POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Weighted code coverage: a tool to combine code metrics and code coverage into new metrics

Supervisors

Prof. Luca ARDITO

Prof. Maurizio MORISIO

Dr. Michele VALSESIA

Candidate

Giovanni TANGREDI

October 2022

Abstract

***Context*:** The usage of Software metrics is well consolidated in software development. Code metrics are software metrics that define methods to measure source code properties. Using these measures, programmers can have a picture of the status of a source code, identify potentially problematic parts, and improve the code during its maintenance. Code Coverage metrics are used to measure the percentage of code covered by a test suite. Knowing the coverage value, a developer can improve the code by adding tests to parts of code not covered since these parts may contain bugs or unwanted changes. Code metrics and coverage are the foundations for creating the metrics proposed in this thesis, which aim to combine these two aspects.

***Objectives*:** This thesis has three main objectives. The first one describes an open-source software developed by Mozilla, called *Rust Code Analysis*, and the series of contributions done to this project. The second one is to describe *Weighted Code Coverage*, the tool developed during this thesis, containing the implementation of the new metrics mentioned above. The third one is to study the added value of these new metrics and show how a developer might use them.

***Method*:** The work done during this thesis can be divided into two main parts. In the first part, we have researched all metrics that are then being implemented into *Weighted Code Coverage*. In addition, we added the minimum and maximum for some metrics already implemented in *Rust Code Analysis*. The second part mainly focused on implementing the researched metrics and developing *Weighted Code Coverage* according to good coding practice. During this stage, we have also improved the tool in performance and memory usage and performed a series of analyses on different repositories. These analyses have been subdivided into two categories: spatial and time analyses. The spatial analysis focuses on understanding the behaviors of the tool on codebases of different sizes. Instead, time analysis observes how the results obtained by the program change for different versions of the same codebase.

***Conclusions*:** The work produced in this thesis consisted of implementing weighted code coverage, a tool that implements four new metrics types. The tool has been written in Rust programming language, and it computes its metrics starting from a series of source codes also written in Rust. Since the project is open source, future contributors can extend Weighted Code Coverage, adding metrics for other programming languages. At last, we have shown the behavior of these metrics in actively maintained repositories, specifying which could be their potential uses.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code metrics and code coverage are becoming well consolidated in software development. These two development aspects are currently separated and rarely interact with each other, despite being very important during the maintenance of software. In this thesis, a new set of metrics have been introduced to combine Code metrics and Code coverage and understand how they interact with each other. Starting from this considerations we created *Weighted Code Coverage*, a tool that can compute four new metrics: **WCC Plain**, **WCC Quantized**, **CRAP**, **SkunkScore**. Rust is the programming language used to write *Weighted Code Coverage*. We chose Rust because is a new programming language that takes inspiration from dynamically-typed languages like JavaScript. Rust's most strong points are speed, memory safety, and parallelism.
Weighted Code Coverage uses two main library as its backbone: *Rust Code Analysis* and *Grcov*. *Rust Code Analysis* is used to compute the code metrics needed by the tool. It handles Static code metrics metrics computed at compile time by analyzing the source code of a program. *Grcov* with is mostly used to generate the JSON parsed to obtain all coverage information about a repository. *Grcov* was selected because is optimized to work with codebases written in Rust. Both libraries are maintained by Mozilla.
We divided this thesis into two main parts:

- The first part explains the main features of *Rust Code Analysis* like the implementation of metrics utilized in *Weighted Code Coverage* and how the parsing of an Abstract Syntax Tree is performed. We also describe all the contribution done to the project.

- The second part is focused on the research and implementation of new kinds of metrics that combine together Code Complexity and Code Coverage in *Weighted Code Coverage*. We describe the tool development starting from

the first prototype until the last released version.Also, we present the result obtained by the tool by analyzing various repositories of different sizes.

We separated the arguments into chapters with the following structure:

- In chapter 2 we will describe the State of Art in static and dynamic metrics. We will focus on the concepts beneath code coverage and code metrics.
- In chapter 3 we will introduce the main characteristic of the Rust language and introduce the main features of *Rust Code Analysis* and how it works.
- In chapter 4 we will introduce *Weighted Code Coverage*. In particular, we will illustrate the third-party libraries that compose this tool in addition to describing how these new metrics have been implemented. We will also delineate the evolution of this software over time, from its sequential computing of the first version to the parallel one contained in the last released version.
- In chapter 5 we will present all the results obtained by *Weighted Code Coverage*. Two kinds of analysis were done: Spatial Analysis and Time Analysis. For Spatial analysis, we will study the behavior of the tool with codebases of different sizes. Time Analysis is based on analyzing the differences in the same repository over different versions. At last, we will present both the performance and the memory usage of the program.
- In chapter 6 we will draw the conclusions about the work that has been done. We will also present future opportunities for the development of the tool and research about covered topics.

Chapter 2

State of Art

The increased complexity of modern software applications also increases the difficulty of making the code reliable and maintainable. Code metrics are a type of measure that provide developers better insight into their code. By using different code metrics, developers can understand which functions, classes, and/or methods should be reworked or more thoroughly tested. There are two types of code metrics:

- **Static** metrics are obtained by analyzing the source code or the program at compile time. These types of metric are more simple to compute since all the information is already written and cannot change over time unless the source code is modified

- **Dynamic** metrics are computed at run-time or after a program is run. Since the source code can be run with a variety of different inputs, the result can also change depending on the input of the program. These types of metrics are often more complex than static ones.

This thesis will concentrate on static source metrics. One of the many pieces of information that static metrics give is the code complexity of a program. Code complexity is a way to quantize how much a piece of code is complicated and unwieldy for a developer to understand. Another important feature of code complexity is the possibility to measure the number and complexity of tests needed for correctly testing that code. Another important measure in modern software applications is Code Coverage. Code coverage is a software testing metric that determines the number of lines of code that are successfully covered by test suites. It is expressed in the percentage of covered code. Path coverage tests all paths in the code. It is equal to the number of paths covered by tests divided by the total number of paths in the code. It can be difficult to measure because a piece of code may have an unlimited or immeasurable number of paths.

2.1 Cyclomatic

Cyclomatic[1] complexity is a software metric used to indicate the complexity of a program. It was developed by Thomas J. McCabe, Sr. in 1976. The cyclomatic complexity of a section of code is the number of linearly independent paths within its control flow graph. The control flow graph of a program can have multiple paths depending on its number of conditional statement such as if, while and for loops. For instance, if the source code does not contain any control flow statements , the complexity would be 1, since there would be only a single path through the code. If the code had one single-condition, for example an if statement, there would be two paths through the code: one where the if statement evaluates to true and another one where it evaluates to false, so the overall complexity would be 2. Two nested single-condition if statements would produce a complexity of 3. From a mathematical point of view, the cyclomatic complexity of a program (or function), given its control-flow graph, is equal to:

$$M = E - N + 2$$

where:

- E = the number of edges of the graph.
- N = the number of nodes of the graph.

The cyclomatic complexity can also be applied to multiple programs (or functions). In that case the overall cyclomatic complexity is equal to the sum of all the complexities for each program.

$$M = \sum_{i=1}^{N} M_i$$

Tom McCabe introduces the following categorisation to interpret cyclomatic complexity for a single module:

- 1 - 10 Simple procedure, little risk
- 11 - 20 More complex, moderate risk
- 21 - 50 Complex, high risk
- $>$ 50 Untestable code, very high risk

Another application of cyclomatic complexity is in determining the number of test cases that are necessary to achieve sufficient coverage for a specific piece of code.

Cyclomatic complexity can be used to obtain these information about the code:

- The cyclomatic complexity is an upper bound for the number of test cases necessary to achieve a complete branch coverage.
- The cyclomatic complexity is a lower bound for the number of paths through a control-flow graph. Assuming that each test is a path, the number of cases needed to reach path coverage is equal to the number of paths that can actually be taken. However some paths may be impossible (or "infinite") so this number can sometimes be less than cyclomatic complexity.

The code below show an example of Cyclomatic complexity for a simple Java function:

```
1  int function() { //Complexity start at 1
2      int n1 = 72, n2 = 120, lcm;
3      lcm = (n1 > n2) ? n1 : n2;
4      while(true) { // while complexity +1
5        if( lcm % n1 == 0 && lcm % n2 == 0 ) {// if complexity +1
6          break;
7        }
8        ++lcm;
9        return lcm
10 }
```

Listing 2.1: Example of Cyclomatic Complexity computation in Java

2.2 Cognitive

Cognitive[2] complexity is a metric developed by G. Ann Campbell in 2018. Cognitive complexity has been formulated to address modern language structures and it measures the cognitive effort required to understand a block or the entire code of a program. Through cognitive complexity, a developer can measure how much a produced code is hard to understand for other programmers.

Cognitive complexity can be calculated using three simple rules:

1. Ignore structures that allow multiple statements to be shorthanded into one
2. Increment by one its value for each *break* in the linear flow of the code
3. Increment by one its value when flow-breaking structures, such as *If*, *While*, etc., are nested.

The complexity score can have four types of different increments:

- Nesting - number of nesting control flow structures one inside the other: *If* inside *whiles*, nested *ifs*, etc...

- Structural - number of control flow structures that are subject to a nesting increment, and that increase the nesting count
- Fundamental - number of statements not subject to a nesting increment
- Hybrid - number of control flow structures that are not subject to a nesting increment, but which increase the nesting count

Any type of increment increases the final score by one, but differentiating among types makes it easier to understand where nesting increments have and have not been applied. Now, the rules defined above will be analyzed more in detail.

2.2.1 Rule 1

Cognitive complexity incentives good coding practices, so it does not consider any construct or technique that makes code more readable. A perfect example of this rule is to used methods instead of conditional statements, like *isEmpty()* to check if a vector is empty instead of checking with an *if* that its length is equal to zero. Breaking code into methods allows you to condense multiple statements into a single, evocatively named call. In this case, the complexity does not increment for methods. Another example can be the Java *?* operator that can shorten the following piece of code:

```
1 //Bad practice version
2 MyObj myObj = null;
3 if (a != null) {
4     myObj = a.myObj;
5 }
6
7 //Shortened Version
8 MyObj myObj = a?.myObj;
```

Listing 2.2: Rule one example

The meaning of the bad practise version takes a moment to be processed by the brain, while the shortened version is immediately clear once the developer understand the use of the *?* operator in Java. When the *?* operator is used, the score does not increase. Instead, when the bad practice version is used, cognitive complexity is increased by one.

2.2.2 Rule 2

Cognitive Complexity assesses structural increments for:

- Loop structures: *for*, *while*, *do-while*, etc...

- Conditionals: ternary operators, *if*, *if-then-else*. In the case of if-then-else, the complexity only increases by one since the cognitive cost has already been paid by the if construct.
- Catches: A *catch* represents a kind of branch in a control flow as well as an if construct. Therefore, each *catch* clause results in a structural increment for Cognitive Complexity, so the score will increase by one independently by the number of handled exceptions. *try* and *finally* blocks are ignored since they do not change the flow of a program.
- Switches: A *switch* and all its *case* constructs increment by one the complexity score. This happens because a *switch* can be read more easily than a chain of if-then-else statements.
- Sequences of logical operators/booleans: cognitive does not increment for each binary logical operator. Instead, it does if the sequence is hard to read A sequence of logical operators is harder to read when there is a change of operators during the sequence. For example, the sequence *a AND b AND c AND d* is easier to read than *a OR b AND c XOR d*. The second sequence is harder to read since boolean expressions are more difficult to understand when composed of mixed operators.
- Recursion: Cognitive Complexity increases by one each method in a recursion cycle. In fact, the execution flow at each recursion step is not intuitive and understandable as an iterative approach.
- *break* or *continue*: These statements disrupt the execution flow of the program.Cognitive complexity increase by one for each one of these statement in the code. The *return* statement is ignored, since it breaks the execution flow making the code much more readable.

2.2.3 Rule 3

For a programmer, a sequence of five if and for structures would be easier to understand than the same structures nested one inside the other, regardless of the number of execution paths present in each structure. Since nesting increases the mental effort to understand a code, Cognitive complexity increments the score for nesting. Specifically, each time a structure, that causes a structural or hybrid type of increment, is nested inside another structure the cognitive complexity must be incremented by the number of nesting. The following example shows how nesting increment works:

```
void myMethod () {
    try {
        if (condition1) { // +1
            for (int i = 0; i < 10; i++) { // +2 (if=1 + nesting=1)
                while (condition2) {  } // +3 (while=1 + nesting=2)
                }
        }
    } catch (ExcepType1 | ExcepType2 e) { // +1
        if (condition2) {  } // +2 (catch=1 + nesting=1)
    }
} // Cognitive Complexity 9
```

Listing 2.3: Example of nesting increment

Cognitive Complexity breaks from the practice of using mathematical models to assess software maintainability. It uses human judgment to assess how structures should be counted and how a code should be maintained in order to be more readable and understandable.

2.3 LOC

Lines of Code (LOC) is a metric that counts the number of lines of text in a program. There is numerous variant for the LOC metrics:

- Source Lines of Code(**SLOC**): It returns the total number of lines in a program.
- Physical Lines of Code(**PLOC**): It returns the total number of instructions and comment lines in a program.
- Logical Lines of Code(**LLOC**): It returns the number of logical lines (statements) in a program – there may be several statements in one physical line of code. If multiple instructions or logical statements are in the same line they are still counter separately and the **LLOC** metrics will increase multiple times.
- Comment Lines of Code(**CLOC**): It returns the number of comment lines in a program.
- **Blank**: it counts the number of blank statements in a program.

To better understand LOC metrics some examples will be shown using C and Pythn[3] as programming languages. The following examples use these languages because they are the most taught from an academic point of view and are known even by people with very basic knowledge of programming languages.

```c
#include <stdio.h>

int main()
{
    int a,b;
    printf("\nHello\n");

    a=1; b=2; //LLOC +2
              //Comment at the end
}
```

Listing 2.4: LOC: example in C

- SLOC equal to ten.
- PLOC equal to seven. There are only seven lines of code that are not comments or empty lines
- LLOC equal to four. The instructions are only the function call and the 2 assignment operations. It can be noted that despite being in the same line the two assignments count s different instructions and therefore are counted separately for this variant.
- CLOC equal to two. There are only two lines of comments.
- Blank equal to two.

LOC metrics are strongly dependent on the programming language syntax used to write code. For instance, if we transpose the same example using Python the number of lines will be slightly smaller since Python does not need brackets. In the example below the SLOC is only equal to seven but the program is the same:

```python
import os

def main():
    printf("\nHello\n")

    a=1; b=2; #LLOC +2
              #Comment at the end
```

Listing 2.5: LOC: example in Python

2.4 WCC

The Weighted Code Coverage (WCC) is a new kind of metric derived from a mechanism proposed by Luca Ardito and others in the Sifis-Home project[4]. This metric presents two different versions :

- WCC Plain: the complexity value of a file (or function) is given to each line as a weight, then the metric score is computed by summing the weights for each lines covered. The score is than divided by the PLOC of the program.
- WCC Quantized: Each line has a different weight depending on its complexity and coverage. Each non covered line has a weight of zero. For each covered line we need to check the complexity of the function it is part of, if the complexity is lower than a given threshold then the line's weight is one otherwise the weight is two. The last step is to sum all the weight and divide the final sum by the PLOC of the file.

Both these metrics are computed either on an entire source file f or considering a single method/function in a program. For both versions, we can use different complexity metrics. As code complexity both Cyclomatic and Cognitive metrics explained in the previous chapters can be used. Both metrics were create as a mechanism to score code considering both complexity and coverage. The metrics are based in the following observations:

- Lines not covered by test are the worst case, without coverage bugs or unwanted changes cannot be detected by any means.
- Lines covered but with a high complexity score are considered safe since bung can still be detected. However, the high complexity can couse some issue at understanding the code.
- Lines covered and with low complexity are the best-case scenario.

Supposing a fixed complexity the minimum and maximum values are:

- WCC plain : [0, comp(f)]
- WCC quantized : $[0,2 * SLOC(f)/PLOC(f)]$

Algorithms 1 and 2 show an implementation for computing the two WCC versions.

For these algorithms the following values are given as inputs:

- **ploc**: the total PLOC of the file/function analyzed.

- **comp**: the total code complexity of the file for WCC Plain. For WCC Quantized the algorithm must implement a way to retrieve the complexity of a single block of code

- **lines**: Every line of a file with the knowledge if they are comments or blank and if they are covered at least once or not.

Algorithm 1 Calculate WCC plain metric for a single file

Require: $plocofthefile$
Require: $complexityofthefile$
  $sum = 0.0$
  if $ploc$<=0 **then return**
  end if
  for *line* in *lines* **do**
    if line is not a comment or blank AND line is covered at least once **then**
      $sum = sum + complexity$
    end if
  end forreturn $sum/ploc$

Algorithm 2 Calculate WCC quantized metric for a single file

Require: $plocofthefile$
Require: $complexityofthefile$
  $sum = 0.0$
  if $ploc$<=0 **then return** NULL
  end if
  for *line* in *lines* **do**
    if line is not a comment or blank AND line is covered at least once **then**
      get complexity of the block the line is part of
      if $blockComplexity > 15.0$ **then**
        sum += 2.0
      else
        sum += 1.0
      end if
    end if
  end for
  return $sum/ploc$

2.5 CRAP

CRAP[5] is a metric created by Alberto Savoia and Bob Evans that relates cyclomatic complexity to path code coverage. C.R.A.P. is an acronym that stands for Change Risk Anti-Patterns. It is designed to analyze and predict the amount of effort and time required to maintain or modify an existing body of code. Let f be a source file:

$$CRAP(f) = comp(f)^2 * (1 - cov(f))^3 + comp(f)$$

- **comp(f)** is the code complexity of the file(or function). CRAP was theorized using Cyclomatic complexity.
- **cov(f)** is the code coverage of a file or a function. It must be given in the range [0,1].

