses a greater interest, because the number of metric measured is far superior to any of the other tools.

OCLint[17] is a static analysis tool for C, C++ and Object-C that detects potential problems on code, such as (Saito, n.d.):

- Possible bugs - empty if/else/try/catch/finally statements,
- Unused code - unused local variables and parameters,
- Complicated code - high cyclomatic complexity, NPath complexity and high NCSS,
- Redundant code - redundant if statement and useless parentheses,
- Code smells - long method and long parameter list,
- Bad practices - inverted logic and parameter reassignment,
- and many others.

OCLint is a standalone tool that runs on Linux and Mac OS X platforms and it can executed through a shell.

Vera++[18] is a programmable static analysis tool which analyses C++ source code files. Vera++ can be seen as a parser for C++ language and the results of its parsing are presented in scripts.

[15] `http://www.sonarqube.org/`
[16] `http://sourceforge.net/projects/cccc/`
[17] `http://oclint.org/`
[18] `https://bitbucket.org/verateam/vera/wiki/Home`

The scripts are responsible for conducting the analysis of the parsed information, so the users can change the scripts to meet his own needs.

CLoc[19] is a portable tool entirely written in Perl which analyses the source code of many different languages. Some of the code metrics analysed by the tool are: blank lines, comment lines, and physical lines.

SLOCCount[20] is a set of tool responsible to count the number of physical source lines of code for many programming languages. According to the author this tools was used in his paper (Wheeler, 2001) to measure the source lines of code of the entire GNU/Linux distributions.

CLang[21] is a static analysis tool for C, C++ and Objective-C programs that can be executed within Xcode[22] or as a standalone tool, enabling its use in other UNIX[23] operating systems instead of only Mac OS X.

Locmetrics[24] is a static analysis tool executable in Windows operating systems analyses C#, C++, Java, and SQL source code files. This tool is capable of measuring some code metrics, such as: lines of code, blank lines of code, comment lines of code, lines with both code and comments, logical source lines of code and cyclomatic complexity.

Egypt[25] is a small script written in Perl that uses both GCC's and Graphviz[26]'s capabilities to generate a function call graph of a C program. The tool also supports C++ source code files, but the support for this programming language is limited.

c_count[27] is an easy portable static analysis tool that measures some code metrics from C source code files, such as number of lines and statements, and the percentage of whitespace, comments and code.

c_lines[28] is a simple script written in Awk[29] whose main function is to measure the lines of code in a C source code file, not including comments, blank lines or form feeds.

ftrace[30] is a tracing tool used to calculate the system calls, function calls and signals of a directory or a single C source code file.

[19] https://github.com/AlDanial/cloc
[20] http://www.dwheeler.com/sloccount/
[21] http://clang-analyzer.llvm.org/
[22] https://developer.apple.com/xcode/
[23] http://www.unix.org/
[24] http://www.locmetrics.com/
[25] http://www.gson.org/egypt/egypt.html
[26] http://www.graphviz.org/
[27] http://invisible-island.net/c_count/c_count.html
[28] http://stjarnhimlen.se/snippets/c_lines.awk
[29] http://web.mit.edu/gnu/doc/html/gawk_toc.html
[30] https://sourceware.org/frysk/manpages/ftrace.1.html

Commercial Tools

This subsection presents the commercial tools studied during the time frame of this research project. However, the tools of interest for this project are the open source, that is why just a few commercial tools were studied.

QA·C++[31] is the tool proposed by Kuehn (2013) to measure all the proposed metrics in Table 2.1. QA·C++ is a static analysis tool for C++ projects and it analyses the code against a chosen coding standard, with metrics and code structure visualizations. The tool has compliance packages for MISRA C++, HIC++ and JSF AV C++ coding standards.

Testwell CMT++[32] is a static analysis tool that measures some code metrics from C++ source code files, such as: lines of code (number of blank lines, lines with comments, physical lines and program code lines), Halstead's metrics, McCabe cyclomatic number and maintainability index.

Parasoft C/C++test[33] is a static analysis tool which performs static code analysis, data flow analysis, and metrics analysis for C and C++ source code files.

The idea of creating or extending an autonomous open source static analysis tool directly tailored to ROS was really interesting, because only some commercial tools are capable of measuring all the metrics outlined above in Table 2.1, Section 2.1.2 and, on the other hand, these tools are not tailored for ROS projects. Besides that, the creation or improvement of a open source tool would be an huge contribute to ROS community.

Fortunately, both ROS Static Analysis Tool and SonarQube are extendable, but the one that most closely matches the intended for this project is the ROS Static Analysis tool, because it is already tailored for ROS and it is extensible for any other kind of static analysis, like the one studied in this project. This extensibility is achievable through the creation and inclusion of one or more plugins responsible to check the compliance of an huge number of ROS packages.

[31] http://www.programmingresearch.com/products/qacpp/

[32] http://www.verifysoft.com/en_cmtx.html

[33] https://www.parasoft.com/product/cpptest/

3. Contribution

As presented in Section 2.1, there is a wide variety of quality models, as well as the set of metrics proposed by each one of them. However, a large part of the assessed quality models (including the one proposed for ROS) are thresholds, and therefore, the quality models of interest for this project are the ones oriented to violations and the set of metrics proposed from them are the ones present in Table 2.1. The main focus for this research project is to develop and implement a system for measuring the code quality of ROS projects in a existent static analysis tool already tailored for ROS. Remarkably, as concluded in Section 2.3, the *ROS Static Analysis Tool* (Santos, 2015) is the most suitable tool for this project, because it allows the definition of different quality model's thresholds, as well as the definition of a new analysis method for measuring the ROS projects, in this particular case, the metric measurement. Hence, in this context, the first step consists in encapsulating some of the static analysis tools presented in Section 2.3 as plugins for this system, therefore this system can be seen as a toolset for the *ROS Static Analysis Tool*. The following step consists in defining a rule (Rule: a metric marked with a threshold value) for each threshold set by the quality models. The ROS Static Analysis Tool is itself oriented towards violations, and so does the threshold rules, since they detect violations to the quality model. Finally, the toolset is applied to a number of repositories, serving not only as validation of the plugin but also to the study the state of the art of ROS corpus.

In order to avoid the constant use of the tool name, from now until the end of the report the ROS Static Analysis tool can also be referred to as the *main tool*, since the developed system is a toolset for this tool. More specifically, the main tool is responsible for fetching source code from the selected ROS projects, executing the plugins on them and display the results through a graphic component.

The following sections are destined to the presentation of the main tool, the developed toolset and data analysis. More particularly, Section 3.1 contains a more detailed presentation of the main tool, emphasizing its architecture, such as creations of plugins and how to use them. Subsequently, Section 3.2 contains a deep introduction to the toolset, referring its functioning, capabilities, the calculated metrics and how to get the information about the quality measurements.

Finally, Section 3.3 contains the data acquired after evaluating a number of ROS repositories.

3.1. ROS Static Analysis Tool Architecture

The ROS Static Analysis tool is executable from a UNIX shell and it is responsible for managing the repositories, updating the database and executing the plugins for the static analysis on source code. The execution of the main tool can be separated in two phases, the startup phase and the execution phase. Consecutively, the execution phase is also subdivided into three different stages, the *update* stage, the *analysis* stage and the *export* stage.

3.1.1. Startup Phase

Users have the control over the executable stages of this tool, and these stages are controlled through the startup files and the command line arguments passed during the startup phase. These arguments allow the user to skip an entire stage of execution or, more specifically, skip some operations in each stage. If no arguments are granted in the startup phase, the main tool does not skip any of the execution phases.

The startup files are composed by three different types of files, *configuration* file, *rules* file and *distribution filter* file. The configuration file contains a list of plugins that can be loaded and executed during the execution phase of the main tool. To avoid possible errors it is imperative that the plugin name matches the file name. For instance, if the configuration file contains a plugin with the name of “*some_plugin*”, a file with the name “*some_plugin.py*” must exist. Listing 3.1 shows an example of a configuration file, which contains the declaration of the plugins that will be presented in the following section.

```
%YAML 1.1
# Configuration file
---
plugins_unused: # The plugins listed here will be skipped
    basic_metrics:
        type: analysis
        subtype: metrics
plugins_all:
    cccc_rules:
        type: analysis
        subtype: rules
    oclint_rules:
        type: analysis
        subtype: rules
```

Listing 3.1: Example of a configuration file, named *config.yaml*.

The previous example shows how the plugins are declared, so it is assumed that a folder named *plugins* exists, containing three plugins with the names of *cccc_rules.py*, *oclint_rules.py* and *basic_metrics.py*. The rules file contains the list of rules to which the plugins may register violations. For each defined rule is possible to add a set of tags, which are used in the process of filtering the rules violated. These tags provide the filtering of a set of violations found for a certain quality model, or even a certain metric. Listing 3.2 shows an example of how rules are currently defined. This example only contains a small set of the defined rules, however the full set of defined rules, for the different quality models, can be found in Appendix B.

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

Listing 3.2: Example of a set of rules defined in the rules file, named *rules.yaml*.

