Research, Implementation and Analysis of Source Code Metrics in Rust-Code-Analysis

Research, Implementation and Analysis of Source Code Metrics in Rust-Code-Analysis

Luca Ardito, Marco Ballario, Michele Valsesia
Dept. of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
luca.ardito@polito.it
s286154@studenti.polito.it
michele.valsesia@polito.it

***Abstract*—The software industry is proliferating at an unprecedented pace, with a massive volume of software being released every day. Among the manifold challenges faced by software engineering researchers, one of the most significant is maintaining and enhancing software quality. Software metrics, designed to quantify various aspects of software, are essential in achieving this goal. They provide developers with a comprehensive snapshot of a codebase's status throughout its evolution, thereby facilitating timely intervention and continual improvement. Tools like Rust-Code-Analysis (RCA), developed and maintained by Mozilla, serve as crucial aids in this endeavour. RCA is a static code analyser that scrutinises a source code without executing it and computes a series of source code metrics, which quantitatively assess code characteristics such as complexity, maintainability, and robustness. The present article seeks to contribute to this area by undertaking a threefold task. Firstly, we intend to explore new source code Java metrics that can be integrated into RCA. We have chosen Java language due to its not yet declined pervasiveness in many industrial software and world of smartphones. The metrics will be selected based on their potential to provide valuable insights into codebase status and facilitate optimisation. Once the new metrics have been identified, the second part of our task involves implementing these metrics within RCA's library and also accessed through its CLI. This involves the coding and integration of the metrics using the modern Rust language, taking advantage of its unique features like memory safety without garbage collection, and data concurrency. Finally, to ascertain the effectiveness and reliability of metrics, we conduct an evaluation using diverse Java repositories. This involves studying the values generated by these metrics across repositories of varying sizes and levels of activity. From the smallest library to large-scale applications, our analysis spans various types of repositories, ensuring comprehensive coverage.**

***Index Terms*—software engineering, software quality, software metrics, static analysis**

I. Introduction

In today's digital age, where an increasingly massive amount of software is continuously produced, executed and maintained, software quality is more crucial than ever. *Software metrics* are methods to quantitatively measure particular aspects of software products, processes, and resources [1]. Measurements enable the evaluation of software quality in the most objective way possible. However, each software metric has restrictions and limitations, and an optimal overall software quality indicator still does not exist.

Source code metrics are a subset of software metrics which specifically focus on measuring particular source code characteristics [2]. Software engineers widely use source code metrics to identify complex or risky areas in the source code, assess the quality and maintainability of a project over time, evaluate software performance, optimise source code for specific targets and improve team productivity. The history of source code metrics can be divided into two main ages [3]. Before 1991, the first age of source code metrics, software engineering researchers proposed metrics mostly based on complexity, such as the number of lines of code (LOC) and its derivative, McCabe's Cyclomatic Complexity (CC) and the Halstead metrics suite. After 1992, with the beginning of the second age of source code metrics, new metrics started focusing more on measuring object-oriented design and paradigm aspects. In this period, Chidamber and Kemerer defined one of the most adopted sets of object-oriented metrics of all time, known in the literature as the CK metrics suite. Lorenz and Kidd and the MOOD metrics suites are also some of the results of this era. *Static code analysers* are automated tools designed to analyse a source code without executing it, making them particularly adapted to quickly computing static source code metrics. Scientists started reasoning about *static code analysis* even before the birth of digital computers by analysing algorithms and programs on paper. However, the development process of automatic static code analysis tools only began in the late seventies, and its progression can be split into three generations [4]. The first generation of static code analysers, in the late seventies, focused only on finding errors and bugs. A product of that generation is *Lint*, a program to flag common mistakes and provide warnings in a C language source code. Despite its excellent design, however, the tool also produced a lot of false positives, which required significant time to review. In the late nineties and early years of the new century, the second generation of static source code analysers introduced two new enhancements: path analysis techniques and separation of the analysis engine from the database of known issues. Path analysis techniques refer to all those mechanisms capable of switching the focus from reasoning about individual source code files to source code file's runtime interactions. Separating the analysis engine

from the database of known issues allows the two entities to develop separately. One can be extended or improved without the other being conceptually affected. However, despite the improvements, these products presented scalability problems and reported still many false positives. To overcome these issues, later in the year two thousand, the third generation of static code analysers started using abstract representations of source code blocks: *Abstract Syntax Trees (AST)*. An AST representation captures the core logic of a program by removing language-specific details and treats source code blocks as a series of nodes in relation to each other, leading the way for the execution of more efficient analyses.

Rust is a modern programming language focused on performance and safety, even in a concurrent environment. It has been designed in 2006 by Graydon Hoare, a Mozilla Research employee, as a personal project. After being announced by Mozilla in 2010, Rust has grown fast thanks to its features, gaining much attention worldwide, even from major software engineering companies. The Mozilla Foundation, in particular, is trying to adopt it for more and more of its projects, including the Firefox browser. *Rust-Code-Analysis*[1] *(RCA)* is a Rust library and a command line project started by Mozilla and then heavily expanded and improved, over the last three years, with the contribution of the Department of Control and Computer Engineering of Politecnico di Torino which has developed most of the metrics present in the software, unit and integration tests, and the Continuous Integration (CI) environment [5]. It is a static code analyser able to extract several source code metrics from various programming languages. The project is open-source and it has been created with the aim of supporting the Firefox browser development process, later explained in this article. To avoid confusing this software with the Rust programming language, henceforth, we are going to call it by its own acronym: *RCA*. This research aims to extend RCA by implementing in its library new source code metrics for the Java language. We have chosen a set of six new metrics starting from an analysis of the state-of-art, both from an academic and market perspective, and considered their implementation feasibility within RCA. As the next step, we have developed the selected metrics in order to have a way to automatically collect measures to study. Finally, we have analysed the data generated by these metrics on codebases with different characteristics with the goal of demonstrating the effectiveness of these metrics on real source code repositories. The main objective is to find answers to the following research questions:

- **RQ1**: Is the value N for a metric X good for a codebase with K lines of code?
- **RQ2**: What do the X metric values for a codebase Y show?
- **RQ3**: When can a codebase be considered maintainable?
- **RQ4**: What are the most valuable source code metrics?
- **RQ5**: What are the thresholds for each metric value?