When a function or a file has a very high CRAP score, they are very risky to change. In fact, either the program is really complex or it has a very low coverage percentage. In both cases applying any change to that program will riskily add bugs or complications to the program. The CRAP formula was derived empirically as a result of a best-fit curve obtained through a trial-and-error process and various tests from numerous projects. Putting together these two aspects lead to a series of considerations. More complex a method is, high complexity, higher are the possibilities to introduce errors by whom is maintaining the program. So more complex functions need more testing in order not to be risky and problematic for changes. On the other hand, when the code is not very complex, but untested, if developers insert a bug in the program it will not be detected with testing and it will ruin the program. Defects and bugs can live both in complex and untested methods, and when that happens, a program presents serious maintainability issues. When the CRAP score is bigger than 30, a software is considered "crappy" and adding more tests or reducing the complexity value is necessary, but these values can vary depending on the programming language or size of the code. In fact if some developer want to obtain the crap score of an entire project it could exceed the threshold easily. To understand how complexity and coverage influence one the other, the following example is provided. With the CRAP formula some empirical consideration can be made. If a function has a cyclomatic complexity of 10, in order not to be crappy it must have a coverage of 42% or more. With a cyclomatic complexity of 25, the function must have a coverage equal to or greater than 80%. If the function cyclomatic complexity exceeds 30, then it is impossible make a method non-crappy. From the formula we can obtain the maximum and minimum value for CRAP depending on the complexity metric:

- The minimum value can be obtain with maximum coverage. This way

$$CRAP(f) = comp(f)^2 * (1 - 1)^3 + comp(f) = comp(f)$$

. The minimum value is $comp(f)$.

- The maximum value can be obtain with coverage equal to zero. This way

$$CRAP(f) = comp(f)^2 * (1)^3 + comp(f) = comp(f)^2 + comp(f)$$

. The maximum value is $comp(f)^2 + comp(f)$.

Algorithm 3 Calculate CRAP for a single file

Require: *comp* already given
  $covered = 0.0$
  $totalLines = 0.0$
  for *line* in *linesCovered* **do**
    if line is not a comment or blank **then**
      $totalLines = totalLines + 1$
      if *lineiscoveredatleastonce* **then**
        covered = covered + 1.0
      end if
    end if
  end for
  $coverage = covered/totalLines$
  return $comp^2 * (1 - coverage)^3 + comp$

2.6 SkunkScore

SkunkScore[6] is a metric that combines code smells, code coverage, and code complexity to know which are the most complex modules with less coverage. This metric was developed by Ernesto Tagwerker on October 2020. It is born has a metric to evaluate the project he was working on. The metric combines some metrics that ware already used in his work by other modules. Code smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality. Code smells are not bugs; they are not technically incorrect and do not prevent the program from functioning.

Let f be a source file:

$$Skunk(f) = (smells(f) + comp(f)/COMPLEXITYFACTOR) * (100 - cov(f))$$

- **smells(f)** is the sum of the cost of all smells present in a source file or function. The cost can vary depending on the framework used for finding code smells. **COMPLEXITYFACTOR** is a magic number used to reduce complexity.

The default value is 25 which was obtained by empirical tests done by the authors. **cov(f)** is the code coverage of a file or function. The values must be in a [0,100] range.

Finding and assigning a cost to code smells is not an easy task.

A programmer may not be able to find code smells or is not interested in them. For these reason we propose a non-smells version of SkunkScore. In this version, let suppose no code smells are present in the code, therefore **smells(f)** is always equal to zero. So the new no-smell formula for SkunkScore is:

$$Skunk(f) = (comp(f)/COMPLEXITYFACTOR) * (100 - cov(f))$$

Henceforth, Skunkscore word refers to the no-smells version. For SkunkScore no-smell version the minimum value is $comp(f)/COMPLEXITYFACTOR$ and the maximum value is $comp(f) * 100/COMPLEXITYFACTOR$. This values are obtained by fixing the complexity and using the minimum and maximumm value of the code coverage. Algorithm 4 and Algorithm5 shows a implementation for SkunkScore and SkunkScore no-smells respectively.

Algorithm 4 Calculate SkunkScore metric for a single file

```
covered = 0.0
totalLines = 0.0
smells = 0.0
COMPLEXITYFACTOR = 25.0
for line in linesCovered do
    if line is not a comment or blank then
        totalLines = totalLines + 1.0
        if lineiscoveredatleastonce then
            covered = covered + 1.0
        end if
    end if
end for
for line in the file do
    if there is a code smell then
        smells = smells 1.0
    end if
end for
coverage = covered/totalLines * 100
cost = (comp/COMPLEXITYFACTOR) + smells return cost * (100 -
coverage)
```

Algorithm 5 Calculate SkunkScore no-smells metric for a single file

```
covered = 0.0
totalLines = 0.0
COMPLEXITYFACTOR = 25.0
for line in linesCovered do
    if line is not a comment or blank then
        totalLines = totalLines + 1.0
        if lineiscoveredatleastonce then
            covered = covered + 1.0
        end if
    end if
end for
coverage = covered/totalLines * 100
cost = (comp/COMPLEXITYFACTOR) return cost ∗ (100 − coverage)
```

Chapter 3

Rust Code Analysis

3.1 Rust

Rust[7] is a modern programming language designed by Graydon Hoare and others at Mozilla Research. First announced in 2010 with support from Mozilla, then the first stable release Rust 1.0 became available in 2015 as an open-source programming language. Its most strong points are speed, memory safety, and parallelism. For these reasons Mozilla and other companies are slowly adopting Rust as programming language to develop new applications. Some examples of functions and variable declaration in Rust:

```
1 // Function are declared with the fn keyword if there is a return
      value type it must be specified after the ->
2 fn function_exemple(arg1: i64, arg2: i64) -> String {
3  if arg1 == arg2
4  {
5      println!("Same arguments")
6  }
7  // Return can e omitted by an expression with no semi column
8  "END OF FUNCTION"
9 }
```

Listing 3.1: Example of function declaration in Rust

```
1  // Each rust executable need a main function
2  fn main() {
3      // Initializing a constant variable with type i64
4      let a : i64 = 1;
5      // Mutable variable can change their value after initialization
6      let mut b : i64;
7      // No need to explicitly say the variable type, the compiler will
       set it depending on next instructions
8      let mut c;
9      b=-4;
10     b=2;
11     // Now variable c has type set to f64, any othee assignment with
       different value type will result in an error by compiler.
12     c=1.5;
13     // Print to stdout
14     println!("b={:?}",b);
15     // ERROR c has type f64 but we are trying to assign a i64
16     c=5;
17 }
```

Listing 3.2: Example of Variable declaration and initialization in Rust

Rust mostly utilize LLVM[8] for compiling , all its upgrades and performance improvement are propagated to Rust. Rust need to be compiled before it can be ran.

In order to achieve memory safety Rust does not permit the use of NULL pointers, dangling pointers, or data races. All data types must be already defined and every type is always assigned to each variable at compile time. Each variable can be mutable or immutable, they are differentiate with the use of the **mut** keyword.The data type can be either preassigned by the programmer or the compiler can derive it from the first assignment of the variable. Rust also supports Generic Data Types. Developers can use generics to create definitions for objects, like function signatures or structs, which they can then use with many different concrete data types. To use generics a function or strucs must be followed by the <> operator with a letter represent the generic type.

The code below show two examples of Generic Data type, one for a function and one for a struct:

```
fn max<T>(list: &[T]) -> &T {
    let mut max = &list[0];

    for item in list {
        if item > largest {
            max = item;
        }
    }
    max
}

struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer_p = Point { x: 5, y: 10 };
    let float_p = Point { x: 1.0, y: 4.0 };
    let number_list = vec![34, 50, 25, 100, 65];
    let result1 = max(&number_list);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result2 = max(&char_list);
}
```

Listing 3.3: Example of Generics in Rust

Rust does not support the NULL values so to emulate variables or pointers being either valid or NULL Rust provides an *Option<T>* type. *Option<T>* has two possible variants:

- **None** indicates a failure or lack of value.
- Some(val), a structure that wraps a value with type T.

Rust does not implement automatic garbage collection, instead the Resource Acquisition Is Initialization (RAII) convention is used to manage resources. The RAII convention stipulate the the resource allocation is done during object initialization, by the constructor, while resource deallocation is done during object destruction, by the destructor. There is the concept of references (using the '&' symbol) but they are all checked at compile time in order of prevent dangling pointers and other forms of undefined behavior. Each reference has a lifetime. A lifetime is a construct borrow checker uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same. Each

reference in Rust has an unique owner, references and values can be borrowed if necessary, for this cases a borrower checker is used at compile time to verify the validity of an action.

For handling shared behavior between different types and struct Rust uses *traits* that can be implemented using the **Trait** keyword in a very similar way as Java interfaces. For example any type that can be printed out as a string implements the Display or Debug traits. The programmer can implement trait for struct or enum with the **impl** keyword.
An example Traits in Rust:

```
// Declaring new trait called Display, with only one function display
trait Display {
    fn display(&self) -> String;
}
struct Person {
    name : String,
    surname : String,
    Age : i64
}
// Implementing trait for Person struct
impl display for Person {
    fn display(&self) -> String {
        format!("{} {}, {} years old", self.surname, self.name, self.
    age)
    }
}
```

Listing 3.4: An example about Trait in Rust

Rust fully supports the usage of macro in any source file. Macros are a way of writing code that writes other code, which is known as *metaprogramming*. Metaprogramming is useful for reducing the amount of code you have to write and maintain, which is also one of the roles of functions. However, macros have some additional powers that functions don't. A function must declare the number and type of parameters the function has. Macros can take a variable number of parameters. Also, macros are expanded before the compiler interprets the meaning of the code, so a macro can implement a trait on a given type. A function can't because it gets called at runtime and a trait needs to be implemented at compile time. The two main downsides of macros are:

- Macros definitions are more complex than functions. Macros have different structures and operations than the ones used in function so a developer can have issues understanding them.
- Macros must be defined **before** the developer can use then, on the other end, a function can be defined anywhere and call anywhere.

Rust implements two types of macros: declarative macros and procedural macros. Declarative macros can be declared like functions. To declare a declarative macro a programmer must use the *macro_rules!* keyword followed by the macro name and then the body between brackets. The macro can be called by using its name followed by the *!* symbol. Procedural macros allow creating syntax extensions as execution of a function. There are three types of procedural macros:

- Function-like macros: macros defined by a public function with the *proc_macro* attribute. these macros must be called in the same way as declarative macros. The function that defines a procedural macro takes a TokenStream as input and produces a TokenStream as an output.
- Derive macros add new traits into the scope of the object they are on. They can be used with the *#[derive(name)* attribute.
- Attribute macros define new attributes to attach to an item.

An example of macros in Rust:

```
// Declarative macro
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
// An example of defining a procedural macro
pub fn some_name(input: TokenStream) -> TokenStream {
}
// An example of using a derive macro
#[derive(Display, Debug)]
struct Pancakes;

// An example of an attribute macro
#[route(GET, "/")]
fn index() { //CODE }
```

Listing 3.5: An example of macros in Rust

Rust packages are called *crates* and cancan be imported by a programmer in a Rust program with the keyword **use** that can be used to import function, *structs*, *enum* and *traits* with the following syntax *use create::{to_import}*. The programmer can import specific function or import anything in the crate using the

∗ symbol. The **crate** keyword can be used to specify the current crate. All variable, function,*traits*, *enum* and *struct* are by default private and can be visible only in module they are defined, To make any of these entity visible outside the package is to add the **pub** keyword and then others package can import the entities with the use declaration. If a construct need to be visible to other modules in the same package but not be visible outside the package the **pub(crate)** keyword need to be used. The Rust ecosystem has a large variety of components that can be exploited, below some of the main components of a Rust application:

- *Rustup*: It is the main Rust toolchain installer. It manages the installation and update all the others Rust components.
- *Cargo*: It is Rust packet manager and build system. Cargo downloads, compiles, distributes, and uploads crates on an official registry called crates.io[9] that maintains all crates developed in the Rust Community. Cargo can be used to create new Rust projects. Each project can be composed by a single crate or multiple crates that may depend one to each other. The set of all crates composing the project is called the workspace. All crates dependencies, specified in a Cargo.toml file along with semantic versioning requirements, this way the programmes can decide which version to use for each dependency. Cargo also wraps other Rust components as clippy and rustfmt.
- *Clippy*: It is Rust's built-in linting tool to improve the correctness, performance, and readability of Rust code. Clippy has a series of rules, which can be browsed online and filtered by category.Some rules are disabled by default, others can be disabled if necessary.
- *Rustfmt*: it is a code formatter for Rust. It takes Rust source code as input and format it in order to produce a code formatted in accordance with the Rust style guide. It can be launched wit Cargo.

3.2 Introduction to RCA

Rust Code Analysis[10] is a Rust library used to analyze and obtain information from the source codes written in different programming languages. The initial version was developed by Mozilla[11] mostly for supporting Firefox development processes. Since Firefox has thousands of change per month, this tool has been created in order to evaluate the inherent risk of a change, prevent the introduction of new defects and avoid increasing code complexity. The library can be found on GitHub[12] and Crates.io[13]. Rust Code Analysis can be executed on the most commons operating systems (i.e Linux, macOS, and Windows). The library dependencies are managed by Cargo[14] and it is tested with a large quantity of

unit tests that takes into consideration both the most typical use and corner cases. Rust Code Analysis has, in addition to the standard APIs usage, also a command line interface called rust-code-analysis-cli which allows a fast and simple use of all Rust Code Analysis functionalities. Rust Code Analysis can be viewed as a combination of two main components:

- Language parsers: parsers are implemented for each language supported by Rust Code Analysis. Parsers are used to analyze the language and construct the Abstract Syntax Tree
- Metrics computation modules: each metric has its own module with all the code used to compute that metric.

The library is still expanding in both of these 2 components since a developer can implement a new metric or a new language without needing to modify the other component.

The Rust Code Analysis library have a large quantity of functions but all of them are need in order to implements the following four main features:

- Parse a source file and generate the relative Abstract Syntax Tree. This tree can also be printed in the command line as shown in figure 3.1.
- Use the information extracted from the Abstract Syntax Tree in order to detect in advance possible parsing errors present in the code.
- Parsing the AST and calculate all the metrics in order to evaluate the code quality.
- Print the metrics in the standard output or exporting them in the following available formats: *JSON*, *TOML*, *CBOR*.

3.3 Spaces

The first step performed by Rust Code Analysis is the creation of an Abstract Syntax Tree of the code to be examined. An Abstract Syntax Tree (AST) is a tree representation of the source code that describes the structure of the source code. Each node in the tree represents a meaningful part of the code. The Abstract Syntax Tree is very similar to a Concrete Syntax Tree but it includes less meaningful information about the code, for example punctuation and parentheses, instead AST carry only vitals information such as:

- Variable types, and location of each variable declaration
- Order and definition of executable statements

- Left and right components of binary operations
- Identifiers and their assigned values

Each information is stored as a syntax node which are the basic data structure used for navigate the whole AST. The syntax nodes can be used to extract a lot of infractions but they are usually used to:

- List all the components present in the analyzed code
- Detect parsing error in the program
- Count the number of components of a certain kind.

Each node contains a series of information that are vital to analyze the entire program. To build an AST, Rust Code Analysis uses *tree-sitter*[15], an open-source library that covers a large multitude of languages including Rust. Each Syntax node in *tree-sitter* has numerous filed and properties but the most important for Rust Code Analysis are the following:

- **kind** and **kind_id**: the type of a node. It can be represented as a String or as a numeric id. The kind parameter varies depending on the considered language, but in general it is used during the AST parsing to analyze the code (i.e finding function declaration, if or while construct, etc..). It is also adopted to represent errors found during the parsing of a code.
- **start_line** and **end_line**: Starting and ending position of a node, defined as a pair of row and column of source code.
- **start_byte** and **end_byte**: Start and ending byte of a node in the code, very useful when code is handled as a collection of bytes.
- **children**: The list of all children of a node. It can be used to traverse the whole AST.
- **parent** and siblings pointers: pointers used to access the parent node pointer. The **next** and **previous** pointers, can be used to access the siblings of the node. This pointers can be used to traverse tree backward or horizontally.

All Abstract Syntax Trees generated by tree-sitter and Rust Code Analysis starts with a root node that encapsulates the whole examined source file. This root node contains all global information of a source file such as the total number of lines and size in bytes. For more details take a look at figure 3.1. After the AST is computed, Rust code analysis parses it and divide the code into a set of spaces. A space is a structure that incorporates a function , a closure, a class and so on depending on the language.

This is the structure implementation in the library:

```
pub struct FuncSpace {
    pub name: Option<String>,
    pub start_line: usize,
    pub end_line: usize,
    pub kind: SpaceKind,
    pub spaces: Vec<FuncSpace>,
    pub metrics: CodeMetrics,
}
```

Listing 3.6: RCA: FuncSpace struct

- **name** is the name of a function, in can be not available in the case the value is None. For the Unit space the name is set to the name of the file. For some closure that may have no name it is set to anonymous.
- **start_line** is the starting row of a space in the code.
- **end_line** is the ending row of a space in the code.
- **kind** is the kind of the space. The space kinds change depending on the language used: function and Unit are used for all languages, class is used in *Java* and *C++*, *struct* can be available in *Rust*, *C* and *C++*, *trait* and *impl* are only available for the Rust language, *namespace* can be found only in C++.
- **spaces** is the list of all children spaces.
- **metrics** is structure that contains all metrics values for that space.

A Unit space is always created at the start of the parsing. The starting Unit space always encapsulates the whole source file and every other retrieved spaces as its children. When all lines in a space have been parsed the space ita merged with its parent space.The parsing process continue until all the spaces are parsed and merged into the Unit space. Starting from the Unit space all metrics and spaces can be reached by traversing the tree. To maintain all the information needed to calculate metrics, two stacks are used: a space stack which keeps track of each space to be examined and a node stack which a couple of two variables (node, nesting_level).

The general algorithm used for parsing an AST in Rust Code Analysis can be summarized in the following steps:

1. The two stacks are initialized: the space stack is empty and the node stack is initialized with the root of the AST and nesting level zero. A **last_level** variable is initialize to zero.

2. A node and nesting level are popped from the node stack. Compare the nesting level with **last_level**. If the nesting level is less than the **last_level** it means the previous node was the last node of a space so we need to make the last computations for that space. These last computations consist of: set minimum , maximum ad average of the space and then merging parent and child spaces together.