As a side note, editing or removing the existing rules might cause the malfunctioning of the plugin. The distribution filter file contains the set of packages that are analysed during the execution phase of the main tool. The user has the possibility of choosing the packages that are analysed, but it must be ensured that the packages that are identified in the filtering file are also present in the *distribution* file. Listing 3.3 shows an example of the filter file, containing a set of Kobuki's packages that are used, as evaluation, in the data analysis.

```
%YAML 1.1
# Filter file.
---
packages:
    kobuki:
        - kobuki_keyop
        - kobuki_node
        - kobuki_random_walker
        - kobuki_safety_controller
    kobuki_core:
        - kobuki_driver
    yujin_ocs:
        - yocs_cmd_vel_mux
        - yocs_safety_controller
        - yocs_velocity_smoother
```

Listing 3.3: Example of a filter file, named *filter.yaml*.

To contextualize, the distribution file follow a concept acquired from ROS, which in turn follows the REP 141 (Thomas, 2013). As stated from Thomas (2013), "REP specifies a set of files which define ROS distributions and facilitate the building, packaging, testing and documenting process". This REP is a revision of the REP 137 (Tully Foote and Mathieu, 2013) version. More specifically, the file contains all the known packages from a ROS distribution and the web address (also called a URL[1]) of their GitHub repositories. Is also important to notice that all the previous files are written in a YAML[2] format, because of its human friendly data serialization.

[1] http://dictionary.reference.com/browse/uniform%20resource%20locator

[2] http://yaml.org/

3.1.2. Execution Phase

Right after the startup phase, the main tool enters in the first stage of the execution phase, the update stage. During this stage the tool, through the reading of the distribution file and filter file, tries to update the local source files and the database, with the information of the remote repository. In this stage the main tool also reads the rules file, in order to keep the analysis rules updated.

Immediately after the update stage comes the second stage of the execution phase, the analysis stage. This is the stage responsible for all the analysis performed at the source code by executing the plugins provided to the tool. The plugins are responsible for verifying the rules violated in the ROS projects, which in turn can be visualized in the main tool. A deeper explanation is given in the following sections.

The last stage of the execution phase is the export stage. This is the stage where the data, present in the database, is exported under data files. All the exported data, like the rules violated in each file or package, can be visualized through the graphic component of the main tool. A deeper presentation of how the graphic component can be used, to inspect the exported data, is given on the Subsection 3.2.2.

3.1.3. Creating Plugins to the Main Tool

Creating plugins is a way to extend the main tool functionality and, as previously mentioned, the main purpose of this research project is the development of plugins capable of assessing the code quality of the ROS projects.

In the current state, the main tool provides a small application program interface (API) to plugins developed in Python. With the aid of the functions provided by the API, the plugins are capable of connect, withdraw information and register its results in the main tool's database. The following list presents some of the functions provided by the API and utilized in the developed toolset.

> PLUGIN_RUN(***self***): PLUGIN_RUN is the function responsible to create a bridge between the plugins and the main tool. This is also the first function to be executed in every plugin, and it has a similar behavior of the main functions defined, for example, in C and C++. The PLUGIN_RUN function must be defined in every plugin, otherwise the main tool will not be able to load it or execute it.
>
> GETRULEINFO(***self***, ***name***=*None*): GETRULEINFO is the function responsible to get the registered rules entries from the database.

GETPACKAGEINFO(***self***): GETPACKAGEINFO is the function responsible to get the registered packages entries from the database.

GETFILEINFO(***self***, ***package_id***=*None*, ***ext***=*None*): GETFILEINFO is the function responsible to get the registered source code files entries from the database.

WRITENONCOMPLIANCE(***self***, ***rule_id***, ***package_id***, ***file_id***=*None*, ***line***=*None*, ***function***=*None*, ***comment***=*None*): WRITENONCOMPLIANCE is the function responsible to register noncompliance occurrences in the database. The arguments *rule_id* and *package_id* are mandatory, while the others are just optional since they are destined to additional information, such as: source file, line number, function name, and a custom commentary.

3.2. Rule Violation Finder

Currently, there are many and different ways to extract the information about the source code metrics, but each static analysis tool provide his own way of doing it. Considering this information, the creation of a plugin to standardize the obtainment of these metrics becomes justifiable. The plugin (i.e. Rule Violation Finder) is incorporated into the *ROS Static Analysis Tool* and it can be seen as a toolset (i.e. a set of plugins).The plugins that constitute the toolset are, in turn, scripts that incorporate some of the studied and presented tools in Section 2.3, in order to assist the measurement of highest number of the required metrics. The toolset is targeted to quality models oriented to violations, so it uses the thresholds proposed by: NASA SATC, HIS, KTH, UNAK and ROS. Thus, the metrics which are intended to be measured are the ones that were previously presented in the Table 2.1 of Section 2.1.2. SIG's proposed quality model is not considered because is it based on grades and, as stated before, this toolset is only focused on quality models oriented to violations.

Despite the large number of studied tools, presented in Section 2.3, the only ones used by this toolset are the CCCC and OCLint, because these are the ones that best suit the needs/requirements of this project. More specifically, the CCCC tool works in different ways depending on the files passed as argument in its execution. If a set of files are passed as argument to the tool, such as a ROS package, a report is produced containing the sum of all the File-based metric measurements of that package. In case of passing a single file as input, the tool produces several reports with the Function-based and Class-based metrics of that file. This reports are exported in two different formats, such as HTML[3] and XML[4], but only the exported XML reports are used by the plugin. The OCLint tool, in particular, is used to obtain the Estimated Static Path Count (NPath complexity). OCLint uses *GNU Compiler Collection (GCC)*[5] to compile the source code of a file before making his analysis. If *GCC* does not report any errors or warnings the tool outputs a set of information about the file, containing the NPath complexity value. It is also relevant to notice that only the values that exceed the threshold are reported when the OCLint finish his execution, so is important to set NPath maximum value to 0. With the NPath complexity rule set to 0 every compilation reports the NPath value as a violation. This tool allows the exportation of the results in different formats, but the one used to export the required information is the XML format. An example of the reports generated from these tools (CCCC and OCLint) can be found in the Sections C.1 and C.2, Appendix C.

Metrics such as NEL and NFC were not possible to be obtained, because none of the studied tools are able to measure them. The tool responsible to measure the ESPC (NPath) NPath complexity metric (OCLint), as previously mentioned, only provides an output with the metric values, if the *GCC* does not find any errors or warnings during his compilation. In many cases the majority of the errors found, during the compilation, are caused by the lack of header files, due to their absence or because the compiler can not locate them.

Given the low rate of the obtained values for the NPath, it can be assumed that this metric will not be measured. But whenever the NPath metric is obtained it is displayed on the main tool, if it exceed any of the thresholds.

Table 3.1 illustrates the metrics calculated from each plugin, as well as the non calculated metrics.

Table 3.1.: Metrics obtained by each plugin.

Plugin	***oclint_rules*** (is a script that incorporates the OCLint tool)	***cccc_rules*** (is a script that incorporates the CCCC tool)	Non-Calculated Metrics
Metrics	Estimated Static Path Count Maximum Nesting of Control	Comment to code ratio Cyclomatic Complexity Coupling Between Objects Number of Immediate Children Weighted Methods per Class Deepest Level in Inheritance Number of Methods Available	Number of Executable Lines Number of Function Calls

[3] http://www.w3.org/html/

[4] http://www.w3.org/XML/

[5] https://gcc.gnu.org/

The sections that follow provide a more detailed analysis of the measured metrics, the plugin functioning, how the violations are found and how they are presented to the user.

3.2.1. Rule Violation Finder Architecture

Rule Violation Finder is the name given to the toolset responsible to measure and find the rules that exceeded the threshold values. The toolset is composed of two plugins, both of them are defined in the main tool and they are responsible for measuring the different code metrics.

As mentioned in Section 3.1 the main tool downloads the ROS projects to a certain path and consecutively the files, of that project, are added to a database. As previously stated, the plugins must have a function named PLUGIN_RUN, which connects them to the main tool, and the way to acquire the location of the packages and their files is done through a function called GETFILEINFO (an usage example of this function can be found in Listing 3.4).

```
def get_files(ctx):
    cpp = ctx.getFileInfo(ext="cpp")
    hpp = ctx.getFileInfo(ext="hpp")
    files = []
    for f in hpp:
        files.append(f)
    for f in cpp:
        files.append(f)
    return files
```

Listing 3.4: An usage example of the GETFILEINFO function, extracted from a made up script named *cccc_rules*

Right after populating the database, all the packages and files are individually read measuring its metrics values, through the tools mentioned in the previous subsections. Right after getting these measurement values an analysis is conducted in order to check if any of the metrics exceeded the value of the thresholds, proposed in the Section 2.1.3. Finally, after the analysis is completed, the plugins uses the WRITENONCOMPLIANCE function to add a line to the database with the rule violated, file location, metric description and the respective threshold (an usage example of this function can be found in Listing 3.5).

```
...
def handle_mnc(ctx, package_id, file_id, function, value, file_path):
    # Maximum nesting control
    if int(value) > 4:
        ctx.writeNonCompliance(13, package_id, file_id=file_id,
                line=0, function=function, comment="Maximum nesting value is
                    greater than 4")
    if int(value) > 5:
        ctx.writeNonCompliance(14, package_id, file_id=file_id,
                line=0, function=function, comment="Maximum nesting value is
                    greater than 5")

def handle_npath(ctx, package_id, file_id, function, value, file_path):
    if value > 80:
        ctx.writeNonCompliance(16, package_id, file_id=file_id,
                line=0, function=function, comment="NPath is greater than 80
                   ")
    if value > 250:
        ctx.writeNonCompliance(17, package_id, file_id=file_id,
                line=0, function=function, comment="NPath is greater than
                   250")
...
```

Listing 3.5: An usage example of the WRITENONCOMPLIANCE function, extracted from a made up script named *cccc_rules*.