[1]https://github.com/mozilla/rust-code-analysis

We have organised the rest of the paper as follows. Section II describes the research process and the series of works we have consulted to choose these six metrics. Section III reports the definition of these metrics and describes their implementation process in detail. Section IV presents the procedure followed to produce the research results. Section V details our findings and reports both quantitative and qualitative results through charts and tables, highlighting the most significant discoveries as well as answering research questions. Finally, Section VI summarises our work and its main outcomes.

II. Related Work

Given the broad utility of software metrics in software engineering, computer science researchers routinely engage in the ongoing review and evaluation of existing metrics and the proposal of modern ones. For our work, we have looked at different software metrics from an academic and a market perspective by reviewing several scientific papers and software products. Combining these two approaches together, we were able to evaluate both formal utility and practical feasibility in choosing the metrics.

Our *academic analysis* started from a study which reports a list of the most cited software metrics in literature in addition to a list of tools to compute them [6]. The work mainly focuses on *software maintainability*, defined as the ease with which a software system or module can be modified in order to be improved, corrected, or adapted to its environment. This Sistematic Literature Review provided valuable information for our research. Still, the research results value is strictly related to software maintainability, which is only one of the aspects considered in our study. Other works have instead produced a list of the most reviewed software metrics in the software engineering state-of-art considering separately *static metrics*, obtainable from the source code, and *dynamic metrics*, only retrievable from a compiled source code model or through its execution [2] [7]. Given the nature of the tool we are working with, we are only interested in static source code metrics. These two articles point out exactly the difference between the two kinds of metrics, in addition to providing us with a trace for finding the optimal metrics to develop by limiting the implementable choices. *Object-oriented metrics* are among the most reviewed types of source code metrics since they measure object-oriented software characteristics. A recent study critically evaluates both static and dynamic object-oriented metrics in depth and introduces a new concept of hybrid metrics, a new class of metrics obtained from the combined usage of both types of metrics [8]. In another article, instead, the five CK metrics, plus some traditional ones, are classified based on five determined properties in order to demonstrate their usefulness in improving object-oriented source code quality [9]. Given the broad usage of this particular programming paradigm, there has undoubtedly been an increasing interest in object-oriented metrics in recent years. However, the most widely known and used object-oriented metrics can only be computed straightforwardly via dynamic analysis, leaving us with only a few possible options.

More recently some researchers have considered two modern additional categories of software metrics: *component-based and aspect-based software metrics* [10]. Software components are the software equivalent of hardware components. In component-based programming, application designers define software components as independent entities to assemble to form the whole software product. Aspects are an abstraction that represents software requirements or functionality scattered across multiple software modules. They are a recent concept designed to ease the difficulty of implementing cross-cutting concerns features in modern software projects. However, the applications of both component-based and aspect-based software metrics are restricted to more advanced software products developed exploiting these new types of programming paradigms. Hence, compared to the object-oriented ones, they are less adopted. Besides maintainability, programmers also use software metrics to assess the security level of programs. A plethora of articles propose specific *security metrics* alongside examples of their computation [11] [12] [13]. Most of the metrics proposed are related to object-oriented safety aspects such as data hiding and error and exception handling, which can cause security issues in the final software product if overseen. However, since they are strictly associated with objects, some proposed metrics are only precisely implementable via dynamic means. Furthermore, even if a static code analyser would implement them, their usage would usually be not advised for performance reasons. Static code analysers aim, in fact, to compute source code metrics precisely and fast.

Finally, for our *market analysis*, we have investigated software metrics produced by a small set of proprietary and open-source code analysers used daily by developers worldwide by inspecting their documentation. The five products selected for this documental analysis are *SonarQube*, *NDepend*, *Visual Studio*, *JaSoMe*[2] and *CK*[3]. This market-focused investigation made emerge practical utility of aspects which have not been considered previously. Among them are the detection of duplicated code, the usage of a database of known issues, and test-related metrics. Duplicated code is commonly subjected to accidental mistakes by programmers since it is easy to forget to update all occurrences of duplicated source code when maintaining a codebase [14]. Using a database of known issues is especially profitable for detecting security threats in the source code. The database consists of a giant list of well-known security issues consultable during the analysis to spot the most common mistakes quickly, working similarly to Lint but on a larger scale. At last, test-related metrics allow users to verify the formal correctness of the inner logic of software quantitatively by performing runtime tests [15].

III. Metrics definition and implementation

Analysing academic works and market products resulted in a list of theoretical candidate software metrics to implement. Before including any of those metrics inside RCA, though, we have performed a feasibility analysis task to determine the most fitting ones. We have performed this practical examination by considering the RCA software architecture, the Rust language capabilities, and the programming languages on which each candidate metric is applicable. RCA can generate metrics at different levels of granularity, adapting metrics computation to the input programming language. To better understand this, we introduce as an example the source code of a *"Hello World"* Rust program. The following Rust code is composed of a main function whose sole purpose is to print out the well-known string.

```
fn main() {
    println!("Hello World!");
}
```

The measurement process is made possible by traversing and reviewing in a pre-order fashion the nodes of an intermediate representation of the input source code, the Abstract Syntax Tree (AST), internally generated by the *Tree-Sitter* library. Figure 1 shows, as graphical evidence, the AST produced by parsing the *"Hello World"* Rust program. On top of the AST,

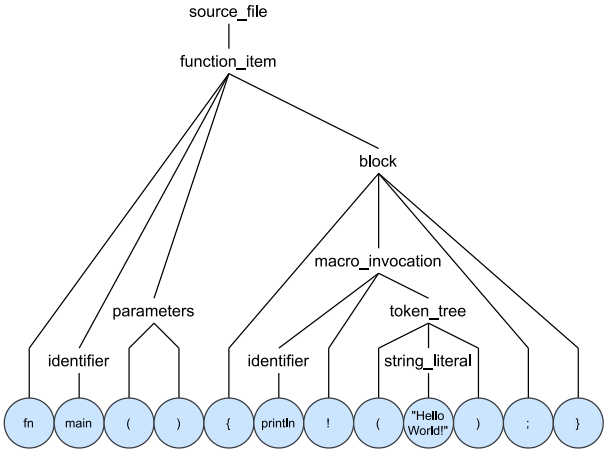

Fig. 1. AST graphical representation of a *"Hello World"* Rust program.