3. The last node is checked whether it is a function declaration. If a new function declaration has been found, a new space is created with its default values and then added to the space stack. Then the level is incremented.

4. The last space is taken from the stack and we used the node to compute and update all the metric for the space.

5. Then all the children of the last node are pushed into the node stack.

6. Repeat from point 2 until the node stack is empty.

7. If after we compute all the nodes there are still spaces in the space stack compute the cumulative values for all the metrics for all the remaining spaces.

For example lets analyze the following piece of code:

```
1 fn f() {
2     println!("I am a function");
3 }
4
5 fn main() {
6     // Initializing variable
7     let a = 3.;
8     f();
9 }
```

Listing 3.7: Code for AST

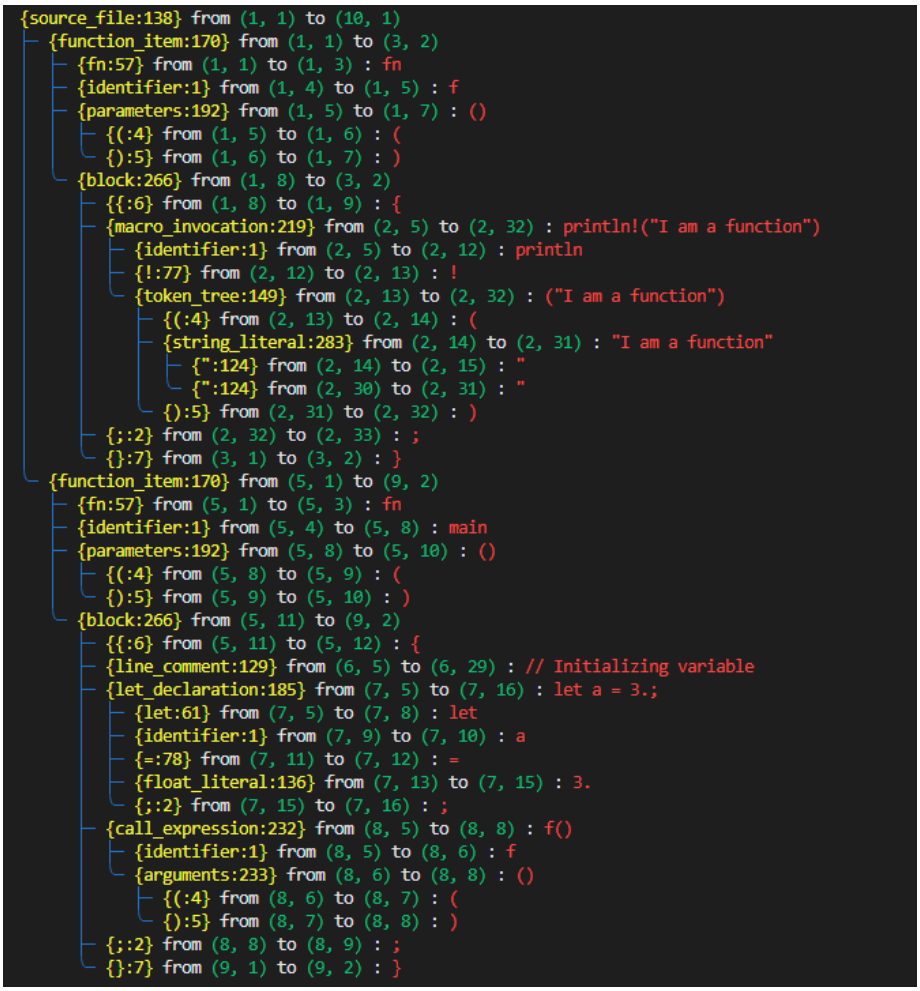

Figure 3.1: Generated AST for the code

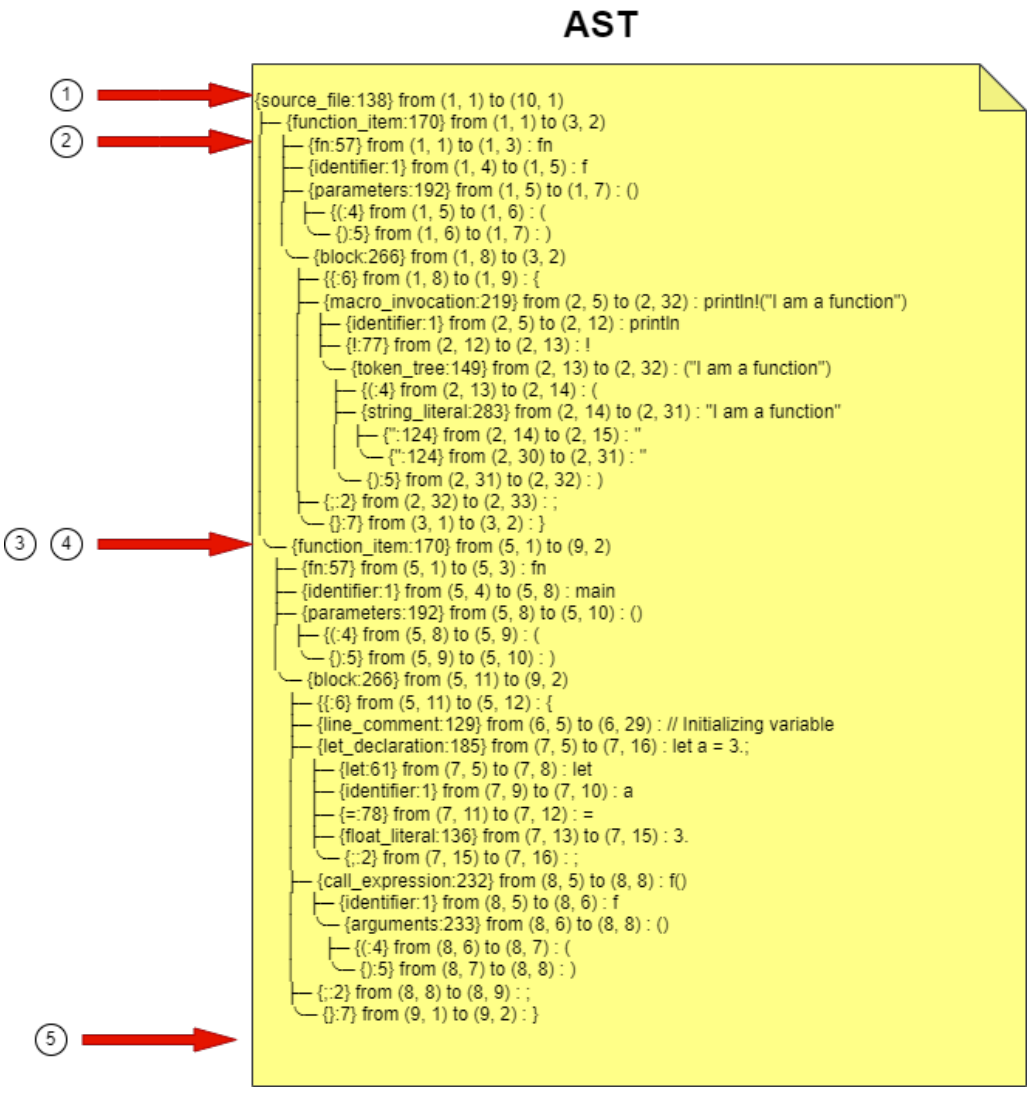

Figure 3.2: Generated AST for the example code

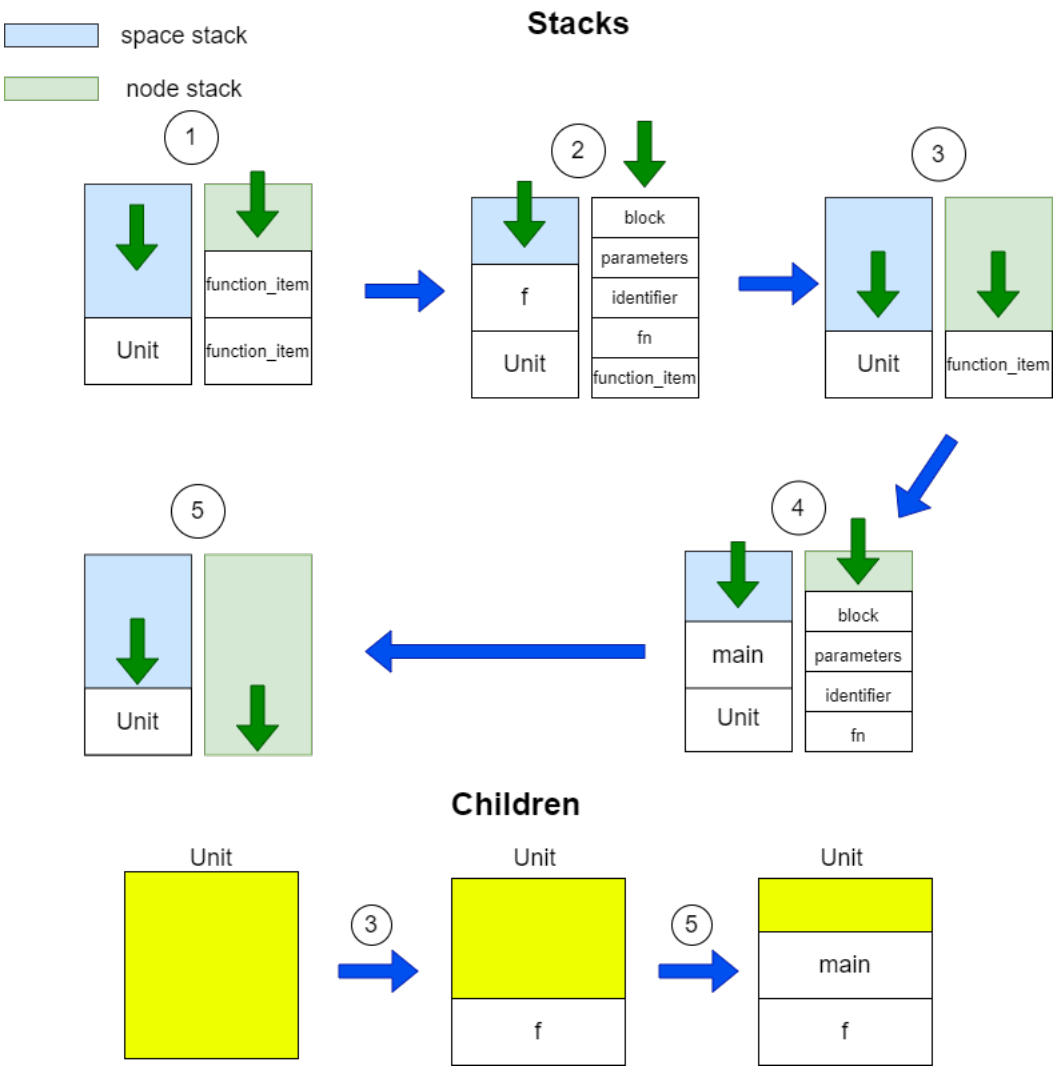

Figure 3.3: Stacks evolution for each step

Its Abstract Syntax Tree is show in figure 3.1. Figures 3.2 and 3.3 show what happen to all the stacks in the algorithm discussed above. These are main steps executed by Rust Code Analysis while parsing the AST:

- The first node of the AST is always the root. When the root is parsed Rust Code Analysis creates a Unit space that will act as a container for all the next spaces. The Unit Space also contains all the metrics for the whole file. Immediately after the space is created all its attributes are set and each metric is set to its default values.
- The two children nodes from the root are then pushed into the stack.
- The first function_item node is parsed. A new function space is created with a name equal to the function's name. The newly created state is pushed into the space stack. After that Rust code analysis starts parsing all the children of this node and updates each metric accordingly.
- When the last child node of the function is parsed, the function space is merged with its direct parent which is the starting Unit space. The function space is then removed from the stack after the merging is done.
- The same operations are done for the second function.
- When we arrive at the end of the AST the last metrics are computed and the Unit space is returned as root with all the information needed. All the other spaces can be obtained by visiting all the Unit spaces for children.

3.4 Minimum and Maximum Metrics

All Rust Code Analysis metrics are computed without running the source code so the library can only compute static metrics and cannot obtain any information at run-time. The library implements the following metrics: *Cyclomatic*, *Cognitive*, *NEXIT*, *NOM*, *NOM*, *NARGS*, *SLOC*, *PLOC*, *LLOC*, *CLOC*, *Blank*, *Halsted*, *Maintainability Index*.

With the exception of Halsted and Maintainability Index, for all the other metrics, Rust Code Analysis computes the average, minimum and maximum values. For each metric, the minimum and maximum values represent the most and the least complex space in a source file. With this information a programmer knows on which part of the code needs to intervene in order to lower the overall complexity.

In Rust Code Analysis each space contains a CodeMetrics struct, used to encapsulate all the metrics computed by the tool, which is just a wrapper for each struct associated to a metric implemented in the library. Despite their differences all the metrics have a set of common behavior, in particular each metric in Rust

Code Analysis has always a function used to merge one metric into another metric of the same type. This happens when two space are merged together. A metric is merged into another one usually when the computation of that metric for each AST node has been completed and the considered space is a sub-space of the one is going to be merged into. The minimum and maximum of each space is computed in two steps :

- After a space has completed the computation of all its nodes the *compute_minmax()* function with compare its complexity with its minimum and maximum and set the rights value. This is useful in case there are spaces already merge. If there are no spaces to merge in that case minimum and maximum will be equal the the only space complexity.
- During a merge the two spaces minimum and maximum will be compered and set accordingly with respect to the merging space.

Each metric has its own implementation of the function *compute*() which usually takes as arguments a node and the struct with the data of the considered metric. This function checks the node kind and updates the metric data when the necessary conditions are satisfied. In order to check the node kind its kind_id is used. For each different language Rust Code Analysis has implemented an enum that maps any type of kind_id for each language supported. In order to implement minimum and maximum for each metric, some changes have been made to allow each space to have information on the parts associated to its code, excluding other spaces information in the process. In this thesis we will put more emphasis on McCabe's Cyclomatic Complexity and the Cognitive complexity as they are the most studied and used during the whole thesis work.

3.4.1 Cyclomatic Complexity

As explained more in detail in section 2.1, McCabe's Cyclomatic Complexity just counts the number of linearly independent paths present in the source code.

```rust
// The `Cyclomatic` metric.
#[derive(Debug, Clone)]
pub struct Stats {
    cyclomatic: f64,
    n: usize,
}
impl Default for Stats {
    fn default() -> Self {
        Self {
            cyclomatic: 1.,
            n: 1,
        }
    }
}
// Merges a second `Cyclomatic` metric into the first one
pub fn merge(&mut self, other: &Stats) {
    self.cyclomatic += other.cyclomatic;
    self.n += other.n;
}
```

Listing 3.8: Cyclomatic implementation in RCA

The first implementation of the cyclomatic metric only contained two variable: *cyclomatic*, which is the cumulative sum of the cyclomatic complexity of the actual space plus the ones of all its sub-spaces, and n, which is the number of spaces used for calculating the average over spaces.

The cyclomatic average value is just obtained by dividing *cyclomatic* per the number of spaces n. When a new space and cyclomatic metric are created, they are both set to 1.

Two cyclomatic metrics are merged in this way: the number of sub-spaces and the *cyclomatic_sum* from the second space are added to the ones of the first space, in this way all the information is propagated one level up.

Every time the *compute*() function examines a node, it just checks whether the considered node is a statement which generates a new path in the source code.

Example of *compute*() cyclomatic implementation for the Rust Language

```
// The 'Cyclomatic' metric.
#[derive(Debug, Clone)]
impl Cyclomatic for RustCode {
    fn compute(node: &Node, stats: &mut Stats) {
        // Enum with all the kind_id for the Rust Language
        use Rust::*;
        // If the node kind\_id is one of these increment cyclomatic
        match node.object().kind_id().into() {
            If | For | While | Loop | MatchArm | MatchArm2 | QMARK
                | AMPAMP | PIPEPIPE => {
                 stats.cyclomatic += 1.;
            }
            _ => {}
        }
    }
}
```

Listing 3.9: Cyclomatic implementation for Rust Language

In the example above the operation performed by the function is pretty simple, it just checks that a *kind_id* numeric identifier is equal to one of the ids of a instruction that can generate a new path in the code for example like *if*, *for* and *while* conditional instructions. If the ids are the node is one of that type of nodes than the cyclomatic variable is incremented by one, otherwise no operation is performed. The implementation is very similar for all the other languages implemented by Rust Code Analysis , the main difference is that different languages may have different ways and keyword to represent conditional statement or loops.

Some changes were made in order to implement the minimum and the maximum values. First three more variables were added to the struct:

- ***cyclomatic_sum***: This field has the same purpose of the cyclomatic variable shown in the previous implementation. This means that the cyclomatic average is now computed using this variable instead of cyclomatic. The default value is zero.

- ***cyclomatic***: Now this variable holds only the cyclomatic value of the space excluding any other sub-space. The default value for this field is one.

- ***cyclomatic_max***: The cyclomatic maximum value of a space. It is obtained by comparing the cyclomatic value of a space with all the ones from its sub-spaces. The default value is zero which represents the minimum value a space can assume.