As previously stated in this chapter, the toolset is composed by two made up plugins, both connected to the main tool, with the names of *oclint_rules.py* and *cccc_rules.py*.

oclint_rules Plugin

The *oclint_rules* is a script written in two programming languages, Perl[6] and Python. This script runs the OClint tool, which is responsible to measure the NPath metric (see Subsection 2.3), and it also calls an external Perl script, which is responsible for obtain the MNC metric value, since the studied automated analysis tool are not able to do it. The Perl script runs in every file and uses regular expressions[7] to find the existing functions. Within each function the script matches any of the following statements *if*, *while*, *do*, *for* and *switch*. Each of the previous statements, followed by the opening brace('{'), marks the beginning of a nested structure and the end of that structure is marked by the matching closing brace('}'). The nested level is increased every time

[6] https://www.perl.org/

[7] http://perldoc.perl.org/perlre.html

a statement is found and decreased right after finding the matching closing brace, keeping stored the highest level achieved in the function.

In some particular cases the statements can be used without the braces. In those cases the nested level is increased to check if it surpassed the maximum nested level and decreased right after it. The following example gives a brief demonstration of the script operation.

```
// Cur_Level = 0, Max_Level = 0
while (a) {   // Cur_Level = 1, Max_Level = 1
    if (b) {  // Cur_Level = 2, Max_Level = 2
        do_something();
    }         // Cur_Level = 1, Max_Level = 2
}             // Cur_Level = 0, Max_Level = 2
while (e) {   // Cur_Level = 1, Max_Level = 2
  if (d) {    // Cur_Level = 2, Max_Level = 2
    for(int i=0;i<10;i++);    // Cur_Level = 2, Max_Level = 3
    }         // Cur_Level = 1, Max_Level = 3
}             // Cur_Level = 0, Max_Level = 3
```

Listing 3.6: Execution of the *MNC_Metric* script.

The output from that script is a simple file containing the MNC metric value from the different functions. Therefore, the file is read from the rest of the *oclint_rules* script.

cccc_rules **Plugin**

The *cccc_rules* is a script written in Python. This script runs the CCCC tool, which is responsible to measure a various number of metrics, such as: COMR, CC, CBO, NOC, WMC, DIT and NOM.

3.2.2. Visualizing the Violated Rules

The visualization of the violated rules is supported by the graphic component of the ROS Static Analysis tool. As mentioned in the Section 3.2.1, the displayed rules in the graphic component are the ones exported during the execution phase of the main tool.

This component builds a diagram with the packages and their dependencies to the other packages. The packages are represented by the nodes and its dependencies are represented by the existing edges between nodes.

The main tool allows the user to interact with the diagram. A few diagram interactions are provided, such as, zooming in, zooming out, transposition and node selection. A detailed information, of the node, is displayed when the node is selected. This information appears in the side bar and it contains the package name, the number of rules violated in that package, the package description and a list of dependencies. The side bar also contains two fields designed for filtering. This fields gives, the user, the possibility to filter the set rules that are reported from a given package, for one or more quality models and/or metrics. The user can access all the information (file name, rule violated, description and more, if the information is available) related to the rules violated in a given package and this information is displayed, as a pop-up, when the number that appears after the "Rule violations" sentence is clicked. The pop-up also allows the user to filter the shown information by a set of tags and the filter works in the same way as the side bar of the diagram. For instance, the tag "ROS" filter all the violations for a set of violations to the ROS threshold.

The operation and demonstration of the use of this graphical component is shown in the following section, along with the data analysis.

3.3. Data Analysis

This section is intended for presenting the information obtained through the source code analysis of several robots. Until the current date, and as stated in Section 2.2, several robots have been analysed, however just two of them will be presented in more detail during this section, Kobuki and Turtlebot. Solely these robots are presented in more detail, because they were a part of the study process, helping understand the architecture and operation of ROS. Moreover, these robots are also used in education and as research for robotics.

The source code analysis of these robots is conducted through the use of the toolset created in this research project. As previously mentioned in Section 2.3, the toolset is comprised by some of the studied open source static analysis tools, in order to measure the source code and check the compliance of the rules. The way how the metric measurement values are obtained and the tools used in the toolset was also expressed in Section 2.3 and Table 3.1.

3.3.1. Analysis of Kobuki

Kobuki consists of 63 packages, where 7 of them are metapackages (i.e. do not contain any C++ code files). These packages contain around 1360 C++ source code files, with more than 150.000

lines of code and has 2031 rules violated. For instance, in this particular case a violation is found on an average of 73 lines of code.

As stated previously in Section 3.2.2, after executing the analysis on several ROS packages the graphic component builds a diagram with all the analysed packages and their dependencies to other packages. Figure 3.1 illustrates the diagram built from the graphic component of the Kobuki's packages, as well as their dependencies. On that same figure we can identify nodes with different color gradients. The color of nodes is directly connected with the number of violations found in the package. A package with fewer violations will display a lighter color, than a package with many violations, which in turn will display a darker color. The displayed packages in this figure are not filtered for any quality model in specific, so this diagram shows all the non-compliance rules found for every quality model.

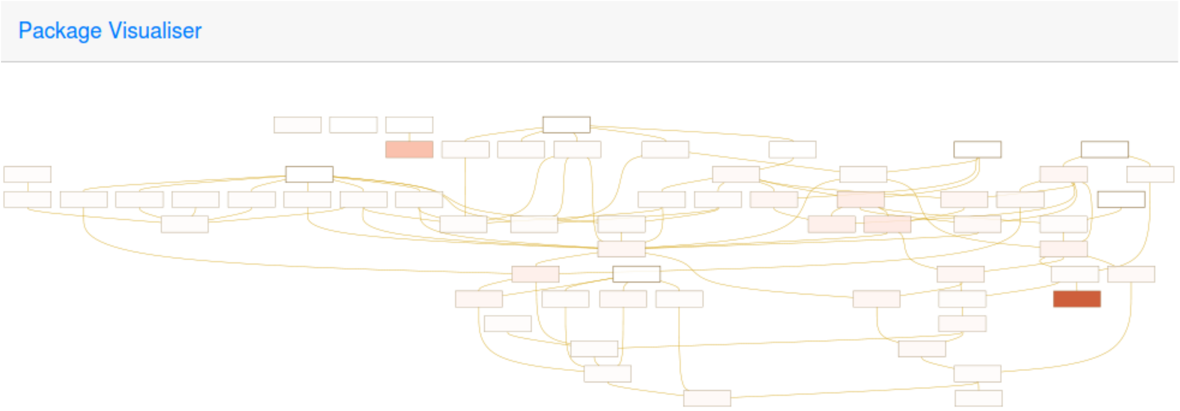


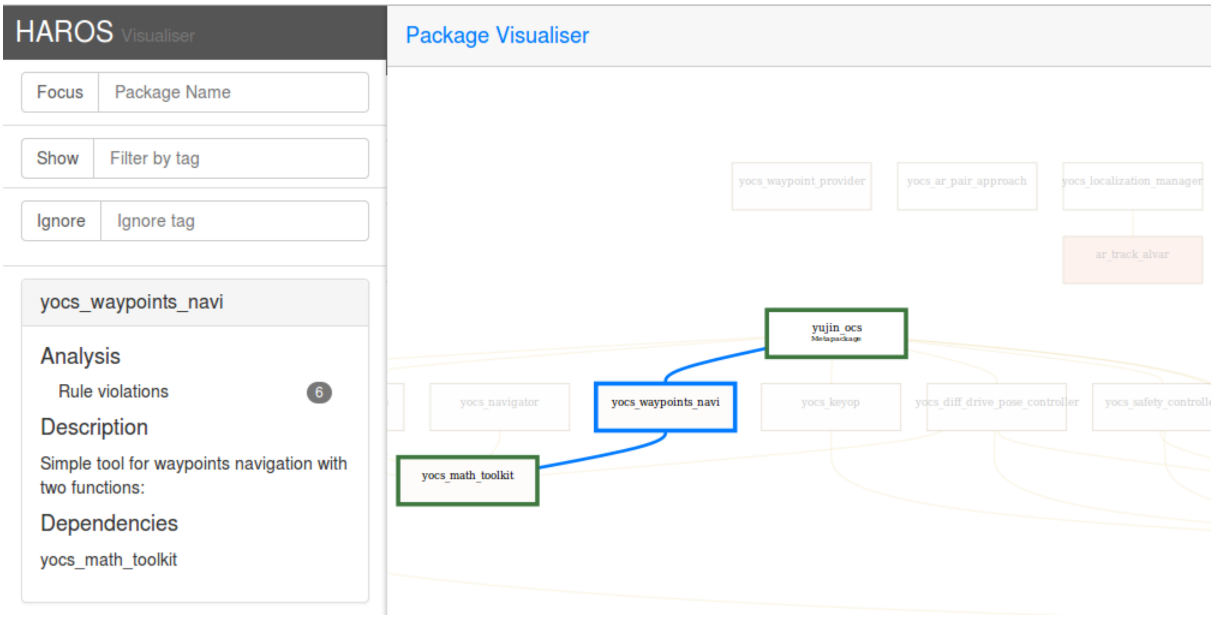
Figure 3.1.: Preview of the Kobuki's packages and their dependencies.