RCA builds another abstraction by dividing the source code into blocks called *spaces*. A space represents any structure in the source code which could contain a function declaration, such as a class or a namespace. Spaces abstraction allows the library to generate metrics at different levels of granularity and export the relative measures into some of the most common data-serialisation formats, such as JSON. In the following source code, we have reported the JSON spaces structure of the *"Hello World"* Rust program. A space type is characterised by its *"kind"* field, which represents its type. In this structure, we define as:

- Outer space: a space containing another space.
- Inner space: a space contained in another space.

In the JSON spaces structure below, for instance, *"main.rs"* is the outer space associated with the whole source code file, while *"main"* is the inner space associated with the program main function.

[2]https://github.com/rodhilton/jasome
[3]https://github.com/mauricioaniche/ck

```
{
    "name": "main.rs",
    "start_line": 1,
    "end_line": 3,
    "kind": "unit",
    "spaces": [
        {
            "name": "main",
            "start_line": 1,
            "end_line": 3,
            "kind": "function",
            "spaces": [],
            "metrics": {...}
        }
    ],
    "metrics": {...}
}
```


We have excluded all dynamic and language-specific metrics from our selection because their implementation would be highly complex and poorly precise. They are born to be computable by dynamic code analysers, which can analyse the program during its execution. At this stage, dynamic information about the runtime behaviour of applications is directly accessible since a compiler has already elaborated the source code. Extracting dynamic data without executing a program would often require emulating what the compiler does, which is a highly complex task for a static code analyser meant to be quick such as RCA. Moreover, an approximate emulation of a compiler's behaviour would produce less accurate results than a dynamic code analyser. Given object-oriented metric relevance in the academic and market fields, we have then established to base our implementation on the *Java* language. If needed, the RCA modular structure allows developers to add these metrics for other programming languages as well since the logic of each metric is confined to a specific Rust module within the library. The selected six metrics are subdivided into three classes: *size metrics*, *object-oriented metrics*, and *security metrics*. Every metric provides some information about distinct software attributes. Size metrics determine the size of a source code or an entire codebase. Object-oriented metrics assess how good the design of a software is according to specific object-oriented programming principles. Lastly, security metrics provide insight into the safety level of a source code.The implementation process has followed a strict review pipeline, including automatic and human controls of changes introduced in RCA. Automated compilation tests on both *Linux* and *Windows* have been performed by an automatic *Continuous Integration (CI)* system. Once the automatic tests are passed, the changes are eventually evaluated by RCA maintainers and improved until the final approval. When every aspect has been positively evaluated, changes are merged into the repository master branch. As a preliminary task, we have also implemented a set of integration tests on the whole library[4]. Moreover, each implemented metric comes with an adequate collection of unit tests to verify the correct metrics computation for the most common cases and some rare corner cases. These unit tests also help new contributors understand how a metric is computed on source code, making a new metric comprehension process more straightforward.

The *Assignments, Branches and Conditions (ABC)* metric is a size metric proposed by Jerry Fitzpatrick in 1997 as a way to count the essential operations performed by a source code [16]. The author proposed the ABC as an alternative code metric to the Halstead and the LOC metrics since, at the time, the Halstead metrics were still highly disputed and not much adopted, while LOC metrics were not objectively defined. The ABC metric is a three-dimensional vector where each component is a counter of three fundamental operations in imperative languages: *storage*, *branching*, and *testing*. The objective is to measure the size of software in terms of the number of basic operations it performs. Imperative programming languages allow the exact counting of the number of basic operations a program performs since they describe exactly how it behaves using statements. In contrast, declarative programming languages hide low-level operations and only describe an application by expressing its expected results. The ABC can also be represented by the magnitude of the vector using the following formula:

$$ABC = \sqrt{A^2 + B^2 + C^2}$$

As a reference for our implementation, we used the source code of *GMetrics*[5], an open-source static code analyser for source codes written in *Groovy*: a programming language alternative to Java for the Java platform. The project source code and its many tests provided a better understanding of how the ABC metric should be computed for the most common cases. We also have added specific implementation details to the metric definition in order to cover unique and complex use cases[6]: a common practice when applying software metrics to real complex scenarios. In particular, for example, we have avoided counting *constant declarations* as assignment operations, as clearly stated by the ABC definition. On the other hand, since the original paper has not specified further indication, we have introduced our mechanism for counting *unary conditional expressions* as conditions. This decision has allowed us to improve the metric precision and detect additional conditions. The idea is to increment the conditions counter with expressions proven to be boolean based on their location, even without knowing their runtime type. With it, we can count the following elements without resorting to advanced dynamic analysis techniques to resolve types:

- Conditions inside other conditions
- Conditions preceded by a *"not"* boolean unary operator inside:
  - Method invocation arguments
  - Assignment operations
  - Returned values in *return* statements

[4]https://github.com/mozilla/rust-code-analysis/pull/724

[5]https://github.com/dx42/gmetrics

[6]https://github.com/mozilla/rust-code-analysis/pull/836

- Returned values in implicit *return* statements in lambda expressions
- Ternary expressions elements

The *Weighted Methods per Class* or *Weighted Method Count (WMC)* is an object-oriented metric announced by Chidamber and Kemerer in 1994 as one of the metrics of the CK object-oriented suite [17]. The WMC was born to measure the complexity of object-oriented source code and can be computed using different complexity measures. However, the most common way to calculate the WMC, and also the way we have chosen to implement it[7], relies on *Cyclomatic Complexity (CC)*, as defined by McCabe [18]. The general metric formula defines the WMC as the sum of CCs of the methods declared in a class:

$$WMC = \sum_{i=1}^{n} CC_i$$

Despite the WMC being the sum of the CCs of the methods defined inside a class, the two metrics have two notable key differences. Firstly, the WMC is associated with classes, so its value cannot propagate from inner spaces to outer spaces in the same way as it happens for the CC. Secondly, the CC for a class takes into account all the complexity enclosed inside a class declaration, including any complexity coming from class *properties initialisation*, such as when using a ternary operator to initialise a class attribute, for example. Instead, the WMC only considers the complexity of the declared methods. Since RCA already computes the CC, our implementation makes use of its value and carefully propagates it from inner to outer spaces only when requested. This happens in the case of a local inner class, i.e. when a class is declared inside another class method.