- ***cyclomatic_min***: The cyclomatic minimum value of a space. It is obtained by comparing the cyclomatic value of a space with all the ones from its

sub-spaces. The default value is the maximum value an integer can store.

```
pub struct Stats {
    cyclomatic_sum: f64,
    cyclomatic: f64,
    n: usize,
    cyclomatic_max: f64,
    cyclomatic_min: f64,
}

impl Default for Stats {
    fn default() -> Self {
        Self {
            cyclomatic_sum: 0.,
            cyclomatic: 1.,
            n: 1,
            cyclomatic_max: 0.,
            cyclomatic_min: f64::MAX,
        }
    }
}
```

Listing 3.10: Changes done to implement Minimum and Maximum

With the changes introduced in the Cyclomatic struct, some modifications need to be applied to the other functions too. The minimum and maximum value for a single space can be computed only when all the nodes that compose a space have been completely parsed. The *compute_minmax*() function has thus the job to set the minimum and maximum value for a space once its parsing has been completed. It also adds the computed complexity to cyclomatic_sum. The last change consists in merging the two spaces. Now when a space is merged into another its the cyclomatic_sum that must be added. After the minimum and maximum value of the two spaces are compered so that the right value can be set. Since we merge one space into another only when that space computation is concluded the cyclomatic_max and cyclomatic_min variable are been set already so we only need to compare them with the respective variable from the other space. These changes are straightforward since each type in Rust has a *min*() and *max*() method that compares the two values and return respectively the minimum and maximum value for them.

```rust
pub fn merge(&mut self, other: &Stats) {
        // Calculate maximum value
        self.cyclomatic_max = self.cyclomatic_max.max(other.
    cyclomatic_max);
        // Calculate minimum value
        self.cyclomatic_min = self.cyclomatic_min.min(other.
    cyclomatic_min);
        // Merge sum and number of spaces
        self.cyclomatic_sum += other.cyclomatic_sum;
        self.n += other.n;
}
```

Listing 3.11: Merge function for Cyclomatic

3.4.2 Cognitive Complexity

The Cognitive Complexity implementation in Rust follows all of the properties explained in the State of Art chapter, section 2.2. The first version of the code for the Cognitive results pretty complex, so one of this thesis milestones consists in the refactoring of this metric code. Differently from Cyclomatic, Cognitive computes the average over the total number of functions in a space instead on the number of different spaces. First of all, we are going to analyze the first code implementation, and then we will explain the minimum and maximum additions:

```rust
pub struct Stats {
    structural: usize,
    nesting: usize,
    total_space_functions: usize,
    boolean_seq: BoolSequence,
}
```

Listing 3.12: Struct used for Cognitive

- ***structural***: this variable is the actual Cognitive metric value. It is the cumulative sum of a space cognitive value with the ones contained in its sub-spaces.
- ***nesting***: this is an internal variable used to compute the nesting level of some conditional construct. This value is then added to structural field.
- ***total_space_function***: number of functions inside this space.
- ***boolean_seq***: This is a very important structure, used to keep track of the sequence of boolean operations in a conditional expression. In fact, if the previous operation is different from the actual one then Cognitive

complexity is incremented. The most important function for this structure is *eval_based_on_prev*() which analyzes an element in the boolean sequence and checks it with its predecessor in the same expression:

- In case the operand analyzed it the first of the sequence then operator is saved and the structural filed is incremented by one.
- If they are different then structural in incremented by one and the operator is saved
- If they are the same no operation is performed and the next operator is checked

Rust Code analysis implements two macros for handling nesting and boolean sequences:

- *compute_booleans*!() calculates the complexity of a boolean sequence.
- *nesting_levels*!() counts the nesting of a node.

This macros are very complex to read as show by the example below.

```
macro_rules! compute_booleans {
    ($node: ident, $stats: ident, $( $typs: pat_param )|*) => {
        get Iterator from the node
        let mut cursor = $node.object().walk();
        // Iterate to all the children of the given node
        for child in $node.object().children(&mut cursor) {
            // Check if the node kind_id is the same that the list of
    id given to the macro
            if let $( $typs )|* = child.kind_id().into() {
                $stats.structural = $stats
                    .boolean_seq
                    .eval_based_on_prev(child.kind_id(), $stats.
    structural);
            }
        }
    };
}
```

Listing 3.13: compute_booleans macro

This macro simply iterates over all operators in a boolean sequence, and for each operator it checks if its different from its predecessor. The next two macros are more complex than *compute_booleans*|(), so the code will not be shown and only a simple explanation will be provided.

The *nesting_levels*!() macro is used to count the number of nesting constructs in a node. Cognitive complexity is then incremented by the depth of the nesting

value. It start from the node and check each one of its parents and increase nesting until we reach the root.

The *compute*() functions for the Cognitive metric does many computations because many node kinds need to be checked, for example: ***break***, ***continue***, not expression and binary expression, etc... . Also some nodes like ***If-Then***, ***For*** and ***While*** expression need to be examined using the *nesting_levels*!() macro with a large number of starting and ending nodes types.

Code changes introduced for the minimum and maximum implementation are similar to the one performed for the Cyclomatic Metric.

```
pub struct Stats {
    structural: usize,
    structural_sum: usize,
    structural_min: usize,
    structural_max: usize,
    nesting: usize,
    total_space_functions: usize,
    boolean_seq: BoolSequence,
}
impl Default for Stats {
    fn default() -> Self {
        Self {
            structural: 0,
            structural_sum: 0,
            structural_min: usize::MAX,
            structural_max: 0,
            nesting: 0,
            total_space_functions: 1,
            boolean_seq: BoolSequence::default(),
        }
    }
}
```

Listing 3.14: Minimum and maximum changes Stats

Three new variables have been added: *structural_sum*, *structural_min* and *structural_max*;

- ***structural_sum***: This field has the same purpose of the structural variable in the previous implementation. This means that the cognitive average is now computed using this variable instead. The default value is set to zero.

- ***structural***: Now this variable holds only the cognitive value of the the space excluding any other sub-space. The default value for this field is 0.

- ***structural_max***: The cognitive maximum value for this space. It is obtained by comparing the cognitive value of this space with those of all the other

sub-spaces. The default value is set to zero which is the minimum value any space can reach.

- ***structural_max***: The cognitive minimum value for this space. It is obtained by comparing the cognitive value of this space with those of all the other sub-spaces. The default value is set to the maximum integer value which is the maximum value any space can reach.

```
pub fn merge(&mut self, other: &Stats) {
        self.structural_min = self.structural_min.min(other.
    structural_min);
        self.structural_max = self.structural_max.max(other.
    structural_max);
        self.structural_sum += other.structural_sum;
    }
pub(crate) fn compute_minmax(&mut self) {
        self.structural_min = self.structural_min.min(self.structural
    );
        self.structural_max = self.structural_max.max(self.structural
    );
        self.structural_sum += self.structural;
    }
```

Listing 3.15: Minimum and maximum changes for merge and compute_min_max

The modification to the *merge*() function follows the same pattern as the one used for the Cyclomatic metric. The first space just compares its values with the ones of a second space, taking the minimum and maximum. After this step, the two structural_sum variables are summed together. The minimum , maximum and structural_sum values of the single space are set before being merged by the *compute_minmax*() function.

In order to improve the overall readability of the code, the Cognitive metric was refactored so that macros will be replaced with a more readable and simple approach. The *compute_booleans* macro was replaced by a function with the same names that instead uses generics. The function is shown below:

```rust
fn compute_booleans<T: std::cmp::PartialEq + std::convert::From<u16
    >>(
    node: &Node,
    stats: &mut Stats,
    typs1: T,
    typs2: T,
) {
    let mut cursor = node.object().walk();
    // Iterate over all operators in the boolean sequence
    for child in node.object().children(&mut cursor) {
        if typs1 == child.kind_id().into() || typs2 == child.kind_id
    ().into() {
        // Compare with previos booleans operator and increment
    complexity if needed
            stats.structural = stats
                .boolean_seq
                .eval_based_on_prev(child.kind_id(), stats.structural
    )
        }
    }
}
```

Listing 3.16: compute_booleans function with genereics

This function simply takes the node that contains the boolean sequence and iterates over all its operators. if an operator is equal to one of the given two arguments the previous operator is checked and the cognitive complexity is incremented if the operators are different. The two typs arguments are of a generic type T which is an enum that represents the various syntax constructs of the programming language used. Plus this generic needs to be comparable and must be converted into an integer. The function always takes two types as arguments because only exactly two types are checked each time the macro was called. In case more than two arguments will be needed in the future the function can be easily modified by using a vector of types.

Another change done is the removal of the *nesting_levels* macro and adding a new approach to compute the nesting of a node. *nesting_levels* computed the nesting starting from the none and checking all of its parents until a stopping point was reached. The new approach uses a hashmap called *nesting_map* which keeps track of the nesting level in a source file. The key of the map is the node id with its value being the nesting level represented by an integer. With this approach nodes just need to check the nesting of its parent saved into *nesting_map* without retraversing the tree. The pseudo-algorithm for this approach can be summarized into the following steps:

1. Initialize an empty *nesting_map*.
2. Get the parent from the node. If the node has no parent set *nesting* equal to zero.
3. Take the nesting from *nesting_map* using the parent node id. If the key is not present in *nesting_map* just set the nesting equal to zero
4. If the node is a type of node that increments nesting increase it by one then modify the cognitive complexity accordingly.
5. Insert the node into *nesting_map* with the pair *(node_id,nesting)*.
6. Continue with the next node until all the nodes are computed.

The types of nodes that modify complexity depends on the programming language used since some languages may a have different syntax. The most common example common for a lot of languages are: *if*, *for* and *while* loops.

```rust
// Increment cognitive complexity with nesting
#[inline(always)]
fn increment(stats: &mut Stats) {
    stats.structural += stats.nesting + 1;
}

impl Cognitive for RustCode {
    fn compute(node: &Node, stats: &mut Stats, nesting_map: &mut FxHashMap<usize, usize>) {
        use Rust::*;
        // nesting_map was already initialized before
        // Check the node parent and get its nesting
        let mut nesting: usize;
        if node.object().parent().is_some() {
            nesting = if let Some(n) = nesting_map.get(&node.object().parent().unwrap().id()) {
                *n
            } else {
                0
            };
        } else {
            nesting = 0;
        }

        match node.object().kind_id().into() {
            // Node type If
            IfExpression | IfLetExpression => {
                // Check if a node is not an else-if
                if !Self::is_else_if(node) {
                    // Increment cognitive complexity
                    stats.nesting = nesting;
                    increment(stats);
                    // Increase nesting for next nodes
                    nesting+=1;
                    stats.boolean_seq.reset();
                }
            }
            // For other node the steps are very similar
            _ => {}
        }
        // Insert node into nesting_map with its correct nesting
        nesting_map.insert(node.object().id(), nesting);
    }
}
```

Listing 3.17: Implementation for nesting for Rust Language

The code above shows the implementation proposed for the Rust programming language. Only one particular case is shown because either the other cases use the same approach or they do node increase nesting and in that case, no operation is performed and the node is simply added to the map with the same nesting as its parent. This approach does consume more memory than *nesting_levels* but speed performances are better since all the operations are done using a hashmap.

Chapter 4

Weighted Code Coverage

Weighted Code Coverage[16] is a Rust library that implements different metrics used to combine together Code Coverage and Code Complexity as a single value to measure code quality. It computes the four metric describes in Chapter 2: **WCC Plain**, **WCC Quantized**, **Crap**, **SkunkScore**. In addition the these metrics the library also computes a st of cumulative metrics: *average*, *minimum*, *maximum* and *project*.

- *Average* is the average over files.
- *Minimum* is the minimum value across all files for each metric.
- *Maximum* is the maximum value over all files for each metric.
- *Project* considers the project as a whole by appending all files together when computing the four metrics.

4.1 Tools Used

This section will show and explain the tools used for *Weighted Code Coverage* development. First, the two main tools *grcov* and *rust-code-analysis* will be analyzed, after that other tools fundamental to the application will be briefly explained, particularly in their contribution to the whole process.

4.1.1 Grcov

Grcov[17] is the tool used to obtain coverage information from these files for all the Rust repositories examined in this thesis. Grcov is a Rust library maintained by Mozilla that collects and aggregates code coverage information for multiple source files. It processes .profraw and .gcda files generated either from llvm/clang or gcc

by running tests on the repositories to analyze. *Grcov* calculate the line coverage of a file. The Line Coverage of a file is the number of executed lines divided by the total number of lines. Only lines that contain executable statements are considered, not those with pure declarations. Coveralls and Covdir are two different formats which describe the coverage of a whole project file by file.
A Coveralls file has the following structure:

```
    "git": {// git info},
    "repo_token": "YOUR_COVERALLS_TOKEN",
    "source_files": [
        {
        "branches": [],
        "coverage": [null,null,0,0,0,0,1,3,5,10,2,1,1,0],
        "name": "examples/example.rs",
        },
        //other files...
    ]
}
```

Listing 4.1: Coveralls JSON structure

The main fields are:

- *git* contains some git information on the repository when available
- *repo_token* : The secret repo token for your repository. the token is used with services not supported by Coveralls.
- *source_file* is an array of JSON objects that represents each file.

The JSON object representing a file has all the coverage information about a single file present in the project root. It has two main filed called *name* and *coverage*:

- *name* is the relative file path starting from the project directory.
- *coverage* is an array of integers. Each element of the array indicates if that line is covered by tests or not.

Each line can have two values:

- NULL means the line was ignored during tests. An example of lines that are ignored during testing are comment lines.
- A positive integer number that indicates the number of times that line has been covered by tests.

The pseudo-algorithm used is explained below.

1. Read the JSON file obtaining the list of all source files. Initialize and empty hashmap called *result*.
2. For each file in source_files, save the name of the files as a key. The name will be saved with the character , in case Windows is used as Operating System all slashes will be replaced with double slashes. With Linux and Mac, the opposite approach is used,
3. From the file take the coverage vector and insert into the hash map the pain key and coverage.
4. Repeat from point 2 until all files are processed. Then return *result*.

Covdir files have the same information as a covdir file plus more information about coverage percentage. Differently from Coveralls, the JSON file is structured in a very different way. The folder is represented as a tree starting from the root with sub-folders and files as children, each child is represented using as a JSON object with as key the name of the file/folder and as value all the other information like children and coverage. Another main difference from coveralls is the coverage vector, instead of NULL Covdir used the value -1 for representing ignored lines.

The example below shos the general structure of a Covdir file:

```
{
    "children": {
        "Array.rs": {
                    "coverage": [
                        -1,
                        -1,
                        -1,
                        2,
                        2,
                        2
                    ],
                    "coveragePercent": 100.0,
                    "linesCovered": 3,
                    "linesMissed": 0,
                    "linesTotal": 3,
                    "name": "Array.rs"
                }, //other children
    },//project information
    "coveragePercent": 77.21,
    "linesCovered": 691,
    "linesMissed": 204,
    "linesTotal": 895,
    "name": ""
}
```

Listing 4.2: Covdir JSON structure

This is the algorithm used:

1. Read and obtain the root. Initialize a stack called source_files that contains a tuple with the JSON value and a string used for saving the path.
2. Initialize a new hash map called *result*.
3. Insert into *result* a value with the coverage array and the coverage percentage of the project with a key equal to PROJECT_ROOT.
4. Pop an item from the stack and save the first value into *object* and the second one into *prefix*.
5. For each children in object["children"], push into the stack the tuple (child,prefixname)
6. if the object is a valid source file, insert it into the hash map with its path as key and all the other coverage information as value.
7. If the stack is not empty repeat from 3 else return the hash map as result.

4.1.2 Rust Code Analysis

Rust Code Analysis[10] is used in the project mainly for two reasons:

- To obtain the complexity used for WCC, CRAP, and SkunkScore. When Rust Code analysis is called for a file it calculates both Cyclomatic and Cognitive complexities.
- To obtain the FuncSpace tree with all needed information about all the spaces present in the file. This is very important, especially for analyzing the complexity of all the functions in a single file.

4.1.3 Other libraries used

These other libraries are used in the project to achieve different results:

- **Crossbeam**[18]: used for concurrency and exchange messages between threads. Simple to use and simple to customize.
- **serde**[19] and **serde-json**[20]: two interconnected libraries **serde** offers a simple and efficient way to serialize and deserialize data structures by simply using macros and work directly with **serde-json** for creating JSON files.
- **csv**[21]: used for writing the CSV file. It is the most simple and efficient library for this purpose.

- **clap**[22]: used to create the command line interface for Weighted Code Coverage. Its the main library used in Rust to create command line applications
- **thiserror**[23]: used for error handling and creation since it provides a convenient derive macro for the standard library's std::error::Error trait. It can also easily handle other types of errors.

4.2 Wcc

Wcc takes as input a folder with all the source files and then a JSON file in Coveralls or Covdir format that contains the code coverage of all the files present in the folder. Weighted Code Coverage supports the same programming languages supported by Rust Code Analysis. Wcc contains three main features:

- For each source file from the folder it calculate four weighted code coverage metrics: *WCC Plain*, *WCC Quantize*, *CRAP* and *Skunkscore*. Wcc also computes some cumulative values for each different metric such as project, average, minimum and maximum
- It can export all the computed information in JSON or CSV files.
- It permits the user to choose which complexity to use for the four metrics between McCabe's Cyclomatic Complexity and Cognitive Complexity.
- It can analyze a project with two different granularity files and functions. With the first, we only analyze all the files in the project as a whole and compute the metrics regardly. With the second we analyzed each file more in detail computing metrics for each different function present in the file.

Wcc implements concurrency in order to speed up computation, the default value of thread launched is two but can be set by the user with a minimum value of two. The program also uses thresholds for each metric to define if a file/function is too complex or not. If at least one of the thresholds is exceeded then it is considered complex. The user can also manually modify the thresholds.

4.3 General algorithm and data structures

To obtain all the metrics Wcc proceeds in the following steps:

1. The user choose the complexity metric to be used, the format of the JSON file, the number of threads, the granularity, and all of the other option that can be selected.

2. The JSON file is then read and all the coverage for each file is recovered. If the file is in the Covdir format also the coverage of a file is obtained.

3. Then the project folder is read and all paths for each source file supported by Wcc are saved on a list. The list is then chunked depending on the number of consumer threads that must be launched as specified by the user.

Wcc saves each set of metric in a following object:

```
pub struct Metrics {
    pub sifis_plain: f64,
    pub sifis_quantized: f64,
    pub crap: f64,
    pub skunk: f64,
    pub is_complex: bool,
    pub coverage: f64,
}
```

Listing 4.3: Metrics Data structure

For all algorithms, either cyclomatic or cognitive complexity can be used. Each algorithm receives as input a space, the list of all lines covered for a single file, and the start and end line. From the root, it obtains the complexity that needs to be used depending on the specifications. The start and end lines are used to indicate the limit of the function we need to analyze.

4.4 WCC

WCC is available into two version *WCC Plain* and *WCC Quantized*. The first one is a simplified version of the second. *WCC Plain* does not analyze the complexity of the line depending on which block it is part of but just assigns as weight the overall complexity of the file to all lines in the code. In the next sub-sections will be shown also the actual code implementation used in the project.