The visual component also allows the user to select a certain node (i.e. package), in order to obtain a more detailed information about it. For instance, Figure 3.2 shows the displayed side bar, with the *yocs_waypoints_navi* package selected.

For instance, after clicking on the number that is shown on the side bar of Figure 3.2 (i.e. number 6) that appears right after the "Rule violations" sentence a pop-up will be displayed with all the rules violated for the selected package. Figure 3.3 shows the displayed pop-up, containing the number of violations for the ROS quality model.

Table 3.2 presents the total number of rules violated, as well as the number of rules violated per quality model (NASA SATC, HIS, UNAK, KTH, ROS) for a set of Kobuki's packages.

According to this table, we can assume that the volume of the code is connected with the number of violations found. Thus, a package that contains a large volume of lines of code have a higher probability of presenting more violations, than a package with a lower volume of code.


HAROS Visualiser

Focus | Package Name

Show | Filter by tag

Ignore | Ignore tag

yocs_waypoints_navi

Analysis

Rule violations 6

Description

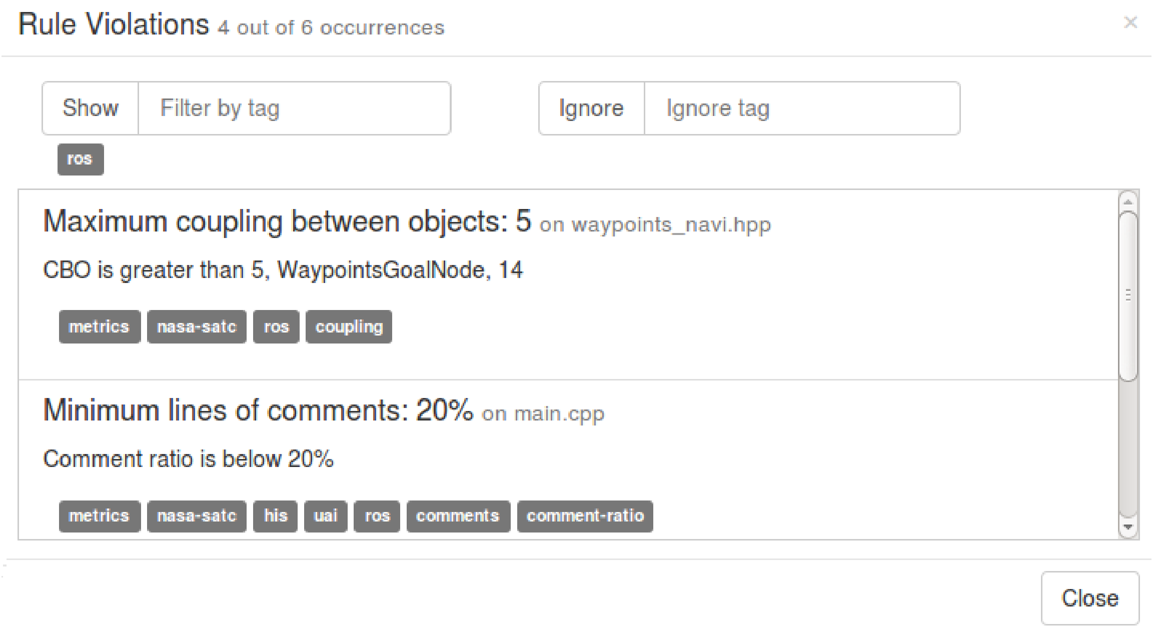
Simple tool for waypoints navigation with two functions:

Dependencies

yocs_math_toolkit

Package Visualiser

Figure 3.2.: Side bar with the information of the selected package, *yocs_waypoints_navi* (blue box).


Rule Violations 4 out of 6 occurrences

Show | Filter by tag

Ignore | Ignore tag

ros

Maximum coupling between objects: 5 on waypoints_navi.hpp

CBO is greater than 5, WaypointsGoalNode, 14

metrics nasa-satc ros coupling

Minimum lines of comments: 20% on main.cpp

Comment ratio is below 20%

metrics nasa-satc his uai ros comments comment-ratio

Figure 3.3.: Rule violations pop-up with one active filter tag, *ros*.

For instance, if we analyse the packages shown in Table 3.2, we can conclude that, in general, the quality models that report more violations are those with higher requirements, such as NASA SATC. This quality model have reported 117 violations, while a less restrictive quality model, like ROS, have reported 26 violations.

Table 3.2.: Number of lines of code, rules violated in total and per quality model for a set of Kobuki's packages.

Package	Source Lines of Code	Total of Non-Compliance Rules	Violations per Quality Model				
			NASA	HIS	UNAK	KTH	ROS
kobuki_driver	3314	75	53	9	31	3	6
kobuki_ftdi	836	35	20	8	11	4	4
kobuki_node	1248	21	17	4	5	0	7
kobuki_dock_drive	543	11	6	1	5	0	0
kobuki_random_walker	390	9	5	3	1	3	4
kobuki_auto_docking	506	6	4	0	2	0	2
kobuki_bumper2pc	122	5	4	1	1	0	1
kobuki_controller_tutorial	109	5	3	0	2	0	1
kobuki_keyop	389	5	3	0	2	0	1
kobuki_safety_controller	289	4	2	0	2	0	0

3.3.2. Analysis of Turtlebot

Turtlebot consists of 17 packages, where 3 of them are metapackages. These packages contain around 101 C++ source code files, with more than 15.000 lines of code and has 154 rules violated. For instance, in this particular case a violation is found on an average of 97 lines of code. Figure 3.4, just like Figure 3.1, illustrates the diagram built from the graphical component, but this time for the Turtlebot's packages, as well as their dependencies.

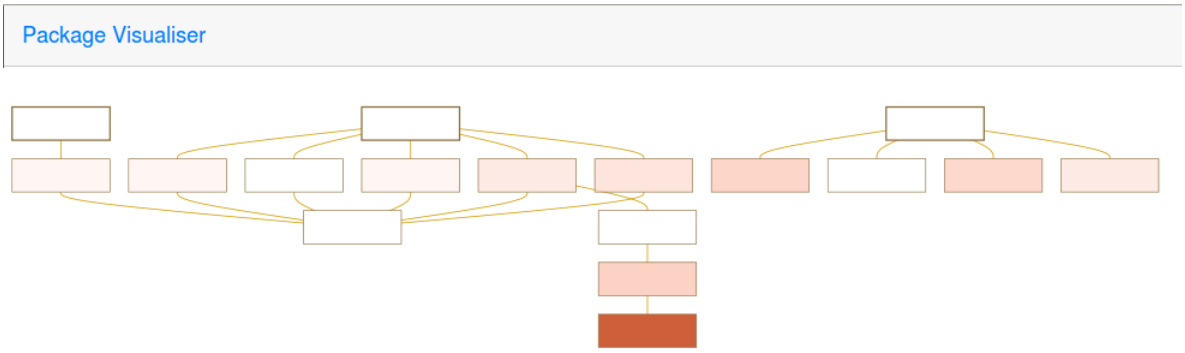


Figure 3.4.: Preview of the Turtlebot's packages and their dependencies.

Table 3.3 presents the total number of rules violated, as well as the number of rules violated per quality model for a set of Turtlebot's packages.

Table 3.3.: Number of lines of code, rules violated in total and per quality model for a set of Turtlebot's packages.

Package	Lines of Source Code	Total of Non-Compliance Rules	Violations per Quality Model				
			NASA	HIS	UNAK	KTH	ROS
turtlebot_arm_ikfast_plugin	5377	16	11	7	2	4	7
turtlebot_arm_block_manipulation	1076	15	12	3	0	3	9
turtlebot_actions	420	10	12	6	1	0	5
turtlebot_panorama	442	8	5	1	2	1	2
turtlebot_arm_kinect_calibration	411	8	6	1	3	0	4
turtlebot_follower	204	3	3	1	0	0	1
turtlebot_teleop	71	3	2	0	1	0	1
turtlebot_navigation	63	3	2	0	1	0	1

For instance, if we analyse the packages shown in previous table (Table 3.3), we can conclude, just like we did from the Table 3.2 analysis, that, in general, the quality models that report more violations are those with higher requirements, such as NASA SATC. This quality model have reported 53 violations, while a less restrictive quality model, like ROS, have reported 30 violations.

3.3.3. Analysis of Other Robots

Besides these two, other robots were studied and analyzed, such as, *abb*, *cob*, *grizzly*, *husky*, *motoman*, *nao*, *pr2*, *shadow* and *universal*. These robots together consist of 225 packages. These packages contain more than 257.532 lines of code and have a total of 4.234 rules violated. Table 3.4 presents the total number of rules violated, lines of source code.

Appendix D presents, in its entirety, all the robot's packages, as well as the rules violated by each one of them.