The *Number of Public Methods (NPM)* and the *Number of Public Attributes (NPA)* are two object-oriented metrics which count respectively the number of public methods and the number of public attributes declared inside a class [19]. We refer to NPM and NPA also as *visibility metrics* because they measure the exposure of class members to the outside classes. The Java language defines three levels of accessibility for class methods and attributes: *public*, *protected* and *private*. Private members are only accessible inside the class, protected members are accessible from subclasses and classes in the same package, and lastly, public members are accessible from everywhere. Public class members are, therefore, the most exposed members of a class in Java. Specifically, a high NPM value for a class may be used as an indicator for two kinds of problems:

- A class has many responsibilities and is performing too many tasks, thus making the class very complex, as in the case of a *utility* class. In this scenario, it can be helpful to check the WMC value of the class to understand whether it is genuinely complex.
- A class presents high coupling toward the other classes of the project. *Class coupling* defines the strength of the relationships between classes and should be reduced to improve class maintainability and reusability [20].

Similar observations can also be made for the NPA metric. Therefore, having classes with low NPA inside a project can be a reasonable strategy to reduce complexity and coupling and increase data safety within classes. Having many public attributes in a class, in fact, could mean that a class is performing many complex tasks, and if it is relying on other classes, it is also exposing many of its data to external security threats. Our implementations for the NPM[8] and NPA[9] metrics deal with a specific propagation of the metrics from inner to outer spaces similar to the one introduced for the WMC. Alongside the NPM and NPA measures, we have also included the counts of the total *Number of Methods (NM)* and total *Number of Attributes (NA)* of a class. These two metrics are also reported in the literature, respectively, as NOM and NOA [21]. Still, we have decided to use the two-letter acronyms since RCA already contains an implementation for the NOM metric. The difference between the two RCA implementations of the NOM and NM metrics is that the NOM acts on all types of spaces, while the NM is computed only for spaces having the field *"kind"* set to class or interface. All the included measures have then been reported and split among classes and interfaces to give the end user more detailed information.

The *Class Operation Accessibility (COA)* and the *Class Data Accessibility (CDA)* metrics are an adaptation of the *Classified Operation Accessibility* (also abbreviated with *COA*) and *Classified Class Data Accessibility (CCDA)* metrics for methods and attributes which are not classified, which means they do not need unconventional security protection [12]. We propose these two adaptations of two existing security metrics because the two original definitions strongly rely on the concepts of classified methods and classified attributes, which are rare to find in mainstream codebases and are concepts related to a specific representation of software called *UMLsec*. UMLsec is an extension of the Unified Modelling Language (UML) which allows the integration of security information inside a traditional UML diagram. It is employed to design secure software applications and control better security characteristics such as confidentiality, access control, and information flow. A *classified attribute* is a class property defined as secrecy in the UMLsec. A *classified method* is a class method interacting with at least one classified attribute. Classified Operation Accessibility (COA) is the ratio of the number of classified public methods to the number of classified methods in a class. Classified Class Data Accessibility (CCDA) is the ratio of the number of classified class public attributes to the number of classified attributes in a class. These two metrics aim to discover security issues at an early stage by allowing software designers to compare the security of program designs and can be expressed with the following formulas:

$$COA = \frac{CPM}{CM} \qquad CCDA = \frac{CCPA}{CA}$$

[7]https://github.com/mozilla/rust-code-analysis/pull/807

[8]https://github.com/mozilla/rust-code-analysis/pull/857

[9]https://github.com/mozilla/rust-code-analysis/pull/861

Where *CPM* is the number of Classified Public Methods declared in the class, *CM* is the total number of Classified Methods declared in the class, *CCPA* is the number of Classified Class Public Attributes, and *CA* is the number of Classified Attributes declared in the class. To make the metrics more beneficial for the average programmer, not used to managing classified class members, we have adapted the original two definitions to all methods and attributes, whether classified or not. The two new definitions provide the user with two new metrics to perform a security evaluation of a class in terms of exposed operations and data, effectively assessing the attack surface of an object of that class. As a reference, we can derive and calculate the two new metrics using the formulas:

$$COA = \frac{NPM}{NM} \qquad CDA = \frac{NPA}{NA}$$

Where *NPM* is the Number of Public Methods, *NM* is the Number of Methods, *NPA* is the Number of Public Attributes, and *NA* is the Number of Attributes. The implementation of the COA and CDA in RCA has been integrated inside the NPM and NPA modules, which provided the NPM and NPA measures and the counts of total methods and attributes per class: the only inputs necessary to compute these two metrics.

IV. Methodology

To test out implementations, we have applied our metrics on some well-known and maintained Java projects of different sizes chosen from the most popular projects, at the time of the research, of diverse categories of application available on the *Maven Central Repository* which we have reported in Table I. We have decided to collect data from Java codebases because of the considerable popularity of the programming language and its numerous modern applications, both web and mobile, in many industrial and commercial fields. The Maven Central Repository is a public and reliable collection of repositories accessed by the Apache Maven build automation and dependency management tool for resolving Java dependencies and building Java projects. By creating a Maven project, developers can import external Java libraries into their projects by adding the corresponding dependency link to the Maven Central Repository. The repository contains many popular Java projects used by programmers worldwide and is continuously maintained. Using those repositories as inputs, our *metrics analysis*[10] has been conducted in two parts:

- *spatial analysis*: An analysis focused on project size, performed on measures obtained from repositories of various sizes pinned to a specific version.
- *temporal analysis*: An analysis focused on project evolution over time, performed on metric values computed from several versions of the same codebases.

