SoftwareX 12 (2020) 100635

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes

Luca Ardito [a,*], Luca Barbato [b], Marco Castelluccio [c], Riccardo Coppola [a], Calixte Denizet [c], Sylvestre Ledru [c], Michele Valsesia [a]

[a] Control and Computer Engineering Dept., Politecnico di Torino, Italy
[b] Luminem, Italy
[c] Mozilla Corporation, United States of America

ARTICLE INFO

Article history:
Received 18 September 2020
Received in revised form 19 November 2020
Accepted 19 November 2020

Keywords:
Algorithm
Software metrics
Software maintainability
Software quality

ABSTRACT

The literature proposes many software metrics for evaluating the source code non-functional properties, such as its complexity and maintainability. The literature also proposes several tools to compute those properties on source codes developed with many different software languages. However, the Rust language emergence has not been paired by the community's effort in developing parsers and tools able to compute metrics for the Rust source code. Also, metrics tools often fall short in providing immediate means of comparing maintainability metrics between different algorithms or coding languages. We hence introduce **rust-code-analysis**, a Rust library that allows the extraction of a set of eleven maintainability metrics for ten different languages, including Rust. **rust-code-analysis**, through the Abstract Syntax Tree (AST) of a source file, allows the inspection of the code structure, analyzing source code metrics at different levels of granularity, and finding code syntax errors before compiling time. The tool also offers a command-line interface that allows exporting the results in different formats. The possibility of analyzing source codes written in different programming languages enables simple and systematic comparisons between the metrics produced from different empirical and large-scale analysis sources.

Code metadata

Current code version	v01
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-20-00050
Code Ocean compute capsule	
Legal Code License	Mozilla Public License v2.0
Code versioning system used	git
Software code languages, tools, and services used	Rust
Compilation requirements, operating environments & dependencies	Cargo (Rust package manager)
If available Link to developer documentation/manual	https://mozilla.github.io/rust-code-analysis/index.html https://docs.rs/rust-code-analysis/0.0.17/rust_code_analysis
Support email for questions	

1. Motivation and significance

Software Maintainability is defined as the ease with which a software component, or system, can be modified and improved, corrected from faults, and adapted to changing environment or requirements [1]. Maintainability is a significant factor in software products' economic success that is not easy to estimate for real-world software projects [2].

Software Metrics are measures of software characteristics that are quantifiable or countable, and can serve as an important aid to the estimation of software maintainability. The building of a maintainability prediction model can allow identifying parts of the software to refactor or modify. *Source Code Metrics* are a subset of Software Metrics that focus on measuring properties of

* Corresponding author.
E-mail addresses: luca.ardito@polito.it (L. Ardito), luca.barbato@luminem.it (L. Barbato), mcastelluccio@mozilla.com (M. Castelluccio), riccardo.coppola@polito.it (R. Coppola), cdenizet@mozilla.com (C. Denizet), sledru@mozilla.com (S. Ledru), michele.valsesia@polito.it (M. Valsesia).

https://doi.org/10.1016/j.softx.2020.100635

the source code of a system and mapping them to numerical values [3]. Researchers have been working on the topic of Software Metrics for more than 40 years by proposing, developing, and validating many different metrics, suites and models, and tools to compute them.

Nuñez-Varela et al. [4], in their recent systematic mapping study, listed 300 source code metrics, linking those metrics with the tools that can use them just by analyzing papers published from 2010 to 2015. In our previous work [5], we identified 174 software metrics – among which we identified a set of 15 most commonly mentioned ones – and 19 metric computation tools. None of those tools was able to work with the Rust programming language.

However, the industry and academia still have no accordance on the best options for both metrics and tools to be adopted to evaluate software maintainability. In the industry, the consolidated best practices usually include continuous integration, testing with code coverage measurement and language sanitization (such as static analysis), and sometimes coding style enforcement. However, the lack of adequate and consolidated tooling makes it challenging to have an automated evaluation of the code maintainability [6]. The reliability of the result is another problem that affects this kind of software. Lincke et al. [7] showed that different software metrics tools provided inconsistent results.

Furthermore, the emergence of new programming languages requires new software to compute and evaluate software metrics. Each language requires a dedicated parser and metrics extractor. *Rust* is a recent programming language whose focus is on developing reliable and efficient systems that exploit parallelism and concurrency. Conciseness, expressiveness, and memory safety are among the principal properties that guided the Rust development [8].