As previously stated, the number of violations found in a package is directly dependent on the chosen quality model. For instance, if we choose a package, such as *Kobuki_driver*, and a more restrictive quality model (e.g. 53 violations are found for the threshold proposed by NASA SATC) the number of violations will be strictly greater than a quality model with low a restriction level (e.g. 3 violations are found for the threshold proposed by KTH). After analysing all the robots it is possible to state that the most violations are found in the NASA SATC quality model (see Table 3.2 and 3.3), and the rule that is violated more often, in all the analysed robots, is the *Maximum lines of comments: 30%* with an average of 32.3%, according to the other rules

Table 3.4.: Number of lines of code, packages, rule violations and the percentage of violations per line of code for all the other analysed robots.

Robot	Lines of Source Code	Number of Packages	Non-Compliance Rules	Violations per Line of Code (%)
Cob	105322	67	1264	1.20%
Shadow	46211	50	1008	2.18%
Pr2	55461	66	974	1.76%
Nao	20818	8	545	2.62%
Abb	10135	7	167	1.65%
Motoman	12606	7	157	1.25%
Husky	4078	8	59	1.45%
Grizzly	1495	10	44	2.94%
Universal	1406	3	16	1.14%

presented in Table 3.5.

Table 3.5.: Number and percentage of each violated rule.

Rule	Rule Violations	
	Numerical	Percentage (%)
Minimum available methods per class: 1	0	0
Maximum available methods per class: 20	21	0.3
Maximum executable lines: 50	0	0
Maximum executable lines: 70	0	0
Maximum immediate children: 10	2	0.03
Minimum weighted methods per class: 1	0	0
Maximum weighted methods per class: 50	7	0.1
Maximum weighted methods per class: 100	4	0.06
Maximum coupling between objects: 5	576	8.9
Minimum lines of comments: 20%	744	11.6
Maximum lines of comments: 30%	2094	32.6
Maximum lines of comments: 40%	1883	29.3
Maximum function calls: 7	0	0
Maximum function calls: 10	0	0
Maximum nesting of control structures: 4	0	0
Maximum nesting of control structures: 5	0	0
Deepest level of inheritance: 5	0	0
Maximum cyclomatic complexity: 10	749	11.6
Maximum cyclomatic complexity: 15	333	5.1
Maximum estimated static paths: 80	0	0
Maximum estimated static paths: 200	0	0
Maximum estimated static paths: 250	0	0

4. Conclusion

Robotics has been an area with a growing evolution, and nowadays the robots are capable of performing more arduous and complex tasks. This growth has become especially notorious since a major part of the robotic code became public, allowing new people to study and develop software to robotics. However, not all the developed robotic source code is public, because some institutions have the entire interest of keeping their developed code private. But, remarkably, nowadays there are more and more communities that are attached to open source middleware frameworks used to develop robotic software. An example of this is the Robotic Operating System, a growing open source middleware framework that counts with the support of a big and active community. ROS also has a policy of keeping all the developed code public, and this provides a unique opportunity to study a large body of robotic software.

The increasing growth in the area of robotics, as well as the increasing use of robots in performing increasingly complex and dangerous tasks, turns the software safety and reliability a real concern. Thus, it is then necessary to ensure that the software used in the robots is rated with high quality. Unfortunately, the existing middleware frameworks for robotics do not offer any kind of system to assess the quality of the developed software. Therefore, this research project offers a system which enables the assessment of ROS projects, according to a set of quality metrics and rules. This final chapter is dedicated to some conclusions about the work done, as well as, one possible improvements to the implemented system and future work.

In Chapter 2, we explore the concept of quality in software engineering, as well as the importance that the code quality has in the development of software for robots. Although, we can analyse the internal quality, external quality and quality in use of a software product in this research project we focus on the internal quality, because it is easier to measure the internal attributes of the code rather than measuring the other attributes related to the external quality or quality in use. Both internal and external quality are composed by a set of characteristics, but to gauge something about the quality of the software it is necessary to focus on a more particular characteristic of the internal quality, the maintainability. The study of maintainability, in turn, is accomplished by measuring a series of internal quality metrics.

Software industry uses metrics with the objective of verifying the efficiency, progress, quality and performance of the software. They also provide information about the software status, which is helpful to obtain an objective assessment of the software. Currently, there are a few different types of quality metrics, however the quality metrics of interest for this project are the ones proposed by the ROS quality model, since it is the targeted system of study in this project. These metrics can be seen on Table 2.1, Section 2.1.2. Although there is already a quality model for the ROS, this was not only used and study in this project, because it was created from others four quality models with different requirement levels and thresholds. Therefore, the threshold values proposed for ROS should be used as a guideline until the ROS community reaches an agreement for a "official" threshold. In this regard, the quality models proposed by NASA SATC, HIS, UNAK and KTH are also studied and used to measure quality metrics.

With the study of various quality models and quality metrics, it became pertinent the creation of a system to measure the quality of ROS systems. In turn, this system would be implemented in another static analysis tool, thereby another study was conducted but this time on a set of tools of open source and commercial static analysis tools, with the interest of finding one that could be upgraded through the inclusion of plugins. Remarkably, a open-source tool that is already tailored for ROS was found, the ROS Static Analysis Tool. This is a flexible and extendable tool to gauge the code quality of ROS projects, more specifically for coding standards. Apart from ROS Static Analysis tool, other tools were studied with the ability to analyse the source code, allowing the collection and measurement of the highest number of metrics of interest for this project. Apart from the ROS Static Analysis Tool, other tools were studied with the ability to analyse the source code, allowing the collection and measurement of the highest number of metrics of interest. From all the other studied tools only two of them were chosen, namely the CCCC and OClint, because they are the ones that best fit the needs of the project. These tools were encapsulated as plugins and they are a part of a toolset, which is, in turn, incorporated into the ROS Static Analysis tool.

The discussion about the choice of these tools, as well as the presentation of the architecture and functioning of the toolset that is, in turn, introduced in ROS Static Analysis Tool was carried out on Chapter 3. In that same chapter, it was presented in greater detail the ROS Static Analysis Tool, emphasizing its architecture, functioning and the visual component. Providing the necessary information for the creation and integration of new plugins in the tool. The toolset, named Rule Violation Finder, is an example of a developed plugin which was incorporated in the tool. The Rule Violation Finder is then used to evaluate and measure the code quality of multiple robots, which are part of the ROS corpus. The non-compliance rules (i.e. metrics marked with a threshold value) found after each robot analysis can be visualized, through the graphic component of the ROS Static Analysis tool. The tool also offer a tagging system, allowing us to filter the displayed violated rules for a set of quality models and/or metric types. This tool, already incorporated with the toolset, is available on GitHub[1].

Finally, and still in this chapter, a data analysis was conducted where the developed system has been used to obtain the metric measurements of several robots, such as: *Abb*, *Cob*, *Grizzly*, *Husky*, *Kobuki*, *Motoman*, *Nao*, *Pr2*, *Shadow*, *Turtlebot* and *Universal*. With the aid of the graphic component, it was possible to retrieve the analysis results from each robot, and these results can be found on Section 3.3 and Appendix D.

4.1. Future Work

Given the low time frame of the current project and the lack of open source static analysis tools, as stated before, it was not possible to measure all the quality metrics proposed by ROS. Thus, one of the opportunities for future work would be the improvement of the metric measurement system (i.e. Violation Rule Finder), through the addition of new tools encapsulated as plugins, providing the measurement of the missing metrics, or even others if necessary. Another opportunity, also related to the toolset, would go through the improvement of the existing plugins, in order to turn the measurement of the metrics more efficient.

As mentioned previously, the ROS quality model was created from the other four studied quality models (NASA SATC, HIS, UNAK, KTH), and its threshold values should be used as guidelines. Thereby, study whether the proposed quality model currently is tailored for ROS, and propose changes if necessary, may also be considered an opportunity for future work.

Through the analysis conducted to multiple robots (see Section 3.3) it was possible to detect the existence of some errors. However, the results obtained do not allow us to take any concrete information about the quality of the robotic software. In this regard, it would be interesting to:

- Study concretely the impact that the violated rules cause to robots;

- Study ways to reduce the violations found in robotic software. One possible solution would be to introduce better programming practices on the robotic communities.

This would entail a more thorough study of the source code implemented in the robots in question.

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

One last opportunity for future work would be to assess an even larger set of robots or ROS projects, in order to classify in full, the quality of the ROS corpus.

In summary, this research project presents reasons which demonstrate the importance of building robots with high quality software, as well as the methods to assess it. This project also delivers an overview of the ROS framework and a review of the available open source static analysis tools for C++ source code. The C++ language is one of the main libraries of ROS, as well as the language of interest for this project. Nowadays, there are plenty of open source tools capable of performing a static analysis to C ++ source code, however many of these tools are not automable neither capable of measuring all the required metrics, and some of them are quite similar since they measure the same type of quality metrics. The importance of ensuring that robots are programmed with high quality software served as motivation in the development of this project. Thus, it was created a system, in a form of a plugin to an existing tool already tailored for ROS, capable of assessing the quality of robots developed from ROS. The assessement of these robots is consider a case study, and both, the developed system and the case study can be seen as a contribution to the ROS community.

Finally, we present several opportunities for future work, which includes the improvement of the toolset developed in this project, modification the ROS quality model, study a way to reduce the number of existent violations, as well as the impact that it has on robots, and assessing the quality of other ROS projects. With this project we intend to highlight the importance of code quality in building high quality software, as well as the growing need to measure and improving it, especially for the software developed from ROS.

Bibliography

Technical report.

Jerzy A. Barchanski. *Mobile Robots - Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training*. InTech, 2011. ISBN 978-953-307-842-7.