We have conducted both analyses using metrics data produced by RCA, obtained from the selected repositories. The raw measures have then been further elaborated and reported into graphs and tables using custom *Python* scripts. The Python

[10]https://github.com/marco-ballario/metrics-analysis

TABLE I
Repositories selected for the analysis.

Name	Size	Version	Java Files
mockito	Very large	4.7.0	949
spring-kafka	Large	2.9.0	502
gson	Medium	2.9.1	218
Java-WebSocket	Medium	1.5.3	175
java-jwt	Small	4.0.0	75
FastCSV	Small	2.2.0	39

visualisation package, *Matplotlib*, has then been employed to produce all graphs of the two analyses. To produce the graphs and tables of the spatial and temporal analyses starting from the RCA library, we have developed three different Python scripts, each of them with a specific purpose:

1) *data-production.py*
2) *spatial-analysis.py*
3) *temporal-analysis.py*

The *data-production.py* script automatically fetches all repository data from the internet, using *Git*, downloads the RCA library, and calls the RCA CLI over all fetched codebases source codes in order to produce JSON files containing the measures needed for the spatial and temporal analysis. The *spatial-analysis.py* script computes cumulative sums, maximums and average values from the JSON files produced by RCA. Cumulative minimums values have not been computed. All the implemented metrics are, in fact, designed to highlight anomalies with high values, and low metric values are, in this case, not considered an issue. The Python script then uses these three cumulative values to create spatial analysis graphs. The code computes the cumulative values in the following way:

$$sum_c(m_f, n) = \sum_{f=1}^{n} m_f$$

$$max_c(m_f, n) = \max(m_1, \cdots, m_n)$$

$$avg_c(m_f, n) = \frac{1}{n} \sum_{f=1}^{n} m_f$$

In the previous formulas, n is the total number of JSON files analysed for a codebase c, and m_f is the metric value for the file f of the same codebase, read from the outermost space in the JSON file. The *temporal-analysis.py* script produces graphs and tables using information about classes and interfaces defined in the projects, in addition to the same data computed by the previous script. Class and interface names, along with their relative metrics values, are extracted from the JSON measures files generated by RCA and stored in specific collections. The script computes this information by applying the previous formulas and the following one:

$$m_c(m_s, m_t, n) = m_s - \sum_{t=1}^{n} m_t$$

In this additional formula, m_c is the true class metric value, m_s is the metric value of the class read from its relative JSON

file space of *"kind"* class, n is the number of *terminal classes* contained in the space of *"kind"* class considered, and m_t is the metric value of the terminal class t. In this context, we define a terminal class as a class that does not contain any additional class declarations inside its declaration body. So a space of *"kind"* class related to a terminal class already contains its true class metric value. This formula is valid for both classes and interfaces for WMC, NPM and NPA. Class or interface ABC is already available in spaces of those kinds. Class or interface COA and CDA are computed using their respective class or interface NPM, NM, NPA, and NA. If a class or interface has NM or NA equal to zero, its COA or CDA is set to zero to avoid introducing infinite values.

V. Results

For the spatial analysis, we have produced three different types of bar graphs, each focused on various metrics and aspects of the measured codebases:

1) Cumulative metrics bar graphs
2) Metric comparisons bar graphs
3) Visibility metrics bar graphs

In the *cumulative metrics bar graphs*, for each codebase, we have represented the three cumulative measures discussed in the previous section: cumulative sum, maximum metric value across every measure file, and cumulative average. Figure 2 illustrates this type of bar graph for the ABC metric. The *metric comparisons bar graphs* compare some of our metrics with some of those already implemented in RCA. The metrics visualised are the ones that share the most similarities between them in terms of objectives and usage. The purpose is to demonstrate the reliability of our metrics and analyse their behaviour in comparison to similar existing trustworthy metrics. They are reported below in Figure 3 and Figure 4. The *visibility metrics bar graphs* display the visibility properties of class measures to show how repositories of different sizes can manage the visibility of their operations and data. Figure 5 shows the percentage of public methods over each declared method in a codebase. Figure 6 shows the percentage of public attributes over each declared attribute in a codebase.

For the temporal analysis, we have produced tables, line and bar charts to visualise the measures of the newly implemented metrics on the selected repositories over time. For this analysis, we have generated:

1) Cumulative metrics line charts
2) Files and classes rankings tables
3) Metrics thresholds bar graphs

The *Cumulative metrics line charts* report the trend of averages and maximums cumulative measures for each version of the considered codebase to visualise how metrics evolve in time within a codebase. Figure 7 shows that the size and complexity of average cumulative measures present a similar trend over time for the *Java JWT* project. Figure 8 displays, instead, the trend of average cumulative visibility measures for the *Java JWT* project. The *Files and classes rankings tables* show the top three files and classes with the highest metrics values over

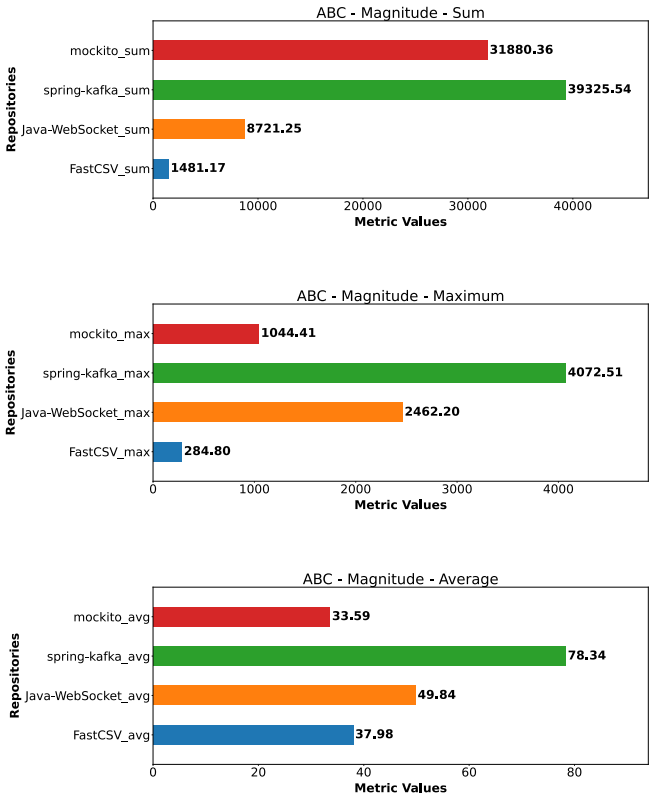

Fig. 2. ABC cumulative measures.