4.4.1 Plain version

The complexity value of the whole file is assigned to each line of code that is covered, and than all the covered lines as summed together to get the total sum value. The WCC plain metrics are the result of the division of the total sum per the PLOC of the file. Here there is the code implemented using Algorithm 1:

```
pub(crate) fn wcc_plain(
    root: &FuncSpace,
    covs: &[Value],
    metric: Complexity,
    is_covdir: bool,
) -> Result<(f64, f64)> {
    let ploc = root.metrics.loc.ploc();
    // Select Right complexity
    let comp = match metric {
        Complexity::Cyclomatic => root.metrics.cyclomatic.cyclomatic_sum(),
        Complexity::Cognitive => root.metrics.cognitive.cognitive_sum(),
    };
    // Iterate over all lines
    let sum = covs.iter().try_fold(0., |acc, line| -> Result<f64> {
        // Check if the line is null
        let is_null = if is_covdir {
            line.as_i64().ok_or(Error::ConversionError())? == -1
        } else {
            line.is_null()
        };
        let sum;
        if !is_null {
            // If the line is not null and is covered (cov>0) the add the complexity  to the sum
            let cov = line.as_u64().ok_or(Error::ConversionError())?;
            if cov > 0 {
                sum = acc + comp;
            } else {
                sum = acc;
            }
        } else {
            sum = acc;
        }
        Ok(sum)
    })?;
    Ok((sum / ploc, sum))
```

Listing 4.4: WCC Plain implementation

4.4.2 Quantized version

The quantized version takes into consideration each space of the file while calculating the metric. The algorithm iterate for each line in the file, check that is covered, and set its weight using a default threshold of 15 for all complexity metrics used. For finding the space in which that line is part of a very simple approach is used:

1. First set the root as result, since it contains all the lines.
2. Traverse all its sub-space to if the line is part of one
3. if yes the set that space as a result and redo point 2
4. Else the last found result is the space of which that line is part.

In the WCC implementation, a stack is used to handle the iterative steps.
The Algorithm explained above is implemented with this function:

```
fn get_min_space(root: &FuncSpace, i: usize) -> FuncSpace {
    // This function find the minimum space for a line i in the file
    // It returns the space
    let mut min_space: FuncSpace = root.clone();
    // Init stack with root
    let mut stack: Vec<FuncSpace> = vec![root.clone()];
    while let Some(space) = stack.pop() {
        // Check each children to ssee if it contains the line
        for s in space.spaces.into_iter() {
            if i >= s.start_line && i <= s.end_line {
                min_space = s.clone();
                stack.push(s);
            }
        }
    }
    min_space
}
```

Listing 4.5: get_min_space function

Here there is the code implemented using Algorithm 2:

```rust
pub(crate) fn wcc_quantized(
    root: &FuncSpace,
    covs: &[Value],
    metric: Complexity,
    is_covdir: bool,
) -> Result<(f64, f64)> {
    let ploc = root.metrics.loc.ploc();
    let sum =
    // For each line find the minimum space and get complexity value then sum 1 if comp>threshold  else sum 1
        covs.iter()
            .enumerate()
            .try_fold(0., |acc, (i, line)| -> Result<f64> {
                // Check if the line is null
                let is_null = // Check if line is null
                let sum;
                if !is_null {
                    // Get line
                    let cov = line.as_u64().ok_or(Error::ConversionError())?;
                    if cov > 0 {
                        // If the line is covered get the space of the line and then check if the complexity is below the threshold
                        let min_space: FuncSpace = get_min_space(root, i);
                        let comp = // Get complexity from space
                        if comp > THRESHOLD {
                            sum = acc + 2.;
                        } else {
                            sum = acc + 1.;
                        }
                    } else {
                        sum = acc;
                    }
                } else {
                    sum = acc;
                }
                Ok(sum)
            })?;
    Ok((sum / ploc, sum))
}
```

Listing 4.6: WCC Quantized implementation

For the functions granularity the algorithms are very similar with the main difference that we do not start from the root but from the function we are analyze and lines outside that function are ignored.

4.5 CRAP

CRAP uses a best fit curve found by Alberto Savoia and Bob Evans that utilize the complexity and the coverage of the whole file in percentage. The total percentage value of the files is obtained by dividing the number of lines covered at least once and the number of all lines excluding NULL lines. This value is between 0 and 1. For files granularity, we use the coverage of the whole file, while for functions granularity we compute the coverage of each function. Code implementation for Algorithm0:

```
pub(crate) fn crap(
    root: &FuncSpace,
    covs: &[Value],
    metric: Complexity,
    coverage: Option<f64>,
) -> Result<f64> {
    let comp = match metric {
        Complexity::Cyclomatic => root.metrics.cyclomatic.
    cyclomatic_sum(),
        Complexity::Cognitive => root.metrics.cognitive.cognitive_sum
    (),
    };
    let cov = if let Some(coverage) = coverage {
        coverage / 100.0
    } else {
        get_coverage_perc(covs)?
    };
    Ok(((comp.powf(2.)) * ((1.0 - cov).powf(3.))) + comp)
}
```

Listing 4.7: CRAP Implementation

4.6 SkunkScore

SkunScore idea is very similar to CRAP with minor differences:

- SkunkScore formula does not contain power operations. It also contains a division.
- The coverage is expressed in the range [0,100]

This thesis has no interest in analyzing code smells inside source files plus detecting code smells is a very complex task. For this reason, Wcc implements only the no-smells version of Skunkscore.

Code implementation for Algorithm0:

```
pub(crate) fn skunk_nosmells(
    root: &FuncSpace,
    covs: &[Value],
    metric: Complexity,
    coverage: Option<f64>,
) -> Result<f64> {
    let comp = match metric {
        Complexity::Cyclomatic => root.metrics.cyclomatic.
    cyclomatic_sum(),
        Complexity::Cognitive => root.metrics.cognitive.cognitive_sum
    (),
    };
    let cov = if let Some(coverage) = coverage {
        coverage
    } else {
        get_coverage_perc(covs)? * 100.
    };
    Ok(if cov == 100. {
        comp / COMPLEXITY_FACTOR
    } else {
        (comp / COMPLEXITY_FACTOR) * (100. - (cov))
    })
}
```

Listing 4.8: SkunkScore no smells implementation

4.7 Files Mode

After the JSON file is read and all the covered files are saved into the hash-map, the project folder given as input to the program is analyzed to obtain a list of all the paths of valid files. A file is valid if its extension is one from one of the accepted programming languages by Wcc. The following algorithm shows how the files are selected.

1. Initialize and stack with initial value the project folder as a path and initialize an empty vector of strings.

2. Pop value from the stack.

3. If the path is a directory then add all the paths in the directory to the stack

4. If the path is a file, check if it is a valid file by checking its extension and if it is insert the path as a String in the vector.

5. If the stack is not empty repeat from point 2 else finish and return the vector as a result.

Once the program has obtained all the valid files it just needs to read them and obtain the needed metric using RCA. All the information about a single file is saved in the FileMetrics struct that contains the metrics, the file path, and the file name.

```
pub struct FileMetrics {
    pub metrics: Metrics,
    pub file: String,
    pub file_path: String,
}
```

Listing 4.9: FileMetrics data structure

During the development of the application, various versions of this part of the code were implemented. First, a simple and sequential version was developed to test how to obtain all the metrics, then this sequential version was transformed into a concurrent one using the crossbeam library and a producer-consumer approach. In both implementations the functions written in the library return as output four variables:

- A vector of FileMetrics structs used to store a the list of all metrics for each file. This variable also contains all the project cumulative metrics.
- A vector of strings that has all the ignored files.
- A vector of FileMetrics structs with all the complex files.
- The overall coverage of the project: this value is computed if the JSON is in coveralls format otherwise it is obtained directly from the JSON.

The algorithm below shows the entire process used to obtain all the file metrics.

1. Initialize two vectors *result* and *files_ignored*. Using the algorithms explained in previous chapters Wcc obtains the hashmap and the vector with all the files in the project.
2. For each file in the project:
3. Check if the hash map contains the file as key, if yet continue else insert the string into *files_ignored* and go to point 7.
4. From the hash map get the coverage array and using the Rust Code Analysis library get the root with all the spaces.

5. With the root and the coverage array get all the metrics using the complexity selected.

6. Create a new struct FileMetrics with the metrics calculated in point 5 and insert it into the result vector.

7. Repeat from point 2 with next until all files are examined.

8. Get the list of all complex files by filtering the result vector with metrics with the *is_complex* field as true.

9. Compute all the cumulative metrics and insert them into the result vector.

10. Terminate and return the result, the files ignored, the complex files, and the project coverage as a tuple.

4.7.1 Files Mode - Concurrent Version

After the serial version was implemented numerous tests were run on different repositories During the tests emerged that the application was slow for larger repositories. For this reason, the application needed to be transformed into a concurrent one. The approach used is very simple and is based on the producer-consumer model. Since there are no metrics that need information about any other file except one then each file can be computed in parallel by different threads.

The main thread acts as producer and performs the following tasks:

- It creates the message channel shared between him and all consumer threads.
- It creates and launches the consumer threads, the number of threads is defined by the user with a default value of two.
- It generates all the messages sent to the channel. Each message contains the information needed to compute the metric for a single file: the path of the file, complexity metric to use, hashmap with coverage, threshold, and prefix.
- After all thread has terminated it computes all the cumulative metrics.

Each consumer thread performs these operations:

- It waits to receive a message from the producer.
- When a message is received it computes the metric for a single file and then adds it to the results.
- If the file must be ignored it add it to the list of ignored files.

- If it receives a *None* message it terminates

The *result* and *files_ignored* variable are are accessed by shared reference protected by mutexes. To implement the channel the rust crossbeam[18] library was used. The message channel accepts a message of type *Option<T>*, this way *None* messages can be sent to force thread termination. The type of message used depends on the format used for the coverage JSON file.

However, after implementing the algorithm as explained above there was not a great increase in the speed of computation. This happened because if a lot of small files are sent with messages the thread loses a lot of time in capturing and creating the messages than computing the metrics. So to increase the computation speed a new approach was used. The approach is very similar just instead of sending the files one by one we divide the list of all the files to read into chunks equal to or very close to the number of consumer threads. Each chunk should have an equal number of elements. In this way also a few messages need to be sent but the computation is still faster.

The Algorithms below show the pseudo code for the producer and the consumer respectively.

PRODUCER:

1. Initialize two vectors result and files_ignored as shared variable between threads protected by a mutex. Using the algorithm explained in the previous section the hashmap and the vector with all the files in the project.
2. Divide all the files in the project into chunks, using the chunk_vector function.
3. Create a crossbeam channel and save the sender and the receiver.
4. Create N consumer threads and saves their handles in a vector. The thread will have the shared variable and the receiver-
5. For each chunk sends a message to the consumer threads using the sender. After all chunks are sent poison the threads by sending an N message with None as value.
6. Wait for all threads to finish.
7. After all threads are terminated get all the complex files, cumulative metrics, and project coverage.
8. Terminate and return the result, the files ignored, the complex files, and the project coverage as a tuple.

CONSUMER:

1. Get a message from the channel if it is None go to point 10.
2. From the message gets the chunk, the hash map, the thresholds, and the complexity to use.
3. For each file in the chunk:
4. if the file is not present in the hash map then lock files_ignored vector and inset the file into it, then unlock it. Go to point 8.
5. From the hash map get the coverage array and using the Rust Code Analysis library get the root with all the spaces.
6. With the root and the coverage array get all the metrics using the complexity selected.
7. Create a new struct FileMetrics with the metrics calculated in point 5 and insert it into the result vector(as on point 3 lock the mutex during operation and unlock it after it is done).
8. Repeat from point 3 with next until all files in the chunk are examined.
9. Go to point 1 and await a new message.
10. Terminate thread.

Function to chuck the files vector into chunks:

```
fn chunk_vector(vec: Vec<String>, n_threads: usize) -> Vec<Vec<String
    >> {
    let chunks = vec.chunks((vec.len() / n_threads).max(1));
    chunks
        .map(|chunk| chunk.iter().map(|c| c.into()).collect::<Vec<
    String>>())
        .collect::<Vec<Vec<String>>>()
}
```

Listing 4.10: chunk_vector function

The *chunk_vector* function is implemented by using the *chunks* function from the rust standard library. This function will try to chunk the vector into chunks with a number of elements given as an argument. The number of elements might not be a multiple of *n_thread* so the number of chunks may be greater than *n_threads* since there will be some remainder that will be put in additional chunks. The *chunks* function return an iterator of chunks, that are very similar to slices, so before returning we convert this iterator into a Vector of Vectors of Strings by

using concatenations of *map* and *collect* function to iterate all over its elements and do all the conversions needed.

As the last step, it was decided to introduce a new cumulative metric called PROJECT. This metric calculates the four metrics considering the project as a whole so each time we process a new file we add to a total its PLOC, lines covered, wcc plain and quantized sums, and total lines in the file. all these sums are used to obtain WCC PLAIN, WCC QUANTIZED, CRAP, and Skunkscore.

All the algorithms shown above are used in case the JSON is in the coveralls format, for the covdir format some changes need to be made. Since covdir already returns the coverage of the file and the total coverage of the project these values cannot be calculated and can be imported directly from the JSON. The most important difference is, of course, the fact that coveralls null values are changed into -1 in the covdir format so each time we analyze the coverage array inside the code we need to handle it differently. For these reasons the hashmap used for the covdir version is different and it has the following struct as a value:

```
pub(crate) struct Covdir {
    pub(crate) name: String,
    pub(crate) arr: Vec<Value>,
    pub(crate) coverage: f64,
}
```

Listing 4.11: Covdir data structure

- **name**: the name of the file.
- **arr**: the coverage array.
- **coverage**: the coverage of the file.

In the library, the coveralls and covdir version are implemented into 2 different functions, so that a developer can freely call the one he needs.

4.8 Functions Mode

Functions Mode is used to obtain more detailed information about the complexity of each different function present in a file. This way a programmer can understand which functions need to be modified to improve the overall complexity of the code. Functions mode is an expansion of files mode, for this reason, it returns the same result of function mode plus some additional information bout functions. For getting all the function in a file we traverse the whole FuncSpace tree returned by Rust Code Analysis and saves all spaces that have Function as type.

This mode has the following differences with respect to the file mode:

- Two new struct used for storing the information about files and functions called RootMetrics and FunctionMetrics.
- All the previous algorithms for calculating the four metrics are very similar but in this case, instead of iterating over the whole coverage array we just iterate over the lines that are part of the function.
- The consumer and producer threads are different because they need to handle new structs and also iterate over functions.

The first step is to find all the functions inside a file, we do this by getting the root of the FuncSpace tree from the file using Rust Code Analysis and after that, we traverse it all. While the tree is being traversed we also save the information about all the previous spaces we are in so that we can maintain information about all the parent nodes of the function, this way the programmer can find the function more easily. For simplicity, we save a string representing a path in the tree starting from the first parent outside the root to the function node with start and end lines for each node. We use a stack to iterate all over the nodes and each time we find a function we save the FuncSpace and its parents' path and insert a vector with all the function spaces.

This Algorithm shows how to get all the functions and their parents' paths.

1. Initialize an empty vector and a stack and put into the stack the tuple (root,"").
2. Pop the tuple from the stack.
3. Save the first element in *space* and the second element into a *path*.
4. For each child in *space.children* :
5. Save into *space_path* a string with the following format *space name* (*start line*, *end line*).
6. Store into *new_path* the string *path*+""+*space_path*.
7. If the child is a function then insert into the vector the tuple *(child,new_path)*.
8. Push into the stack the tuple *(child,new_path)*.
9. Repeat point 4 until all children are finished.
10. If the stack is not empty repeat from point 2.
11. Terminate and return result.

The following struct is used for storing all information about file metrics:

```rust
pub struct RootMetrics {
    pub metrics: Metrics,
    pub file_name: String,
    pub file_path: String,
    pub start_line: usize,
    pub end_line: usize,
    pub functions: Vec<FunctionMetrics>,
}
```

Listing 4.12: RootMetrics Data structure

- **metrics**, **file_name** and **file_path** have the same functions as they have for the FileMetrics struct.
- **start_line** and **end_line** are two integer starting from 1 and represent the number of lines in the file.
- **functions** is a list of all the functions present in the file with all the information.

Each function is represented by the FunctionMetrics struct:

```rust
pub struct FunctionMetrics {
    pub metrics: Metrics,
    pub function_name: String,
    pub function_path: String,
    pub start_line: usize,
    pub end_line: usize,
}
```

Listing 4.13: FunctionMetrics Data structure

- **metrics** the struct with all the metrics needed.
- **start_line** and **end_line** are two integer starting from 1 and represent where the function is positioned inside the file.
- **function_name** is a string with the format *function name (start, end)*, if the name of the function is not available the name <anonymous>.
- **function_path** is the path with all its parent nodes except the root.

The consumer thread is very similar to the one of files mode with the main difference that we also need to compute and saves the metrics for all the functions, then everything is saved inside a vector then we compute the metrics for the whole file but this time we save all the information inside the RootMetrics struct with also the vector of functions.

4.9 JSON and CSV file

Weighed Code Coverage can export all the information into a CSV or a JSON file. The CSV file was used because it can easily be exported to spreadsheet applications such as Excel or Google Sheets. The CSV file contains and header and then a list with all the files and metrics, for the functions mode after each file, there is a list with all the functions present in the file.

```
1  Files Mode
2  FILE,WCC PLAIN,WCC QUANTIZED,CRAP,SKUNK,IGNORED, IS COMPLEX, FILE PATH
3  app.rs,79.215,0.792,123.974,53.535,false,true,src/app.rs
4  ....
5
6  Function Mode
7  FUNCTION,WCC PLAIN,WCC QUANTIZED,CRAP,SKUNK,IGNORED, IS COMPLEX,
       FUNCTION PATH
8  app.rs,79.215,0.792,123.974,53.535,false,true,src/app.rs
9  "app_new_only_test (406, 415)",1.111,1.111,1.000,0.000,false,false,"/
       app_new_only_test (406,415)"
10 "multiple_app_test (418, 482)",7.738,0.967,8.007,1.548,false,false,"/
       multiple_app_test (418,482)"
11 ....
```

Listing 4.14: CSV file structure

The JSON file can be used to obtain a more structured way to export the information, it is also more useful for analyzing the data used in custom-made applications. The JSON file contains the following fields:

- **project_folder**: the project folder path.
- **number_of_files_ignored**: number of files ignored depending on the mode selected.
- **number_of_complex_files** :number of complex files/functions depending on the mode selected.
- **metrics**: List of all files with the metrics, if function mode is used it also contains the list of all the functions in the file.
- **files_ignored**: list of paths of all the ignored files.
- **project_coverage**: the coverage of the entire project.