N. Bevan and M. Azuma. Quality in Use: Incorporating Human Factors into the Software Engineering Lifecycle. pages 169+. IEEE Computer Society, 1997. ISBN 0-8186-7837-2. URL http://portal.acm.org/citation.cfm?id=854938.

Andrew Binstock. Integration watch: Using metrics effectively, 2010. URL http://sdtimes.com/integration-watch-using-metrics-effectively/. Accessed: 2015-03-23.

Andy Brooks. Metrics overview, 2008. URL http://staff.unak.is/andy/StaticAnalysis0809/metrics/overview.html. Accessed: 2015-01-17.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. ISSN 0098-5589. doi: 10.1109/32.295895. URL http://dx.doi.org/10.1109/32.295895.

Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197–211, November 1991. ISSN 0362-1340. doi: 10.1145/118014.117970. URL http://doi.acm.org/10.1145/118014.117970.

Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994. ISSN 0018-9162. doi: 10.1109/2.303623. URL http://dx.doi.org/10.1109/2.303623.

Ken Conley. Nodes, 2012. URL http://wiki.ros.org/Nodes. Accessed: 2015-03-02.

Richard Torkar Aleš Ži̇vkovič Danijel Radjenović, Marjan Heričk. Information and software technology, 2013. URL http://romisatriawahono.net/lecture/rm/survey/software%20engineering/Software%20Fault%20Defect%20Prediction/Radjenovic%20-%20Software%20fault%20prediction%20metrics%20-%202013.pdf. Accessed 2015-02-20.

Ayssam Elkady and Tarek Sobh. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012:15, 2012. URL http://downloads.hindawi.com/journals/jr/2012/959013.pdf.

David A. Garvin. *Managing quality*. Free Press [u.a.], New York, NY, 1988. ISBN 0029113806. URL http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+025783971&sourceid=fbw_bibsonomy.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, QUATIC '07, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2948-8. doi: 10.1109/QUATIC.2007.7. URL http://dx.doi.org/10.1109/QUATIC.2007.7.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical report, IEEE, 1990. URL http://dx.doi.org/10.1109/ieeestd.1990.101064.

ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

J. M. Juran. *Juran's Quality Control Handbook*. Mcgraw-Hill (Tx), 4th edition, 1974. ISBN 0070331766. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0070331766.

Barbara Kitchenham and Shari L. Pfleeger. Software quality: The elusive target. *IEEE Softw.*, 13(1):12–21, January 1996. ISSN 0740-7459. doi: 10.1109/52.476281. URL http://dx.doi.org/10.1109/52.476281.

Adam Kolawa. When, why, and how: Code analysis, 2010. URL http://www.codeproject.com/Articles/28440/When-Why-and-How-Code-Analysis. Accessed: 2015-05-05.

Armin Krusko. Complexity Analysis of Real Time Software – Using Software Complexity Metrics to Improve the Quality of Real Time Software. Master's thesis, KTH Royal Institute of Technology, 2004. URL https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2004/rapporter04/krusko_armin_04032.pdf.

Helmar Kuder. HIS Source Code Metrics. *HIS Source Code Metrics*, page 8, 2008.

Johannes Kuehn. code_quality, 2013. URL http://wiki.ros.org/code_quality. Accessed: 2015-01-17.

Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. URL http://dl.acm.org/citation.cfm?id=800253.807712.

Microsoft. Code Metrics Values, 2015. URL https://msdn.microsoft.com/en-us/library/bb385914.aspx. Accessed: 2015-06-15.

NASA. *SOFTWARE ASSURANCE GUIDEBOOK, NASA-GB-A201*. NASA, 1989. URL http://www.hq.nasa.gov/office/codeq/doctree/nasa_gb_a301.pdf.

Brian A. Nejmeh. Npath: A measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188–200, 1988. doi: 10.1145/42372.42379. URL http://doi.acm.org/10.1145/42372.42379.

Paul Omar and J. Hagemeister. Metrics for assessing a software system's maintainability. *IEEE*, pages 337–344, 1992.

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.

Ryuichi Saito. Oclint, n.d. URL http://oclint.org/. Accessed: 2015-07-01.

André Santos. Applying Coding Standards to the Robot Operating System. Master's thesis, University of Minho, 2015.

W. A. Shewart. *Economic control of Quality of Manufactured Product*. Van Nostrand Reinhold Co., New York, 1931.

Software Assurance Technology Center. Software Metrics Research and Development, 2000a. URL http://web.archive.org/web/20000303021415/http://satc.gsfc.nasa.gov/metrics/index.html. Accessed: 2015-01-17.

Software Assurance Technology Center. SATC Historical OO Metrics Database, 2000b. URL http://web.archive.org/web/20080916010911/http://satc.gsfc.nasa.gov/metrics/codemetrics/index.html. Accessed: 2015-01-17.

Dirk Thomas. ROS distribution files, 2013. URL http://www.ros.org/reps/rep-0141.html. Accessed: 2015-08-01.

Dirk Thomas. ROS Introduction, 2014. URL http://wiki.ros.org/ROS/Introduction. Accessed: 2015-02-25.

Dirk Thomas Tully Foote and Paul Mathieu. ROS distribution files, 2013. URL http://www.ros.org/reps/rep-0137.html. Accessed: 2015-08-01.

Joost Visser. SIG/TUViT Evaluation Criteria Trusted Product Maintainability: Guidance for producers - Version 6.1. *SIG/TUViT Evaluation Criteria Trusted Product Maintainability: Guidance for producers*, page 8, 2014.

Sarah Vonnegut. Open Source vs. Commercial Tools: Static Code Analysis Showdown, 2015. URL https://www.checkmarx.com/2015/03/17/open-source-vs-commercial-tools-static-code-analysis-showdown-2/. Accessed: 2015-07-26.

Stefan Wagner. *Software Product Quality Control*. Springer, 2013. ISBN 978-3-642-38570-4. doi: 10.1007/978-3-642-38571-1. URL http://dx.doi.org/10.1007/978-3-642-38571-1.

David A. Wheeler. More than a gigabuck: Estimating gnu/linux's size. 2001. URL http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.

A. Internal Quality Metrics

A.1. Quality Models

A.1.1. NASA SATC

Table A.1.: NASA SATC File-metrics

Comment to code ratio	
MIN	MAX
0.2	0.3

Table A.2.: NASA SATC Function-metrics

Cyclomatic complexity		Number of executable lines		Number of function calls		Maximum nesting of control structures		Estimated static path count	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
1	10	1	50	–	–	–	–	–	–

Table A.3.: NASA SATC Class-metrics

Coupling between objects		Number of immediate children		Weighted methods per class		Deepest level of inheritance		Number of methods available	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
–	5	–	–	–	100	–	5	1	20

A.1.2. ROS

Table A.4.: ROS File-metrics

Comment to code ratio	
MIN	MAX
0.2	–

Table A.5.: ROS Function-metrics

Cyclomatic complexity		Number of executable lines		Number of function calls		Maximum nesting of control structures		Estimated static path count	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
1	15	1	70	1	10	–	5	1	250

Table A.6.: ROS Class-metrics

Coupling between objects		Number of immediate children		Weighted methods per class		Deepest level of inheritance		Number of methods available	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
0	5	0	10	1	100	–	5	1	20

A.1.3. KTH

Table A.7.: KTH Function-metrics

Cyclomatic complexity		Number of executable lines		Number of function calls		Maximum nesting of control structures		Estimated static path count	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
1	15	1	70	1	10	–	5	1	250

A.1.4. HIS

Table A.8.: HIS File-metrics

Comment to code ratio	
MIN	MAX
0.2	–

Table A.9.: HIS Function-metrics

Cyclomatic complexity		Number of executable lines		Number of function calls		Maximum nesting of control structures		Estimated static path count	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
1	10	1	50	1	7	–	4	1	80

A.1.5. UNAK

Table A.10.: University of Akureyri in Iceland File-metrics

Comment to code ratio	
MIN	MAX
0.2	0.4

Table A.11.: University of Akureyri in Iceland Class-metrics

Coupling between objects		Number of immediate children		Weighted methods per class		Deepest level of inheritance		Number of methods available	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
–	–	–	10	1	50	–	5	–	–

A.1.6. SIG

Table A.12.: SIG's metric values (1 of 2)

	Volume	Complexity per unit	Duplication	Unit size	Unit interfacing
4 $\star$	229,000 LOC	$>$5 CC ($<$25.4%) $>$10 CC ($<$9.6%) $>$25 CC ($<$1.1%)	$<$4.4%	$>$20 LOC ($<$35.6%) $>$50 LOC ($<$9.7%) $>$100 LOC ($<$2.2%)	$\geq$3($<$14.2%) $\geq$5($<$2.7%) $\geq$7($<$0.7%)

Table A.13.: SIG's metric values (2 of 2)

	Module coupling	Component balance	Component independence
4 $\star$	$>$10 Dependencies ($<$23.1%) $>$20 Dependencies ($<$14.8%) $>$50 Dependencies ($<$7.2%)	# of components $\sim$9 Gini coefficient $<$0.35	% code in modules with dependence $<$14.3%

A.2. NPath Expressions

Table A.14.: NPath Expressions - (Nejmeh, 1988)