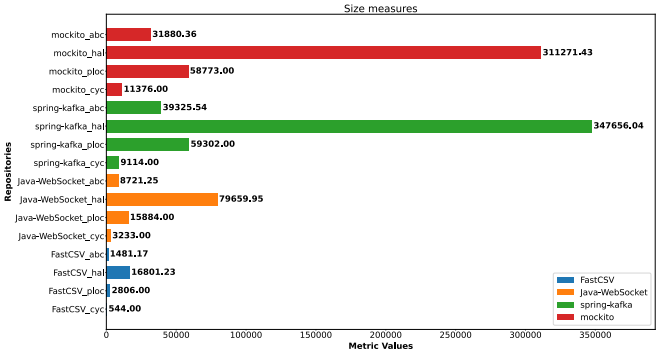

Fig. 3. Size measures comparison.

time. Table II shows the most complex classes for the *Java JWT* project for what concerns the WMC metric. Table III illustrates the classes with the highest NPM present in the *Java JWT* project. The column on the left contains the class name. The columns on the right contain the metric values for each version or range of versions of the codebase. The *Metrics thresholds bar graphs* illustrate the number and percentage of files and classes which exceed each metric's known thresholds over time. Figure 9 shows the number of source code files exceeding the ABC threshold for the *Spring for Apache Kafka* project, which is increasing over time. Similarly, Figure 10 displays the same information as percentages.

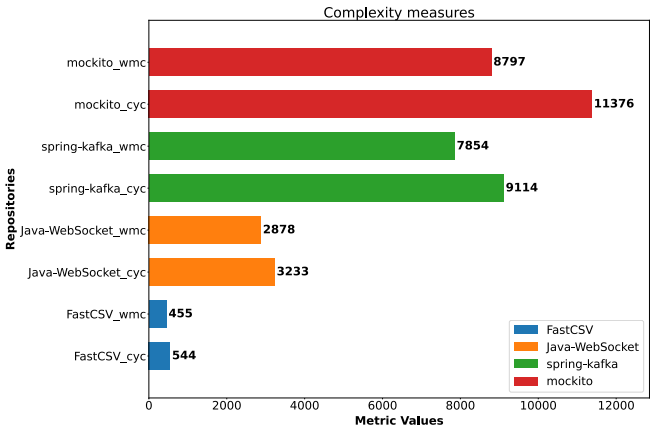

Fig. 4. Complexity measures comparison.

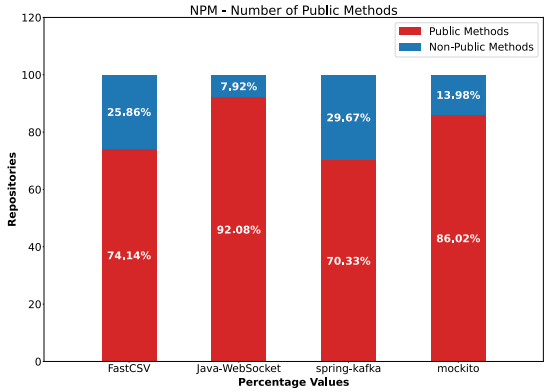

Fig. 5. Percentages of public methods.

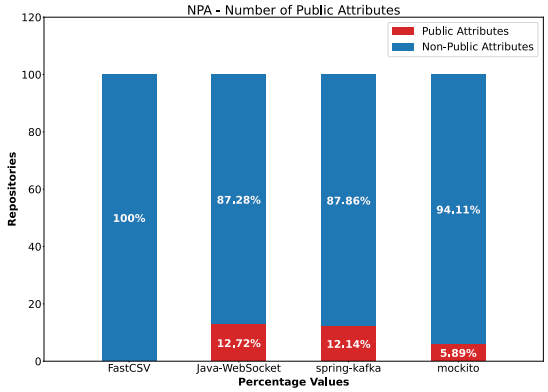

Fig. 6. Percentages of public attributes.

TABLE II
TOP 3 *Java JWT* CLASSES WITH HIGHEST WMC VALUES.

Class	v1 - v8	v9	v10
BaseVerification			86
ECDSAAlgorithmTest	101	112	102
ECDSABouncyCastleProviderTests	87	87	
JWTVerifierTest	79	79	94

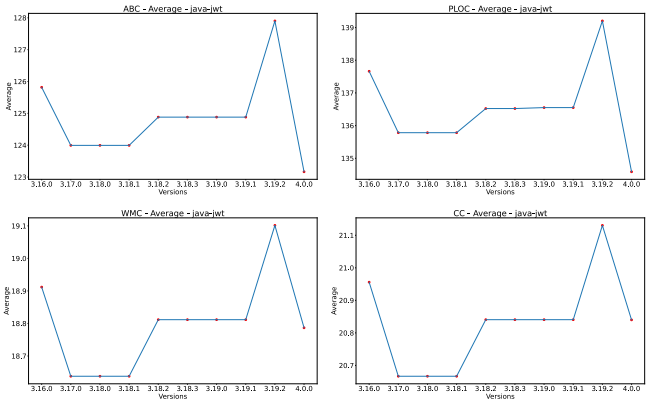

Fig. 7. *Java JWT* average measures - Part 1.