Chapter 5

Results

This chapter will contain some of the graphs generated during the thesis. These graphs were created using Python[3] scripts mainly based on the *Matplotlib*[24] library. All the scripts used and the graphs generated are available publicly on the GitHub repository *wcc-analysis*[1].

5.1 Files

Our analysis on files mode is focused on analyzing some cumulative metrics derived from *Weighted Code coverage*: average over files, minimum and maximum. Two types of analysis are performed:

- *Spatial Analysis*: We analyzed the behaviors of the system with repositories of different sizes, starting from a project with only a few thousand lines to a project with more than 75000 lines of code. The main objective is to highlight the complexity of a single repository and its link with its size.
- *Time Analysis*: We study the behavior of a repository between different of its releases during time. The main objective is to understand how code quality varies during development.

5.1.1 Spatial Analysis

Various repository where analyzed using all 4 metrics. For each metric, the value has been represented as a float with 3 digits precision or as NaN in case that particular file was not found in the coveralls JSON. In particular, seahorse was

[1] `https://github.com/giovannitangredi/wcc-analysis`

chosen as a small repository and rust-crypto as an example of a medium repository.

Seahorse - Metrics for all files				
FILE	WCC PLAIN	WCC QUAN-TIZED	CRAP	SKUNK
multiple_app.rs	0	0	552	92
single_app.rs	0	0	20	16
help.rs	1.5	0.5	3	0.12
app.rs	79.215	0.836	123.974	53.535
lib.rs	0.077	0.077	1	0.04
flag.rs	34.696	0.801	48.329	15.87
error.rs	0.531	0.031	257.941	64
context.rs	24.316	0.737	33.468	9.962
action.rs	NaN	NaN	NaN	NaN
command.rs	26	0.722	40.777	22.244

Table 5.1: Seahorse results - File mode

As shown in Table 5.1 each file has different metrics results. We can see that multiple_app.rs and single_app.rs are not covered at all since the CRAP and SkunkScore metrics are very high and the Sifis metrics are both zero. error.rs has both very high complexity and very low coverage. The best-covered file is app.rs which has a wcc quantized value close to 1 and the other metrics are still on acceptable values. Also, action.rs has NaN values since it is a type and has no need to be tested. We can see how the metrics work differently depending on the context of the file. CRAP and Skunkscore have a higher result when the coverage of the file is very close to 0 instead WCC algorithms tend to 0 the lower the coverage.

The first analysis performed is to show how Weighted code coverage behaves with different repositories. Figure5.1 shows an average, minimum, and maximum both while using complexity cyclomatic and cognitive. Each repository is identified by different colors. We can observe how cognitive complexity has lower values than cyclomatic, this happens because usually cognitive have smaller values than cyclomatic. Also the bigger the repository the bigger the average and maximum in particular by looking at crap and Skunkscore we can see that almost every repository has a very large untested file but the bigger the repository the more complex is the file. The minimum shows us how almost every repository has either a file with 0 coverage or a file with 100% coverage as we can see each repository has always the minimum value of 0 (or 1 in case of CRAP cyclomatic).

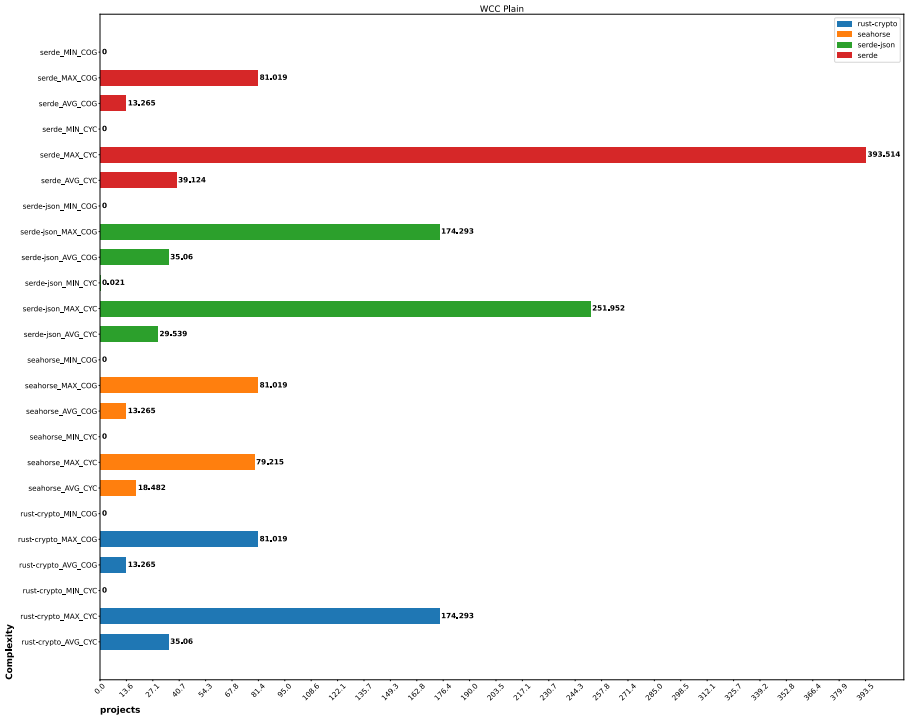

Figure 5.1: Static Analysis - WCC Plain

5.1.2 Time Analysis

Weighted Code Coverage can also be used to analyze the between different versions of the same repository. The result that will be shown in this section were obtained by analyzing different versions of two repositories: Seahorse[25] and rust-analyzer[26]. For Seahorse the following versions have been analyzed: **0.6.1**, **0.6.2**, **0.7.0**, **0.7.1**, **1.0.0**, **1.1.0**, **1.1.1**, **1.1.2**, **2.0.0**, **2.1.0**. For rust-analyzer the following versions have been analyzed: **2021-12-13**, **2021-12-27**, **2022-01-10**, **2022-01-24**, **2022-02-14**, **2022-02-28**, **2022-03-14**, **2022-03-28**. There are two types of analysis that have been performed:

- A graph that shows the number of complex files with respect to the total number of files analyzed for each version. This graph is shown both in absolute values and in percentages.

- A graph showing the variation of a cumulative metric for each version for all

4 main algorithms. There is one graph for each cumulative metric such as average, minimum and maximum.

Let us start with the first type of graph.

Figure5.2 shows that between versions the total number of files increases but the new files added are complex, therefore in the last version, we have 10 total files which six are complex. Figure5.3 show the percentage of complex files and it clearly shows that the percentage of complex files in the project increases as version increase.

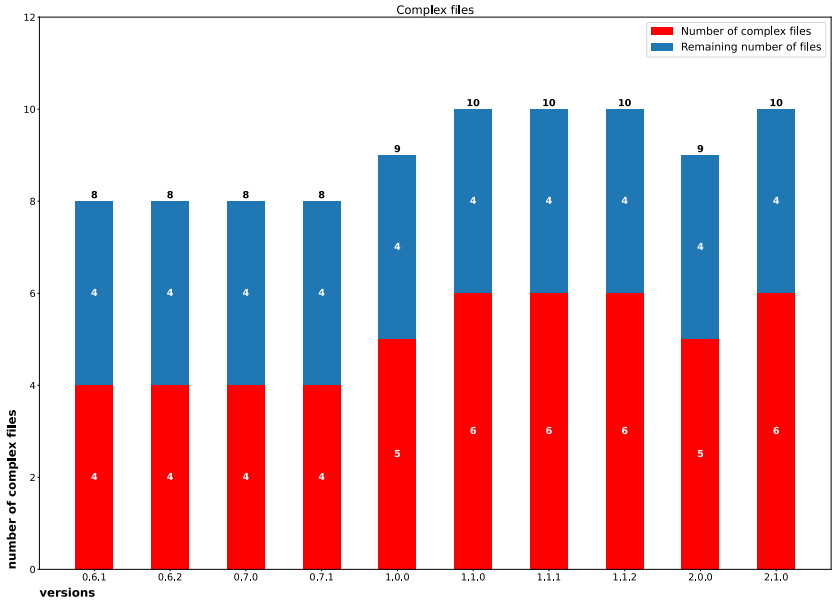

Figure 5.2: Complex files for each version of Seahorse

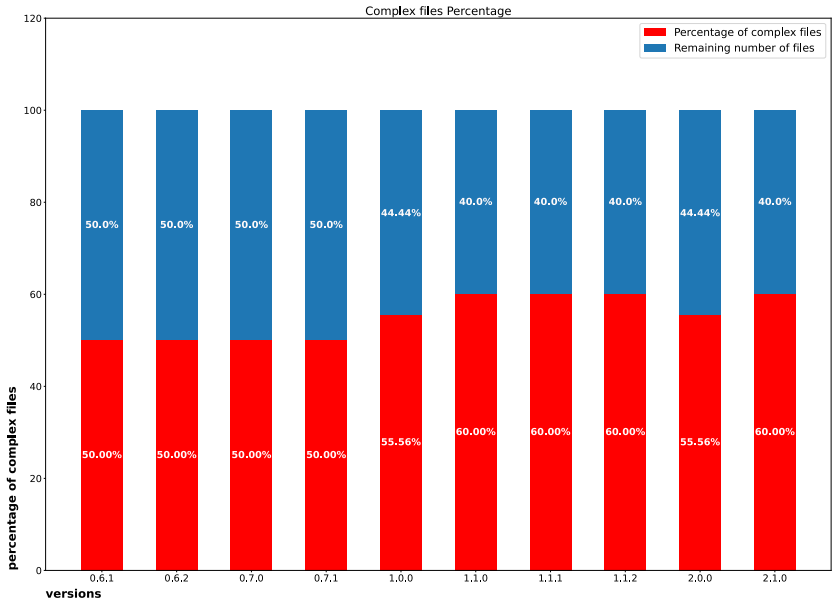

Figure 5.3: Percentage of complex files for each version of Seahorse

Rust-analyzer has more files than Seahorse. The number of complex files is still quite large. As shown in Figure5.4 the pattern is similar to seahorse. For each version the total number of files increases, but this time both the complex and non-complex number of files increase. This means that the ratio between complex and non-complex files remains stable. Figure5.5 better show that despite both increases the percentage number of complex files slightly reduces between versions even though is still very high reaching seventy-six percent in the last analyzed version.

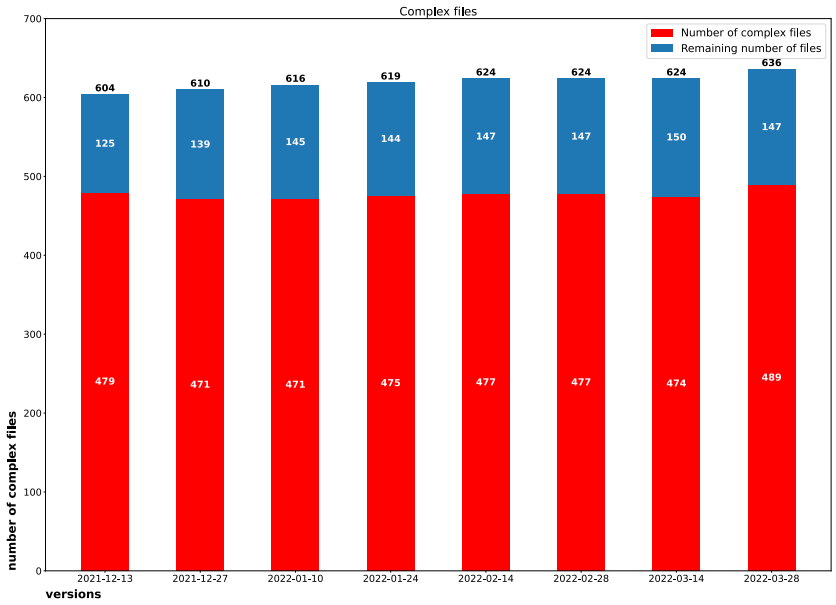

Figure 5.4: Complex files for each version of rust-analyzer

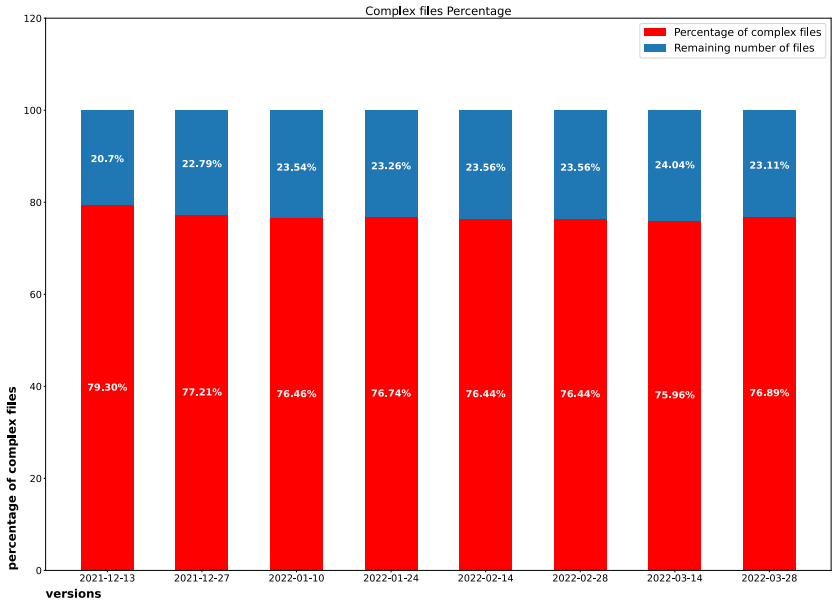

Figure 5.5: Percentage of complex files for each version of rust-analyzer

The second type of graph can be used to understand the variation of the average between different versions. The average over a file variation is particularly useful to see how the changes have affected the project as a whole.

Figure5.6 show the average variation for each metric. We can see that CRAP and SkunkScore have a similar tread mostly because they use the same approach but with a different formula, but both show that there are two periods of relative stability in the average and the on versions **1.0.0** and **2.1.0** there are two big spikes that increase the complexity of the repository. Probably because in this version 2 very large untested files were added. Instead, WCC Plain and WCC Quantized remain pretty close and don't have very big spikes. The same analysis can be done for the maximum.

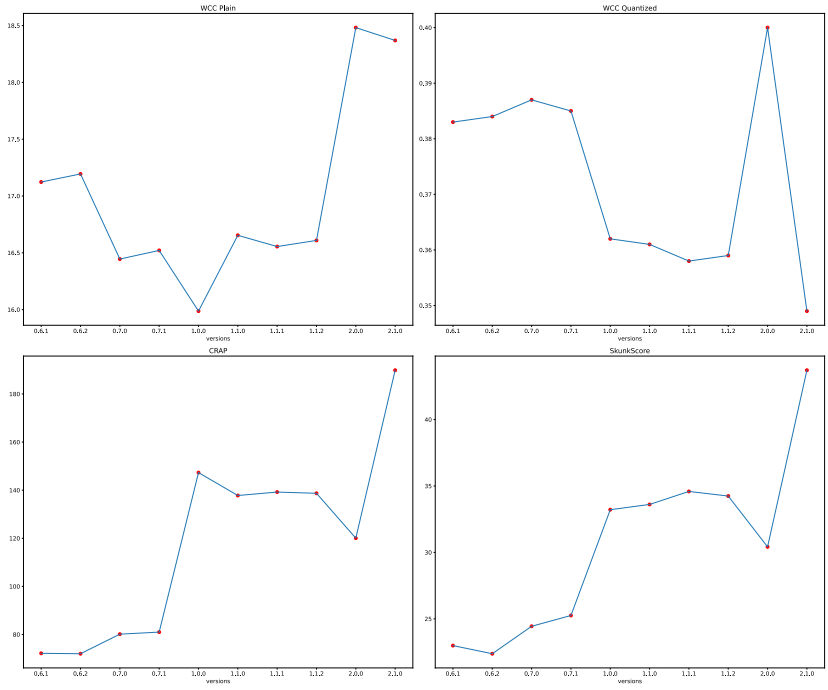

Figure 5.6: Average variation between different version of Seahorse

For the rust-analyzer, the average shows us a different type of tread. In fact Figure5.7 it show us that CRAP and SkunkScore decrease but the WCC metrics increase. This tells us that the files added or modified during the different versions of rust-analyzer had the effect to increase the coverage of the project.

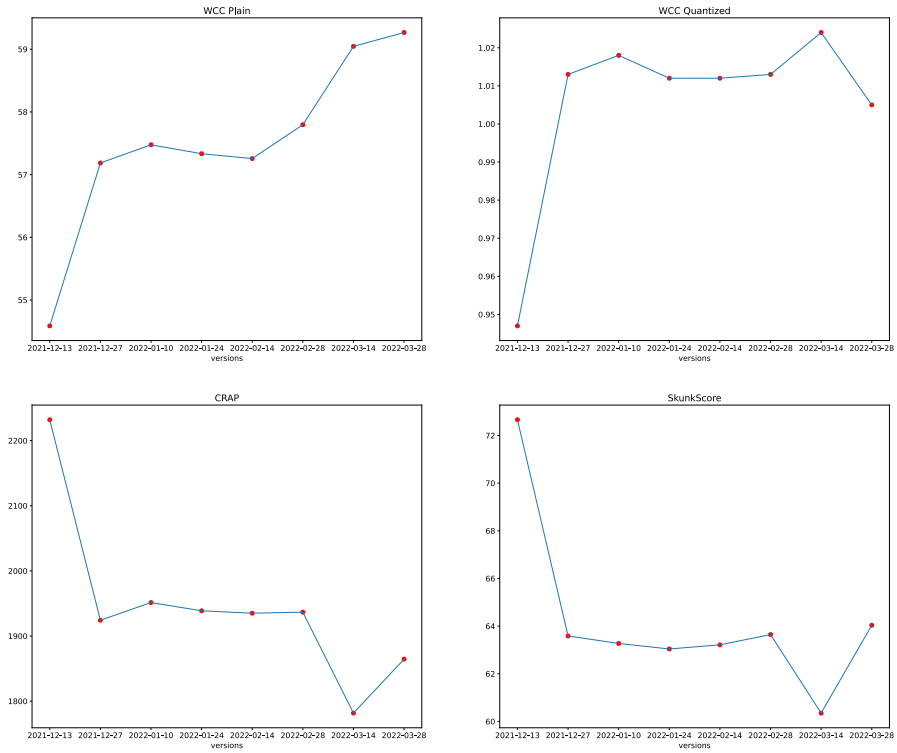

Figure 5.7: Average variation between different version of rust-analyzer

5.2 Functions

Functions mode add more granularity while analyzing the project since the cumulative project metrics remains the same all the new graphs and analysis produced are focused on how we gain an advantage by showing each function present in the file.