Structure	**Expression**
if	NPATH(expr) + NPATH(statements in **then**) + NPATH(statements in **else**) + 1
while	NPATH(expr) + NPATH(statements in **while**) + 1
do while	NPATH(expr) + NPATH(statements in **do**) + 1
for	NPATH(expr1) + NPATH(expr2) + NPATH(expr3) + NPATH(statements in **for**) + 1
switch	NPATH(expr) + $\sum_{i=1}^{n}$ (NPATH(statements in **case**$_i$)), *where n = # cases + 1(default)*
Expressions	Number of **&&** and || in expression
Other statements	1
Functions	$\prod_{i=1}^{n}$ NPATH(**statement_i**)

B. Declared Rules

```
%YAML 1.1
# Rules file.
---
-
    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%"
    tags:
        - metrics
        - nasa-satc
        - comments
        - comment-ratio
-
    id:             3
    name:           MAX_COM_RATIO
    scope:          file
    description:    "Maximum lines of comments: 40%"
    tags:
        - metrics
        - uai
        - comments
        - comment-ratio
-
    id:             5
    name:           MAX_CC
    scope:          function
    description:    "Maximum cyclomatic complexity: 10"
    tags:
        - metrics
        - nasa-satc
        - his
        - hicpp
        - cyclomatic-complexity
-
    id:             6
    name:           MAX_CC
    scope:          function
    description:    "Maximum cyclomatic complexity: 15"
    tags:
        - metrics
        - kth
        - ros
        - cyclomatic-complexity
-
    id:             8
    name:           MAX_EXE_LINES
    scope:          function
    description:    "Maximum executable lines: 50"
    tags:
        - metrics
        - nasa-satc
        - his
        - executable-lines
-
    id:             9
    name:           MAX_EXE_LINES
    scope:          function
    description:    "Maximum executable lines: 70"
    tags:
        - metrics
        - kth
        - ros
        - executable-lines
-
    id:             11
    name:           MAX_FUN_CALLS
    scope:          function
    description:    "Maximum function calls: 7"
    tags:
        - metrics
        - his
        - function-calls
-
    id:             12
    name:           MAX_FUN_CALLS
    scope:          function
    description:    "Maximum function calls: 10"
    tags:
        - metrics
        - kth
        - ros
        - function-calls
-
    id:             13
    name:           MAX_NEST_CONTROL
    scope:          function
    description:    "Maximum nesting of control structures: 4"
    tags:
        - metrics
        - his
        - nesting-control
-
    id:             14
    name:           MAX_NEST_CONTROL
    scope:          function
    description:    "Maximum nesting of control structures: 5"
    tags:
        - metrics
        - kth
        - ros
        - nesting-control
```

```
-
    id:             16
    name:           MAX_STATIC_PATH
    scope:          function
    description:    "Maximum estimated static paths: 80"
    tags:
        - metrics
        - his
        - static-path
-
    id:             17
    name:           MAX_STATIC_PATH
    scope:          function
    description:    "Maximum estimated static paths: 250"
    tags:
        - metrics
        - kth
        - ros
        - static-path
-
    id:             18
    name:           MAX_COUPLING
    scope:          class
    description:    "Maximum coupling between objects: 5"
    tags:
        - metrics
        - nasa-satc
        - ros
        - coupling
-
    id:             19
    name:           MAX_CHILDREN
    scope:          class
    description:    "Maximum immediate children: 10"
    tags:
        - metrics
        - uai
        - ros
        - immediate-children
-
    id:             20
    name:           MIN_WEIGHT_METHODS
    scope:          class
    description:    "Minimum weighted methods per class: 1"
    tags:
        - metrics
        - uai
        - ros
        - weighted-methods
-
    id:             21
    name:           MAX_WEIGHT_METHODS
    scope:          class
    description:    "Maximum weighted methods per class: 50"
    tags:
        - metrics
        - uai
        - weighted-methods
-
    id:             22
    name:           MAX_WEIGHT_METHODS
    scope:          class
    description:    "Maximum weighted methods per class: 100"
    tags:
        - metrics
        - nasa-satc
        - ros
        - weighted-methods
-
    id:             23
    name:           MAX_INHERIT_LEVEL
    scope:          class
    description:    "Deepest level of inheritance: 5"
    tags:
        - metrics
        - nasa-satc
        - uai
        - ros
        - inheritance-level
-
    id:             24
    name:           MIN_AVAILABLE_METHODS
    scope:          class
    description:    "Minimum available methods per class: 1"
    tags:
        - metrics
        - nasa-satc
        - ros
        - available-methods
-
    id:             25
    name:           MAX_AVAILABLE_METHODS
    scope:          class
    description:    "Maximum available methods per class: 20"
    tags:
        - metrics
        - nasa-satc
        - ros
        - available-methods
```

Listing B.1: Rule declarations for the quality models.

C. Tool Reports

C.1. CCCC Tool Report

```
<!-- Detailed report on module Rb2Vw -->
<CCCC_Project>
  <module_summary>
  <lines_of_code value="87" level="0"/>
  <lines_of_code_per_member_function value="******" level="0"/>
  <McCabes_cyclomatic_complexity value="13" level="0"/>
  <McCabes_cyclomatic_complexity_per_member_function value="******" level="2
      "/>
  <lines_of_code value="1" level="0"/>
  <lines_of_code_per_member_function value="********" level="2"/>
  <lines_of_code_per_line_of_comment value="87.000" level="2"/>
  <McCabes_cyclomatic_complexity_per_line_of_comment value="13.000" level="2
      "/>
  <weighted_methods_per_class_unity value="5" level="0"/>
  <weighted_methods_per_class_visibility value="5" level="0"/>
  <depth_of_inheritance_tree value="0" level="0"/>
  <number_of_children value="0" level="0"/>
  <coupling_between_objects value="0" level="0"/>
  <IF4 value="0" level="0"/>
  <IF4_per_member_function value="********" level="0"/>
  <IF4_visible value="0" level="0"/>
  <IF4_visible_per_member_function value="********" level="0"/>
  <IF4_concrete value="0" level="0"/>
  <IF4_concrete_per_member_function value="********" level="0"/>
  </module_summary>
  <module_detail>
  <description>definition</description>
  <source_reference file="/home/miguel/ros/repos/kobuki_core/kobuki_driver/
      src/test/velocity_commands.cpp" line="38"/>
  <lines_of_code value="5" level="0"/>
  <McCabes_cyclomatic_complexity value="0" level="0"/>
  <lines_of_comment value="0" level="0"/>
  <lines_of_code_per_line_of_comment value="------" level="0"/>
  <McCabes_cyclomatic_complexity_per_line_of_comment value="------" level="0
     "/>
  </module_detail>
  ...
<CCCC_Project>
```

Listing C.1: Example of a XML report extracted from CCCC.

C.2. OClint Tool Report

```
<oclint version="0.8.1" url="http://oclint.org">
<datetime>2015-07-18T20:15:56Z</datetime>
  <summary>
  <property name="number of files">1</property>
  <property name="files with violations">1</property>
  <property name="number of priority 1 violations">0</property>
  <property name="number of priority 2 violations">8</property>
  <property name="number of priority 3 violations">58</property>
</summary>
  <violations>
    <violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
       test/velocity_commands.cpp" startline="44" startcolumn="95" endline="
       44" endcolumn="101" rule="long variable name" priority="3" message="
       Variable name with 23 characters is longer than the threshold of 20"/
       >
    <violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
       test/velocity_commands.cpp" startline="83" startcolumn="61" endline="
       83" endcolumn="76" rule="long variable name" priority="3" message="
       Variable name with 21 characters is longer than the threshold of 20"/
       >
    <violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
       test/velocity_commands.cpp" startline="170" startcolumn="3" endline="
       170" endcolumn="10" rule="long variable name" priority="3" message="
       Variable name with 25 characters is longer than the threshold of 20"/
       >
```

```
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="171" startcolumn="3" endline="
   171" endcolumn="10" rule="long variable name" priority="3" message="
   Variable name with 26 characters is longer than the threshold of 20"/
   >
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="20" startcolumn="1" endline="
   36" endcolumn="1" rule="high npath complexity" priority="2" message="
   NPath Complexity Number 12 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="42" startcolumn="5" endline="
   42" endcolumn="89" rule="high npath complexity" priority="2" message=
   "NPath Complexity Number 1 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="44" startcolumn="5" endline="
   53" endcolumn="5" rule="high npath complexity" priority="2" message="
   NPath Complexity Number 2 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="55" startcolumn="5" endline="
   81" endcolumn="5" rule="high npath complexity" priority="2" message="
   NPath Complexity Number 5 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="83" startcolumn="5" endline="
   87" endcolumn="5" rule="high npath complexity" priority="2" message="
   NPath Complexity Number 1 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="89" startcolumn="5" endline="
   133" endcolumn="5" rule="high npath complexity" priority="2" message=
   "NPath Complexity Number 21 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="137" startcolumn="1" endline="
   224" endcolumn="1" rule="high npath complexity" priority="2" message=
   "NPath Complexity Number 49 exceeds limit of 0"/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="137" startcolumn="10" endline=
   "137" endcolumn="14" rule="unused method parameter" priority="3"
   message="The parameter ’argc’ is unused."/>
<violation path="/home/miguel/ros/repos/kobuki_core/kobuki_driver/src/
   test/velocity_commands.cpp" startline="137" startcolumn="20" endline=
   "137" endcolumn="27" rule="unused method parameter" priority="3"
   message="The parameter ’argv’ is unused."/>
```

```
    ...
  </violations>
</oclint>
```

Listing C.2: Example of a XML report extracted from OClint.