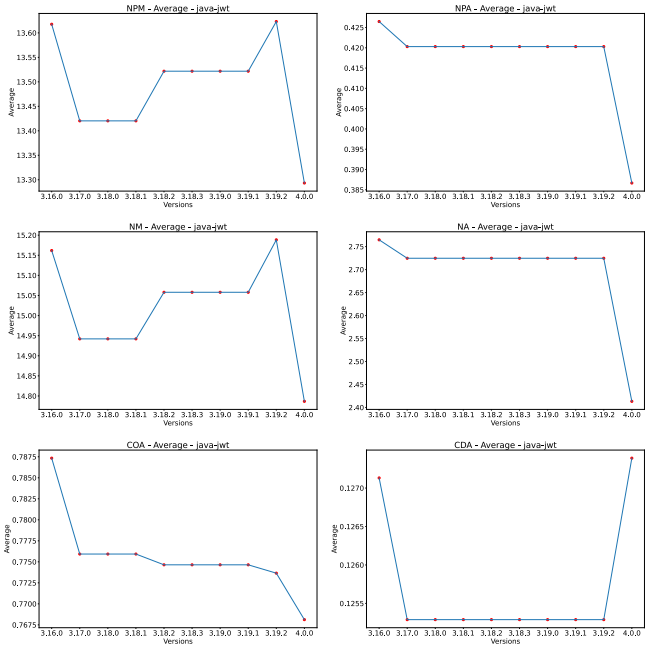

Fig. 8. *Java JWT* average measures - Part 2.

TABLE III
TOP 3 *Java JWT* CLASSES WITH HIGHEST NPM VALUES.

Class	v1 - v8	v9	v10
ECDSAAlgorithmTest	84	91	81
ECDSABouncyCastleProviderTests	84	84	74
JWTVerifierTest	77	77	92

VI. DISCUSSION

The graph in Figure 2 shows that the ABC metric can be used to assess a codebase size by counting the number of fundamental operations it performs (**RQ2**). We can notice that the ABC cumulative sum for the *Mockito* project is lower than the *Spring for Apache Kafka* project one, despite *Mockito* having more source code files than *Spring for Apache Kafka* (**RQ3**). Furthermore, the ABC cumulative average for *Mockito* is smaller than that of *Spring for Apache Kafka* repository. These measures indicate that, considering the ABC metric

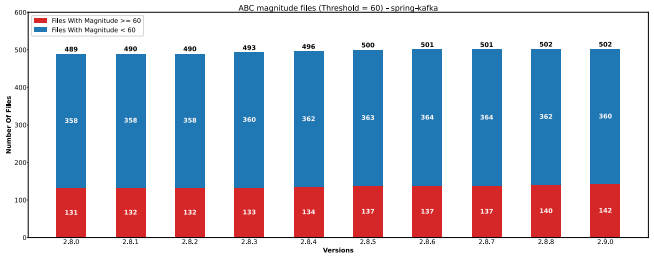

Fig. 9. ABC threshold measures for *Spring for Apache Kafka*.

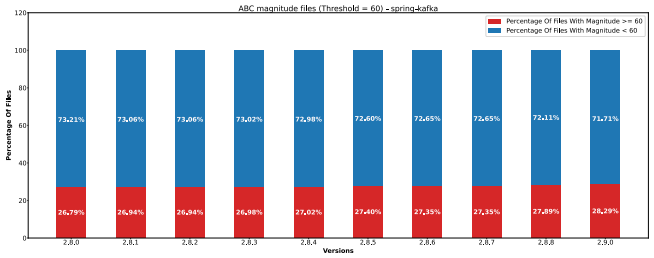

Fig. 10. ABC threshold percentages for *Spring for Apache Kafka*.

definition, the *Mockito* codebase is better managed than the *Spring for Apache Kafka* one. Despite having more files, *Mockito* source code files perform fewer operations, making the codebase less complex and more maintainable. The graph in Figure 3 proves that the ABC metric is, in fact, a valid size metric for a codebase with a determined number of lines of code (**RQ1**). We can assert that because the ABC cumulative sum value bar behaves similarly to other metrics bars, which implies a relation of direct proportionality between the reported size metric, including SLOC, a metric which counts the number of lines in a code. A similar aspect is noticeable in Figure 4, which compares WMC cumulative sums with their respective CC cumulative sums in the codebase. However, this picture shows that WMC cumulative sums bars are always lower than the CC ones. As already explained in a previous section, the WMC shows the complexity of a codebase is only based on the complexity of class and interface methods without considering some details that the CC includes (**RQ2**). The graphs in Figure 5 and Figure 6 allow us to discuss the behaviour of our implemented visibility metrics. A significant NPM value may indicate a large attack surface area for security threats and high complexity. Similarly, a significant NPA value may indicate the number of data the application needs to protect and its overall internal complexity. The NPM and NPA measures are unrelated to the codebase size (**RQ1**). The graphs show that the NPA percentage values are far smaller than the NPM ones, and this is a sign of particular care for safe class data management and less consideration for the safety of class operations (**RQ2**). Among all other implemented metrics, NPM and NPA, and consequently the derived COA and CDA, are certainly important metrics to assess the security of a project because they directly measure the exposure of class operations and class data outside a class (**RQ4**). However, the security context is particularly relevant when managing class operations and data visibility metrics.

The four graphs represented in Figure 7 display some interesting similarities. The *Java JWT* repository is more maintainable for versions where the four metrics assume their lowest values because the codebase is both small and, therefore, not very complex. Cyclomatic Complexity can, in fact, also be interpreted as an advanced technique to measure source code size since it essentially counts the number of linearly independent paths in a source code (**RQ2**, **RQ3** and **RQ4**). To interpret the six graphs reported in Figure 8, it is important to note that COA and CDA trends cannot be derived from NPM, NM, NPA, and NA graphs since these four metrics report aggregated values relative to all project files and, thus, unrelated to individual classes. These graphs determine a way to comprehend class data and operation security and operation over time (**RQ2**) and show that the NPM and NM average values and NPA and NA average values of the project follow a similar trend. This aspect may indicate that most of the methods and attributes declared in the project are public. We can also deduce that programmers have performed more changes on method declarations than attribute declarations since the number of source code files does not vary much over time. COA and CDA average values are more difficult to evaluate, and it is essential to inspect them with the aid of the metrics on which they depend. The tables Table II and Table III report classes or interfaces with exceptionally high metric values over different project versions. Developers can focus on specific files and classes with high metric values to understand the cause of those anomalies, mitigate them, and consequently better maintain a repository (**RQ3**). A developer can also simultaneously consult multiple tables to spot classes with discordant values for more than one metric and evaluate them from different perspectives. For example, in this case, one can notice that the top three classes with high NPM values are also the ones with the highest WMC values and deduce that these classes are very complex, but their methods are not much protected. The graphs in Figure 9 and Figure 10 allow us to reason about metrics thresholds. By setting thresholds, we can identify instances where a particular characteristic of the project deviates from our desired standard. Specifically, the thresholds have been set to the following values (**RQ5**):