5.2.1 Spatial Analysis

For the spatial analysis, the attention was focused on considering what functions influenced the most the project in terms of complexity. In particular, the most 5 complex functions per file are shown for each one of the four algorithms to see what functions the developer needs to put more work on. Since showing all the functions for all files would be too much only the most interesting or important files are shown but, of course, each developer can do this analysis by them-self very easily.

Figure5.8 show the five most complex functions for the app.rs file in the seahorse repository. In the title, there is the metric value of the file and the graph shows the functions. Each sub-plot indicates a different metric so that we can differentiate better between each metric. By looking at the graph we can see aging how each metric puts different weights on coverage and complexity. In particular SkunkScore and CRAP share some functions because they both put a high weight into code coverage. In this way, we can see which functions need a lesser complexity and which ones need more coverage in tests. The developer can then see which function affects the file and where he needs to put effort into reducing the complexity or increasing the coverage of the file.

Figure5.9 uses the same approach but in a much larger repository than seahorse, but we can still see how the same pattern appears for this file again.

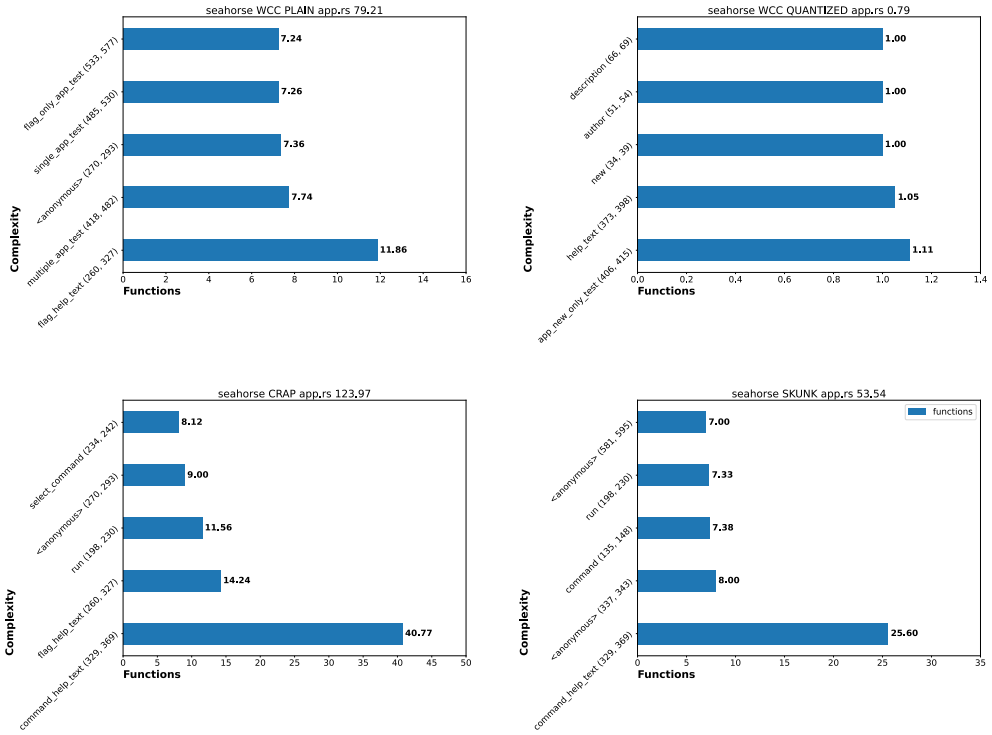

Figure 5.8: Most 5 complex function for Seahorse app.rs file

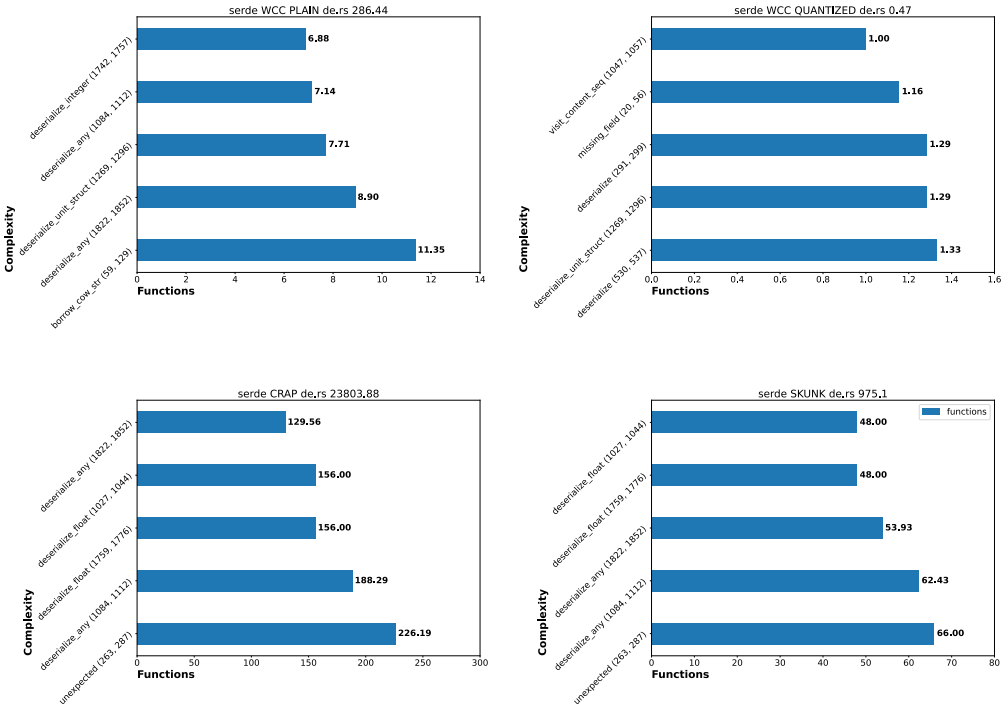

Figure 5.9: Most 5 complex function for serde attr.rs file

In Figure5.10 show the de.rs file from rust-analyzer. We can see that in this file the two WCC versions share the same set of functions, the same for CRAP and Skunkscore. In particular, we can see how the source_map function is more than 50% of the total Skunkscore of the file, thus showing if we can reduce the complexity of the file or increase the coverage we will greatly improve the file complexity.

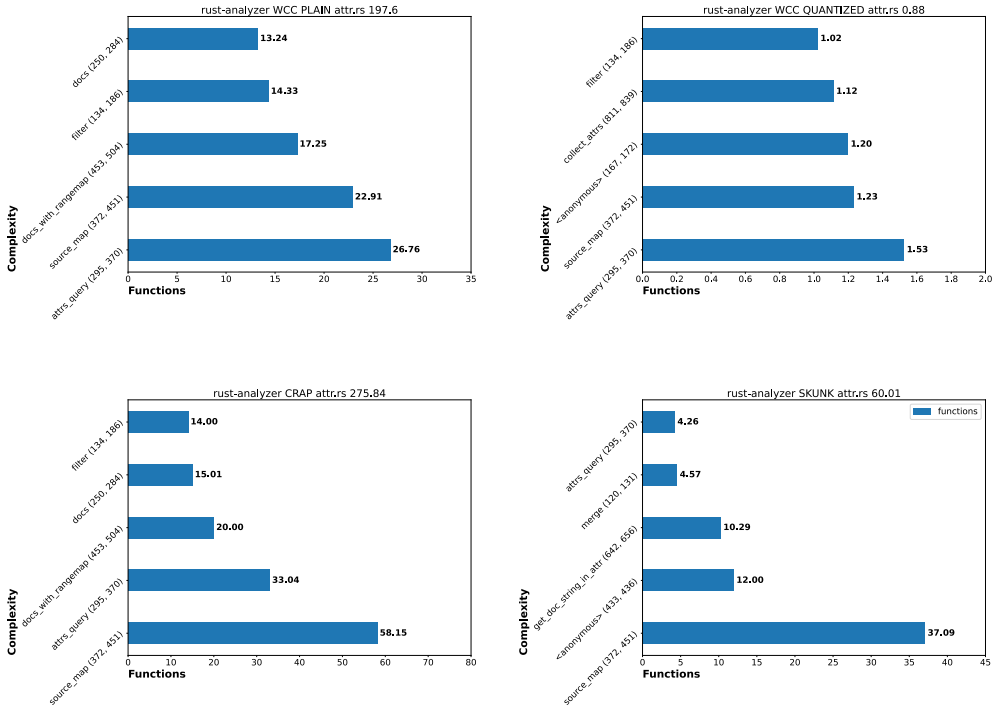

Figure 5.10: Most 5 complex function for rust-analyzer de.rs file

5.2.2 Time Analysis

For the time analysis, the same repositories from file mode are used with the same versions. The following analyses were performed:

- Analysis about the number of complex functions over the number of non-complex functions. This is similar to the one performed in the time analysis for the function with the main difference being that we are considering the single functions and not the file as a whole.
- Analyze the behavior of the most complex function in a project during different version to observe its complexity and when it changes with a different function.
- Analyze the most complex functions on the project between different versions, in this section we decided to see the most 3 complex functions in the project.

Figure5.11 and Figure5.12 show the relation between complex and non-complex functions. Differently from the file mode, we can see that the number of the complex functions is very low thus showing that a small set of functions is responsible for the complexity of some files and that some of the complexity of a file is outside functions. Despite the low number of complex functions, we can notice that the number of complex functions increases between versions, especially at version **1.0.0**and version **2.1.0**.

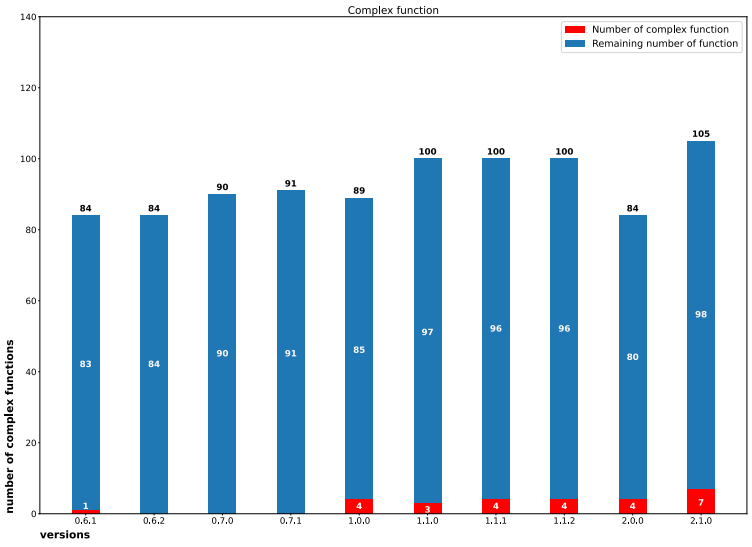

Figure 5.11: Complex function for each version of Seahorse

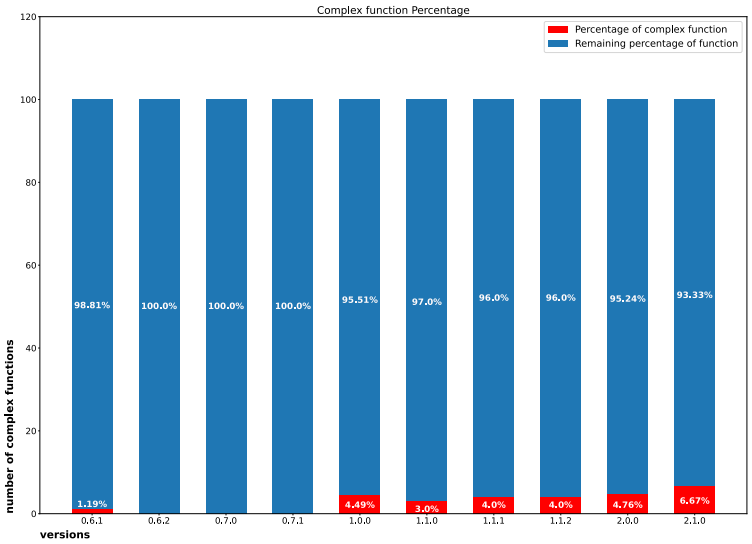

Figure 5.12: Complex function for each version of Seahorse

Rust-analyzer has a different pattern to the seahorse repository, in this repository, the number of functions is very large, and looking at Figure5.13 we can see that the number of total functions is steadily increasing for each version and the number of complex and non-complex functions as increasing as well. Despite this increment by looking at percentages as in Figure5.14 we can notice how the percentages remain very similar with very small variation between versions, thus we can conclude that the project maintains a good number of complex functions and no new update is destroying the equilibrium.

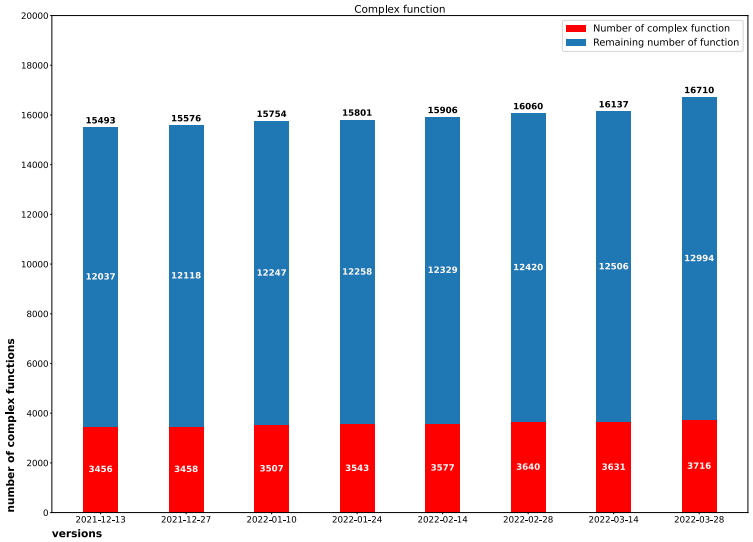

Figure 5.13: Complex function for each version of rust-analyzer

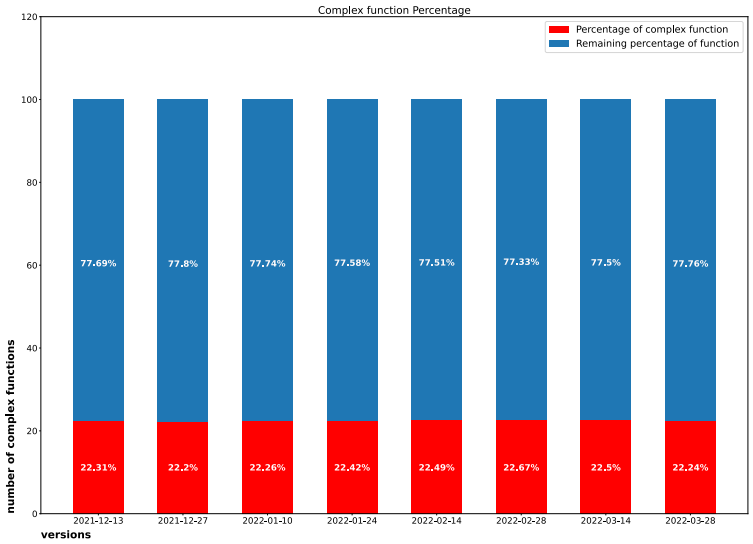

Figure 5.14: Complex function for each version of rust-analyzer

The second and third analyses are pretty similar with the main difference that for the second one we show the result as a line plot while for the third one it was more simple to use a bar plot. By analyzing the most complex function in the project we can see how the function is modified between each version for example by looking at Figure5.15 we can deduce how in rust-analyzer the most complex function is always the same and it is just slightly modified between different versions. Instead on seahorse (Figure5.16) we can see how the most complex function change 4 times and the new function always is more complex than its predecessor except for version **1.0.0**. After version **1.1.0** the situation became stable and we have the same function with small variations.