This paper introduces **rust-code-analysis** (RCA), a Rust library, to analyze and extract maintainability information from source codes written in many different programming languages. The library is based on a parser generator tool and an incremental parsing library called Tree Sitter.[1]

To the best of our knowledge, the library constitutes the first attempt to develop a Software Metrics extractor with the Rust programming language. The usage of Rust can guarantee several advantages for the development and usage of the library, such as: (i) guaranteed memory-safety and thread-safety, with the possibility of eliminating many classes of bugs at compile-time [9]; (ii) fast and memory-efficient parsing and metrics computation [10]; (iii) easy integration with other programming languages.

At the same time, RCA is the first open-source tool able to compute maintainability software metrics for the Rust language. Hence, it can serve as a valuable aid to evaluate software written in Rust and the programming language itself.

2. Software description

rust-code-analysis is a *Rust* library to analyze and extract information from source code written in the following programming languages: *C++*, *C#*, *CSS*, *Go*, *HTML*, *Java*, *JavaScript*, *Python*, *Rust*, *Typescript*.

Mozilla developed the initial version of this library to support the Firefox development processes. Every month, up to 6000 changes[2] from more than 500 developers can land in its codebase. This library is one of the elements developed to help to evaluate the inherent risk of a change and preventing the introduction of new defects [11], especially at the uplift phase [12,13].

The library is available on GitHub,[3] published on Crates.io[4] and released under the Mozilla Public License v2.0. It can run on the most common platforms (i.e., Linux, macOS, and Windows). The library's dependencies can be managed and built with the standard package manager distributed with the compiler: *Cargo*. Cargo eliminates the need for manual management of large dependency graphs and simplifies building the software, unlike tools written in C/C++. **rust-code-analysis** guarantees the metrics correctness by running unit testing that executes code with well-known characteristics.

For the comparison, we created unit tests considering two main goals:

- verifying the **rust-code-analysis** correctness in typical use cases, i.e., using source code that represents real cases and that can usually be found in a codebase;
- checking the **rust-code-analysis** correctness in extreme cases using unusual parameters (corner case).

The values used for producing the **rust-code-analysis** outputs are obtained by manually calculating the metrics of known source code pieces used as input for the tests. The unit tests described above can be found in the repository.

The library comes coupled with a command-line tool called *rust-code-analysis-cli*, which allows interacting efficiently with the APIs exposed by the library. The command-line tool can also be used to print and export the metrics in different formats, and it can be installed through *Cargo* as well.

The codebase of **rust-code-analysis** can be considered a combination of two loosely-coupled components:

- Language parsers;
- Metrics computation modules.

Ideally, a contributor interested in adding a new language or metric to the codebase could add a new *metric* or *language* to the respective module, leaving the rest of the code untouched. In addition, any developer is allowed to contribute to the development of **rust-code-analysis** through pull requests.

2.1. Features

The **rust-code-analysis** library is mainly thought for developers. Indeed, through its APIs, it is possible to carry out various tasks related to software code metric computation and maintenance analysis:

- Print the Abstract Syntax Tree (AST) of a source code passed as input;
- Use the information extracted from AST nodes to detect in advance possible parsing errors present in the code. For example, it is possible to catch grammar errors before a program is compiled;
- Print a series of Source Code Metrics to evaluate the quality of source code;
- Export metrics in different output formats.

Leveraging the power of Rust, some of the APIs are implemented in a multi-threaded fashion, speeding up considerably the entire computation.

Also, to help users interacting with the APIs in an easy way, a command-line interface called *rust-code-analysis-cli* has been developed. Indeed, through the CLI's commands, a user can visualize the output produced by the APIs on the shell or decide to store it within either text or binary files.

[1] https://github.com/tree-sitter/tree-sitter Last visited 10/06/2020.
[2] https://www.openhub.net/p/firefox.
[3] https://mozilla.github.io/rust-code-analysis/index.html.
[4] https://crates.io/crates/rust-code-analysis.

Table 1
Metrics computed by rust-code-analysis.