- 60 for the ABC, based on previous implementations of the metric in *Ruby* and *Java* and the analysis of empirical results found on two blogs[11][12].
- 34 for the WMC, following a modern study based on a dataset of about one hundred systems [22].
- 40 for the NPM and 10 for the NPA, in accordance with a recent article based on a huge program collection [23].
- 1 for both COA and CDA. Since we have defined and implemented those metrics, after an empirical analysis, we came up with a threshold value of 1 which represents the worst-case value for both of them.

[11]https://jakescruggs.blogspot.com
[12]https://tenpercentnotcrap.wordpress.com

In general, we have selected these thresholds since they come from some of the most recent sources available at the moment of our research and are based on a large number of Java programs with different purposes. Over time, the project has maintained an almost constant percentage of files exceeding the ABC threshold because, along with the number of files exceeding the ABC threshold, the total number of files in the project has also slightly increased over time.

Considering a possible theoretical real-case scenario for RCA usage, on one hand, this software can be integrated into an IDE as a front-end plugin with the aim of providing information which can increase code quality during the code-writing process, while on the other hand, RCA can be inserted as an initial step of a Continuous Integration workflow, with the precise purpose of uniforming the code added by different programmers to a determined quality level decided by project maintainers. Therefore, the results provided by RCA metrics become of fundamental importance in reviewing programmers' contributions. As a practical real-case scenario instead, RCA metrics have been used as inputs for machine learning algorithms, specifically for those models which attempt to estimate a patch risk which is going to be merged into a codebase, and in this case, the Firefox codebase [24]. The innovative insights of this article are not mainly based on the implementation of these metrics but on the possibility of having an open-source software which computes them in an acceptable amount of time on repositories of various sizes. Furthermore, defining a way to interpret these values from a developer's perspective provides the possibility to control their outcomes, hence changing the code accordingly. Having a rapid execution also allows a developer to obtain an immediate feedback on the code quality aspect highlighted by these metrics.

VII. Conclusions

In conclusion, we conducted a comprehensive research on the state-of-the-art of various software metrics from both academic and market perspectives. Subsequently, we implemented six Java metrics into RCA, a Mozilla static analyser written in Rust. As next step, we applied these metrics to actively maintained Java repositories and analysed the obtained results, over space and time, providing answers to the proposed research questions and emphasising their potential usage. We also provided some theoretical and practical real-case scenarios in which a developer might adopt to integrate RCA in a project, starting from a quick code quality analyser until its usage within the Mozilla ecosystem. For future work, it could be worth extending these metrics to additional object-oriented languages, such as C++ and Python, and since RCA is an open-source project, other contributors could decide to take up the task. Another possible improvement could be to expand our research on a greater amount of repositories, hence providing a quantitative, and not only a qualitative, analysis. This work was partially supported by the SIFIS-Home project, funded by the H2020 program of the European Union (Grant Agreement No. 952652), within the context of WP2 Guidelines and Procedures for System and Software Security and Legacy Compliance, Task 2.2 Dynamic Code Quality/Security Evaluation.

References

[1] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[2] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[3] A. L. Timóteo, A. Álvaro, E. S. De Almeida, and S. R. de Lemos Meira, "Software metrics: A survey," *Sl: sn*, 2008.

[4] E. Lebanidze, "The need for fourth generation static analysis tools for security–from bugs to flaws," in *Application Security Conference*, 2008.

[5] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "rust-code-analysis: A rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, 2020.

[6] L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A tool-based perspective on software code maintainability metrics: A systematic literature review," *Scientific Programming*, vol. 2020, pp. 1–26, 2020.

[7] J. Kumar Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of computer science and technology*, vol. 25, pp. 1016–1029, 2010.

[8] R. Ponnala and C. Reddy, "Object oriented dynamic metrics in software development: A literature review," *International Journal of Applied Engineering Research*, vol. 14, no. 22, pp. 4161–4172, 2019.

[9] L. H. Rosenberg and L. E. Hyatt, "Software quality metrics for object-oriented environments," *Crosstalk journal*, vol. 10, no. 4, pp. 1–6, 1997.

[10] R. S. Chhillar and S. Gahlot, "An evolution of software metrics: a review," in *Proceedings of the International Conference on Advances in Image Processing*, 2017, pp. 139–143.

[11] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proceedings of the fourth international workshop on Software engineering for secure systems*, 2008, pp. 57–64.

[12] B. Alshammari, C. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 11–20.

[13] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Software design metrics for object-oriented software." *J. Object Technol.*, vol. 6, no. 1, pp. 121–138, 2007.

[14] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 100–109.

[15] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for formal verification," in *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003. Proceedings 12*. Springer, 2003, pp. 111–125.

[16] J. Fitzpatrick, "Applying the abc metric to c, c++, and java," *More c++ gems*, pp. 245–264, 1997.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[18] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[19] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[20] J. Al Dallal, "Object-oriented class maintainability prediction using internal quality attributes," *Information and Software Technology*, vol. 55, no. 11, pp. 2028–2048, 2013.

[21] D. Wu, L. Chen, Y. Zhou, and B. Xu, "A metrics-based comparative study on object-oriented programming languages," *State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China, DOI reference number*, vol. 10, 2015.

[22] T. G. Filó, M. Bigonha, and K. Ferreira, "A catalogue of thresholds for object-oriented software metrics," *Proc. of the 1st SOFTENG*, pp. 48–55, 2015.

[23] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[24] M. Böck, "Machine learning for interactive performance prediction," *Thesis - Diploma Thesis*, p. 74, 2022.