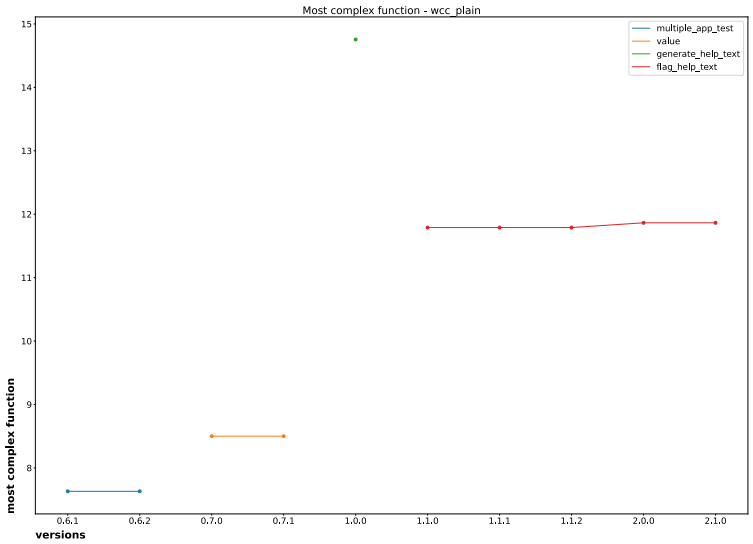

Figure 5.15: Most complex function rust-analyzer - WCC Plain

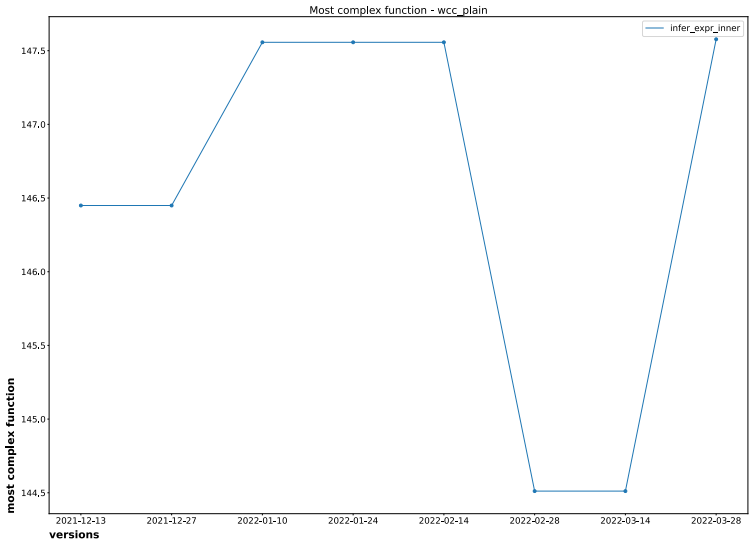

Figure 5.16: Most complex function Seahorse - WCC Plain

The most 3 complex function are shown in Figure5.17 and Figure5.18, for rust-analyzer and seahorse respectfully. We can see how the most complex function is the same as before but this analysis gives us more information than the precedent one. In particular, we can notice how in the seahorse repository the complex function almost changes every version, and it's never really stable, this is probably because seahorse is a small repository and every change affects it a lot. Instead in rust-analyzer, the top 3 complex functions are always the same and their complexity just changes slightly between each version. This happens because rust-analyzer is a larger repository than seahorse and every change implemented in each different version is probably adding new functionalities and now modifying the previous code.

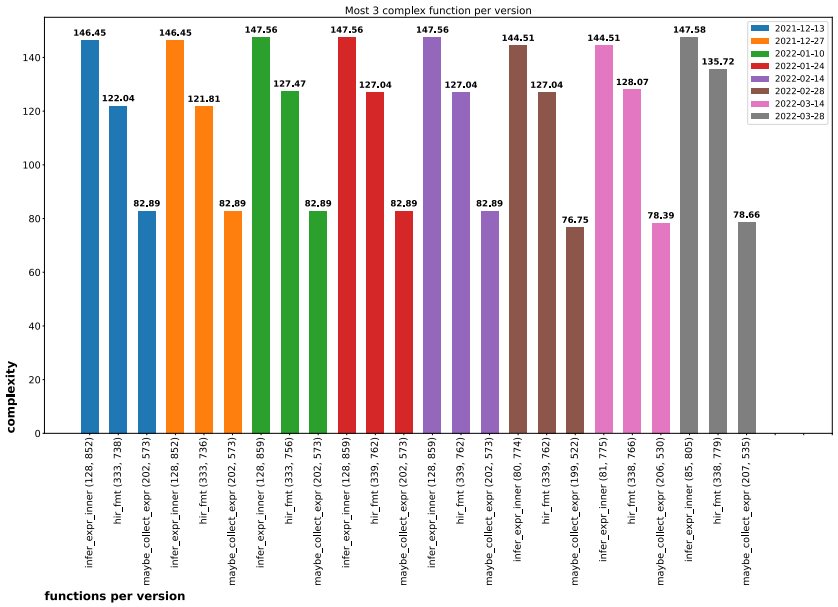

Figure 5.17: Most 3 complex function rust-analyzer - WCC Plain

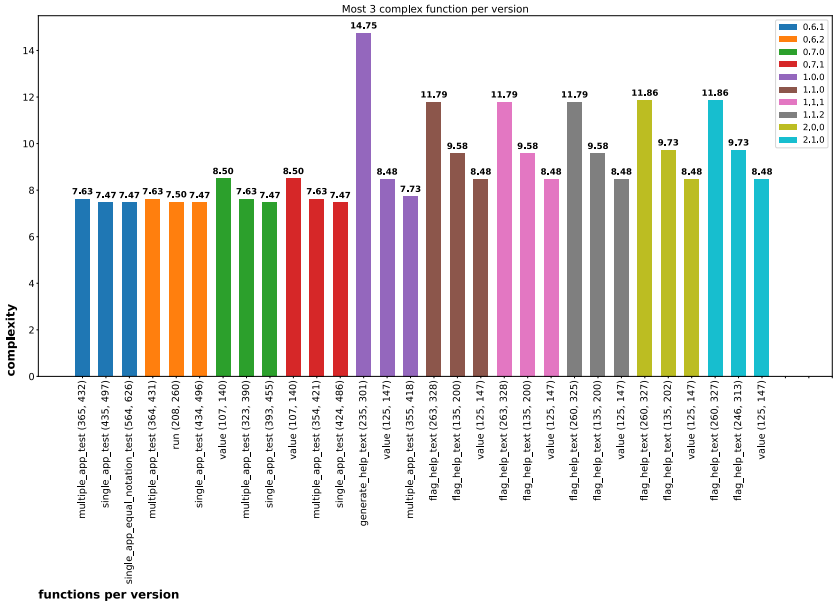

Figure 5.18: Most 3 complex function Seahorse - WCC Plain

5.3 Performance

One of the most important characteristics of any tool is its performance. In this section, we will analyze and show how Weighted Code Coverage performs in both time and memory usage. All benchmarks have been done on rust-analyzer[26] as the testing repository. Rust-analyzer has more than 75000 lines of code. Both results have been calculated for both Files and Functions mode. The tool used for obtaining the average competition times for each thread is hyperfine[27]. Hyperfine was chosen because is simple to use with a very simple command line interface. It speeds up the bench-marking process and it still has numerous configurations possible to add to the benchmarks. Plus hyperfine also supports multiple ways to export all the benchmark data in numerous formats. For memory usage analysis two tools were used : Linux *time* command and *bytehound*[28]. *Bytehound* is an open-source memory profiler for Linux. It is divided into two sub-tools:

- The profiler that gathers all the information while a program is executed.
- A web server that acts as a GUI that can be used to analyze all the data gathered by the profiler.

We used *bytehound* to analyze the memory usage over time and the memory leak present in the program execution.

5.3.1 Time

For each mode, we analyzed performances with the powers of two starting from one until 16. We choose 16 as the final value because the machine performing the benchmarks could handle 16 concurrent threads running without a loss in performance. Bear in mind that if a better machine is used to run the program the number of threads can still be increased. The case with only one thread running has very similar performances to a sequential version and it has been added to understand the difference in speed with the concurrent version.

WCC speed - File Mode		
N. threads	Mean time (s)	Speed relative to sequential
1	5.052 ± 0.030	1
2	3.821 ± 0.025	1.32
4	2.507 ± 0.028	2.01
8	1.979 ± 0.037	2.55
16	1.860 ± 0.085	2.71

Table 5.2: Mean Time - File Mode

Table 5.2 show the average competition time for files mode. As we can see with 16 the program is 2.71 times faster than the sequential version. Of course the more threads we use faster the program will complete. We can notice how stated from 4 threads the program is already 2 times faster than the sequential version. When using 2 threads we only have a 25% increase in performances. There are multiple factors for these results but the main factor could be that the two threads may not finish at the same time and the slower thread is impacting performances.

WCC speed - Function Mode		
N. threads	Mean time (s)	Speed relative to sequential
1	5.287 ± 0.023	1
2	4.008 ± 0.034	1.32
4	2.729 ± 0.080	1.93
8	2.110 ± 0.054	2.51
16	1.981 ± 0.081	2.67

Table 5.3: Mean Time - Function Mode

Table 5.3 show the result for the function mode. As can be seen, the thread is the same in terms of improving speed. The function mode takes more time to

complete than respect to files mode because we analyzed the code more in detail so more computations are needed. Despite this, the program is between 5% and 10% slower independently of the number of threads used.

5.3.2 Memory usage

Memory usage was analyzed using the Linux *time* command and analyzing the Maximum Resident set size. The Maximum Resident set size is the highest portion of main memory (RAM) occupied by the process during its entire execution time. This measure can give information about how much RAM we need to run one or more instances of our application with the same inputs. Unfortunately, the Maximum Resident set size does not give any in-depth information about the memory used by function, classes, or at a specific point in time.

WCC Memory usage		
Mode	N. Threads	Maximum resident set size (MB)
Files	1	134.1
Files	8	380.1
Files	16	545.8
Functions	1	137.4
Functions	8	378.2
Functions	16	547.6

Table 5.4: WCC Memory usage

Table 5.4 show the Maximum Resident set size for Weighted code coverage. We considered both modes while using 1, 8, and 16 threads. The result shows that both modes use the same amount of memory independently from the number of threads. The main variable that influences the memory usage of the program is the number of threads used. The more threads the program run, the higher the Maximum resident set size. The memory occupied by eighth threads is 2.83 times greater than by the sequential version, while 16 threads occupy 1.43 times more memory than 8 threads.

We can analyze the program more in detail with *bytehound*. We put most effort in analyze the impact of memory leaks during the program execution. All the data was gathered with rust-analyzer as repository while using 8 threads.

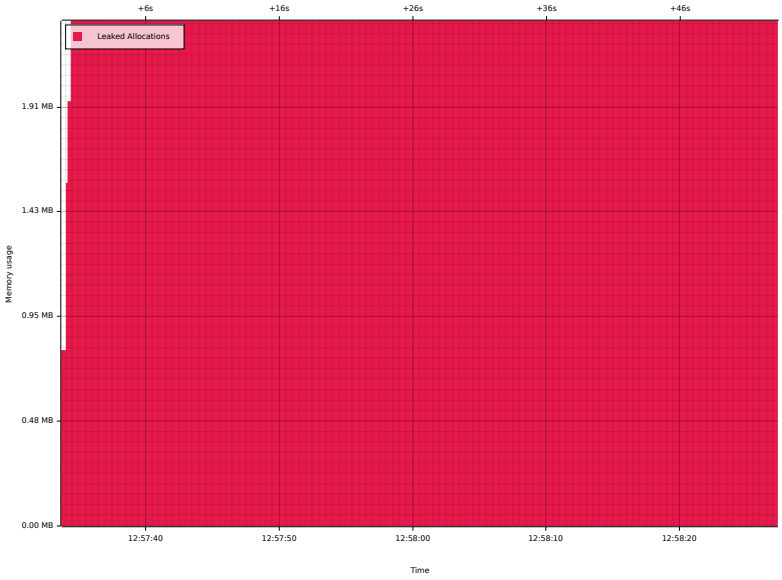

Figure 5.19: Bytehound - Memory leaks during execution

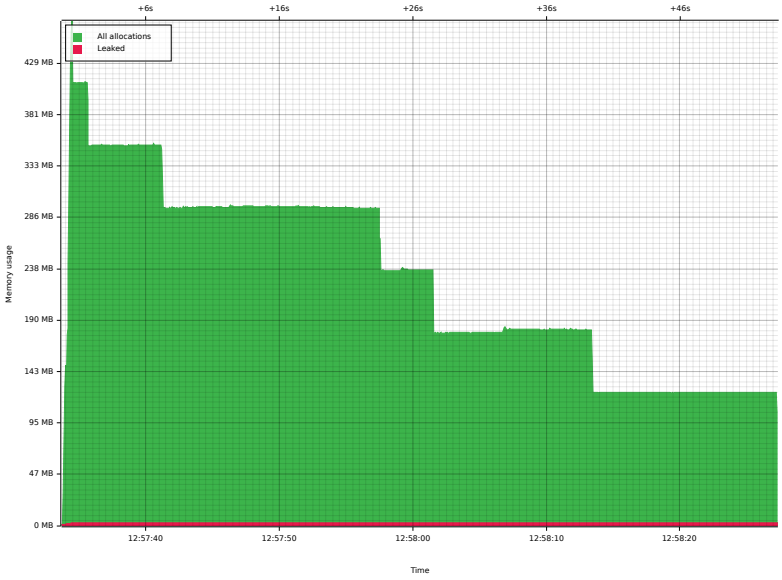

Figure 5.20: Bytehound - Total Memory usage during WCC execution

Figure5.19 show the impact of memory leak during program execution. The impact is very low, only 2 MB of memory is wasted over a hundred MB total used by the program. The leaked memory is lost directly at the program start probably

showing that the problem is located during the initialization phase of the whole process.

Figure5.20 shows the trend during the execution of the program with eighth threads. We can see more in detail how memory leaks barely affect the execution of the whole program as they are hardly visible with respect to the total memory used by WCC. As we can see *Weighted Code Coverage* reaches immediately its peak and then slowly start to deallocate memory until the end. This phenomenon happens because at the start of all threads and during the program execution, they are terminated when they finished performing their jobs.

Chapter 6

Conclusions

Code coverage and Code metrics are widely used in a great number of software development projects around the world. The standard is still focused on considering these two aspects well separated, but new possibilities and paradigms arise when it is decided to combine these two important aspects together. *Weighted Code Coverage* showed that the combination between Code Coverage and Code metrics may be exploited in successful ways to improve the overall understandability of codebases. The project successfully integrated all the new code metrics found or invented for this project and give developers a starting point to improve their maintainability processes.
Weighted Code Coverage makes it possible to compute these metrics very fast and still, it does not put a heavy tool on memory or resource management. This new tool may start a new trend that can be still research to improve software maintenance for a great number of software developers. The project can still be improved in different aspects:

- **Memory**: the project does not consume a large quantity of memory but more optimization and improvements can be applied to further reduce the memory usage of the tool.
- **Unified thread mechanism**: The Rust language can be exploited more to unify the thread mechanism used for *Weighted Code Coverage* for both functions and file mode. In its current state, the tool has just two modes separated into two files and each mode implements its own thread mechanism.
- **Integration with other tools**: The tool could b integrated with other tools that are used for computing Code metrics or Code coverage, for examples like *Grcov* or RCA.
- **New metrics**: New metrics that combine Code Coverage and Code complexity could be invented or added to the tool. Also, support for other complexity

metrics computed by RCA could be added in order to give more choices to developers.

During the whole thesis work, this candidate acquired new skills and improved his understanding of open-source development.

- The candidate understands how the production and development of open-source projects are structured.
- The candidate has gained experience in reading and writing research papers and understands the process of researching new material.
- The candidate improved his development skills. In particular the usage of best practices and improvements for better understandability of its own projects.

Bibliography

[1] T.J. McCabe. «A Complexity Measure». In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837 (cit. on p. 4).

[2] G. Ann Campbell. «Cognitive Complexity: An Overview and Evaluation». In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58. ISBN: 9781450357135. DOI: 10.1145/3194164.3194186. URL: https://doi.org/10.1145/3194164.3194186 (cit. on p. 5).

[3] *Python*. URL: https://www.python.org/ (cit. on pp. 8, 61).

[4] *Sifis-Home Mechanism (section 2.4.1)*. URL: https://www.sifis-home.eu/wp-content/uploads/2021/10/D2.2_SIFIS-Home_v1.0-to-submit.pdf (cit. on p. 10).

[5] *This Code is CRAP*. URL: https://testing.googleblog.com/2011/02/this-code-is-crap.html (cit. on p. 12).

[6] *Introducing Skunk: Combine Code Quality and Coverage to Calculate Your Project's SkunkScore*. URL: https://www.fastruby.io/blog/code-quality/intruducing-skunk-stink-score-calculator.html (cit. on p. 13).

[7] *Rust Language*. URL: https://www.rust-lang.org/ (cit. on p. 16).

[8] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/ (cit. on p. 17).

[9] *The Rust community's crate registry*. URL: https://crates.io/ (cit. on p. 21).

[10] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. «rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes». In: *SoftwareX* 12 (2020), p. 100635. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100635. URL: https://www.sciencedirect.com/science/article/pii/S2352711020303484 (cit. on pp. 21, 45).

[11] *Mozilla foundation*. URL: https://www.mozilla.org/en-US/ (cit. on p. 21).

[12] *rust-code-analysis github*. URL: https://github.com/mozilla/rust-code-analysis (cit. on p. 21).

[13] *rust-code-analysis crates.io*. URL: https://crates.io/crates/rust-code-analysis (cit. on p. 21).

[14] *Cargo Rust packet manager*. URL: https://doc.rust-lang.org/cargo/ (cit. on p. 21).

[15] *Tree-Sitter Documentation*. URL: https://tree-sitter.github.io/tree-sitter/ (cit. on p. 23).

[16] *Weighted Code Coverage*. URL: https://github.com/SoftengPoliTo/weighted-code-coverage (cit. on p. 42).

[17] *Grcov Github*. URL: https://github.com/mozilla/grcov (cit. on p. 42).

[18] *Crossbeam Channel git*. URL: https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-channel (cit. on pp. 45, 55).

[19] *Serde*. URL: https://serde.rs/ (cit. on p. 45).

[20] *Serde JSON Github*. URL: https://github.com/serde-rs/json (cit. on p. 45).

[21] *CSV reader and writer for Rust*. URL: https://github.com/BurntSushi/rust-csv (cit. on p. 45).

[22] *Command Line Argument Parser for Rust*. URL: https://github.com/clap-rs/clap (cit. on p. 46).

[23] *thiserror*. URL: https://crates.io/crates/thiserror/1.0.24/dependencies (cit. on p. 46).

[24] *Matplotlib: Visualization with Python*. URL: https://matplotlib.org/ (cit. on p. 61).

[25] *seahorse A minimal CLI framework written in Rust*. URL: https://github.com/ksk001100/seahorse (cit. on p. 63).

[26] *rust-analyzer A Rust compiler front-end for IDEs*. URL: https://github.com/rust-lang/rust-analyzer (cit. on pp. 63, 81).

[27] *Hyperfine A command-line benchmarking tool*. URL: https://github.com/sharkdp/hyperfine (cit. on p. 81).

[28] *Bytehound - a memory profiler for Linux*. URL: https://github.com/koute/bytehound (cit. on p. 81).