D. Data Analysis

D.1. Turtlebot Package Analysis

Table D.1.: Number of rule violations found on Turtlebot's packages.

Turtlebot			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
pano_core	71	turtlebot_follower	3
pano_py	17	turtlebot_teleop	3
turtlebot_arm_ikfast_plugin	16	turtlebot_navigation	3
turtlebot_arm_block_manipulation	15	pano_ros	0
turtlebot_actions	10	turtlebot_arm_moveit_demos	0
turtlebot_panorama	8	turtlebot_bringup	0
turtlebot_arm_kinect_calibration	8	turtlebot_calibration	0
Total of **154** violations found			

D.2. Kobuki Package Analysis

Table D.2.: Number of rule violations found on Kobuki's packages.

Kobuki			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
ecl_eigen	830	ecl_formatters	11
ar_track_alvar	288	kobuki_dock_drive	11
ecl_devices	84	ecl_sigslots_lite	11
kobuki_driver	75	yocs_ar_pair_tracking	9
ecl_time	58	ecl_linear_algebra	9
ecl_command_line	56	ecl_exceptions	9
ecl_geometry	53	kobuki_random_walker	9
ecl_threads	37	yocs_waypoint_provider	9
kobuki_ftdi	35	ecl_filesystem	8
ecl_containers	34	ecl_mobile_robot	8
ecl_streams	31	yocs_diff_drive_pose_controller	7
ecl_time_lite	30	yocs_waypoints_navi	6
ecl_core_apps	28	kobuki_auto_docking	6
ecl_utilities	28	kobuki_keyop	5
kobuki_node	21	yocs_safety_controller	5
ecl_config	17	yocs_virtual_sensor	5
ecl_io	17	kobuki_bumper2pc	5
ecl_converters	15	kobuki_controller_tutorial	5
ecl_math	15	yocs_velocity_smoother	4
ecl_sigslots	15	kobuki_safety_controller	4
ecl_concepts	15	yocs_keyop	4
ecl_ipc	14	yocs_math_toolkit	4
ecl_type_traits	13	ecl_statistics	4
ecl_mpl	12	ecl_converters_lite	4
yocs_navigator	11	yocs_joyop	2
ecl_errors	11	yocs_controllers	2
yocs_cmd_vel_mux	11	kobuki_testsuite	0
yocs_ar_marker_tracking	11	yocs_ar_pair_approach	0
		yocs_localization_manager	0
Total of **1031** violations found			

D.3. Abb Package Analysis

Table D.3.: Number of rule violations found on Abb's packages.

Abb			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
simple_message	103	abb_irb2400_moveit_plugins	16
industrial_robot_client	42	abb_driver	6
Total of **167** violations found			

D.4. Grizzly Package Analysis

Table D.4.: Number of rule violations found on Grizzly's packages.

Grizzly			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
grizzly_motion	28	grizzly_teleop	1
roboteq_driver	10	grizzly_robot	0
grizzly_base	3	roboteq_diagnostics	0
grizzly_msgs	2		
Total of **44** violations found			

D.5. Husky Package Analysis

Table D.5.: Number of rule violations found on Husky's packages.

Husky			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
husky_base	59	husky_robot	0
husky_control	0	ur_driver	0
husky_bringup	0		
Total of **59** violations found			

D.6. Motoman Package Analysis

Table D.6.: Number of rule violations found on Motoman's packages.

Motoman			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
simple_message	103	motoman	0
industrial_robot_client	42	motoman_driver	0
industrial_utils	12		
Total of **157** violations found			

D.7. Nao Package Analysis

Table D.7.: Number of rule violations found on Nao's packages.

Nao			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
orocos_kdl	325	nao_path_follower	6
naoqi_driver	207	orocos_kinematics_dynamics	0
nao_teleop	7	nao_extras	0
Total of **545** violations found			

D.8. Universal Package Analysis

Table D.8.: Number of rule violations found on Universal's packages.

Universal			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
ur_ kinematics	16	universal_ robot	0
ur_ driver	0		
Total of **16** violations found			

D.9. Shadow Package Analysis

Table D.9.: Number of rule violations found on Shadow's packages.

Shadow			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
orocos_kdl	325	ros_ethercat_loop	7
sr_mechanism_controllers	90	cyberglove_trajectory	5
sr_robot_lib	78	sr_ronex_hardware_interface	4
sr_hand	67	sr_ronex_utilities	4
ros_ethercat_eml	56	sr_gazebo_sim	3
sr_edc_ethercat_drivers	40	sr_example	2
sr_ronex_drivers	39	sr_ronex_external_protocol	2
sr_self_test	36	sr_hardware_interface	1
sr_ronex_transmissions	27	sr_core	0
hand_kinematics	25	logitech_r400	0
sr_tactile_sensors	22	shuttle_xpress	0
sr_ronex_controllers	22	orocos_kinematics_dynamics	0
sr_mechanism_model	20	shadow_robot_ethercat	0
ros_ethercat_hardware	19	sr_edc_controller_configuration	0
cyberglove	19	ros_ethercat	0
sr_remappers	17	sr_tools	0
sr_movements	17	sr_grasp_fast_planner	0
sr_utilities	17	sr_interface	0
kdl_coupling	16	sr_robot_commander	0
ros_ethercat_model	15	sr_grasp	0
sr_ronex_examples	13	sr_ronex	0
Total of **1008** violations found			

D.10. Pr2 Package Analysis

Table D.10.: Number of rule violations found on Pr2's packages.

Pr2			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
orocos_kdl	325	pr2_tilt_laser_interface	3
ethercat_hardware	98	pr2_controller_interface	3
pr2_mechanism_controllers	61	pr2_run_stop_auto_restart	2
pr2_mechanism_model	54	pr2_gripper_action	1
robot_mechanism_controllers	48	single_joint_position_action	1
pr2_arm_kinematics	36	pr2_computer_monitor	1
pr2_navigation_self_filter	33	pr2_ethercat	1
pr2_gripper_sensor_action	29	pr2_tuckarm	0
pr2_calibration_controllers	29	joint_trajectory_action_tools	0
pr2_gripper_sensor_action	29	pr2_common_actions	0
pr2_controller_manager	27	pr2_tuck_arms_action	0
semantic_point_annotator	27	orocos_kinematics_dynamics	0
pr2_gripper_sensor_controller	26	pr2_app_manager	0
pr2_teleop_general	24	pr2_apps	0
pr2_power_board	23	pr2_mannequin_mode	0
app_manager	16	pr2_position_scripts	0
ethercat_trigger_controllers	16	pr2_move_base	0
ocean_battery_driver	16	pr2_navigation	0
pr2_arm_move_ik	12	fingertip_pressure	0
pr2_mechanism_diagnostics	12	pr2_ethercat_drivers	0
power_monitor	11	pr2_gripper_sensor	0
pr2_teleop	8	pr2_robot	0
joint_trajectory_generator	8	pr2_power_drivers	0
laser_tilt_controller_filter	6	pr2_camera_synchronizer	0
pr2_hardware_interface	5	pr2_delivery	0
joint_trajectory_action	4	pr2_kinematics	0
pr2_head_action	4	pr2_mechanism	0
pr2_navigation_perception	4	pr2_controllers	0
Total of **974** violations found			

D.11. Cob Package Analysis

Table D.11.: Number of rule violations found on Cob's packages.

Cob			
Package	Non-Compliance Rules	Package	Non-Compliance Rules
orocos_kdl	325	cob_pick_place_action	7
schunk_libm5api	112	cob_linear_nav	7
cob_twist_controller	108	cob_scan_unifier	6
cob_light	61	cob_control_mode_adapter	6
schunk_sdh	59	cob_voltage_control	4
cob_obstacle_distance	57	cob_base_velocity_smoother	4
schunk_powercube_chain	55	cob_cam3d_throttle	3
cob_camera_sensors	46	cob_lookat_action	3
cob_sick_s300	39	cob_object_detection_visualizer	2
cob_utilities	28	cob_sound	2
cob_cartesian_controller	27	cob_driver	0
cob_kinematics	25	cob_mimic	0
cob_relayboard	25	cob_command_gui	0
cob_canopen_motor	24	cob_command_tools	0
cob_vision_utils	24	cob_dashboard	0
cob_phidgets	24	cob_monitoring	0
cob_trajectory_controller	21	cob_script_server	0
cob_undercarriage_ctrl	9	cob_control	0
cob_generic_can	17	cob_grasp_generation	0
cob_base_drive_chain	17	cob_manipulation	0
cob_collision_velocity_filter	17	cob_moveit_interface	0
cob_frame_tracker	17	cob_tactiletools	0
cob_omni_drive_controller	16	cob_tray_monitor	0
cob_image_flip	12	cob_navigation	0
cob_interactive_teleop	12	cob_perception_common	0
cob_sick_lms1xx	10	orocos_kinematics_dynamics	0
cob_teleop	10	schunk_modular_robotics	0
cob_head_axis	9	schunk_sdhx	0
cob_footprint_observer	7	schunk_simulated_tactile_sensors	0
cob_model_identifier	7		
Total of **1264** violations found			