Metric	Description
CC	McCabe's cyclomatic complexity: it calculates the code complexity examining the control flow of a program. It is measured as the number of linearly independent paths through a piece of code [14].
SLOC	Source lines of code: it returns the total number of lines in a source file.
PLOC	Physical lines of code: it returns the total number of instruction and comment lines in a source file.
LLOC	Logical lines of code: it returns the total number of logical lines (statements) in a source file.
CLOC	Comment lines of code: it returns the total number of comment lines in a file.
BLANK	it counts the number of blank lines in a source file.
Halstead	The Halstead suite, a set of seven statically computed metrics, all based on the number of distinct operators (*n1*) and operands (*n2*) and the total number of operators (*N1*) and operands (N2) [15]. The suite provides a series of information, such as the effort required to maintain the analyzed code, the size in bits to store the program, the difficulty to understand the code, an estimate of the number of bugs present in the codebase, and an estimate of the time needed to implement the software.
MI	Maintainability index: a suite to measure software's maintainability, calculated both on source files and functions [16].
NOM	Number of Methods: it returns the number of methods in a source file.
NARGS	Number of arguments: it counts the number of arguments of each method in a source file.
NEXITS	Number of exits: it counts the number of possible exit points of each method in a source file.

Metrics

All **rust-code-analysis** metrics are calculated starting from the source code of a program without executing it, i.e., the library only computes *static metrics*. Static metrics allow the evaluation of software quality, discover sections of a source code that might be more difficult to maintain, and compare the difference between programming languages.

The implemented metrics are divided into three main groups:

- *Line* metrics detect the number of lines of a certain kind, such as the number of instructions, comments, and statements in code.
- *Function* metrics count the number of functions and closures in a code. They can also extract further data, such as the number of arguments and exit points of a function.
- *Global* metrics provide a series of information on the effort required to maintain and understand a codebase, including an estimate of the number of bugs or the time needed to implement software. Also, they can evaluate the complexity of a codebase by examining the control flow of a program.

Metrics are computed independently for each function and then merged to determine the general quality of a program.

At the current state of implementation, the tool implements the list of metrics reported in Table 1. The implemented metrics include established suites available in the literature (e.g., the Halstead suite [15]), but also metrics that were specifically defined for the first time in the **rust-code-analysis** tool, such as NARGS and NEXITS.

2.2. Code representation

rust-code-analysis builds, through the use of an open-source library called *tree-sitter*,[5] an Abstract Syntax Tree (AST) in order to represent the syntactic structure of a source file. An AST differs from a Concrete Syntax Tree because it does not include information about the source code less important details, like punctuation and parentheses.

The AST defines a series of *Syntax Nodes* as a basic data structure to inspect the Syntax tree. They are usually used to:

- List every construct present in a codebase;
- Count the number of constructs of a certain kind;
- Detect the number of parsing errors in a program.

Each Syntax node contains a *type* information (the grammar rule of the inspected code that the node represents), in addition to its position in the source code in terms of row/column coordinates. *tree-sitter* supports the distinction between Concrete and Abstract Syntax Trees, by separating named and anonymous nodes, with the former being the only one concurring in the generation of ASTs.

On top of the generated AST, **rust-code-analysis** performs a division of the source code in *spaces*. A space is any structure that incorporates a function. It contains a series of fields such as the name of the structure, the relative line start, line end, kind, and a *metric* object, which is composed of the values of the available metrics computed by **rust-code-analysis** on the functions contained in that space. All metrics computed at the function level are then merged at the parent space level, and this procedure continues until the space representing the entire source file is reached.

Here a list of the space kinds that can be found in source files of different programming languages: *function*, *class* (Java, C++), *struct* (Rust, C, C++), *trait* (Rust), *impl* (Rust), *unit* (all languages), *namespace* (C++).

Table 2
Parameters for the **rust-code-analysis** command line interface.

Parameter	Description
-metrics	Compute and print metrics.
-p	Define the path to a file or a directory.
-O	Select the output format used to export metrics between the following values: *cbor*, *json*, *toml*, *yaml*.
-o	Specify the path for the output file to be saved.
–pr	Enable *json* and *toml* output files to be exported as *pretty-printed*.
-I	Consider only files with a specified extension.
–f	Scan the code in search of all the nodes of a certain type.
–count	Count the number of nodes of a certain type.
-d	Print the entire AST on the shell.
–ls, –le	Consider only the portion of the file between the lines specified by ls and le.
–serve	Make the cli act as a server on the localhost.
–port	Specify the port used by the server.

2.3. Command line interface

rust-code-analysis offers a command-line interface to compute metrics and extract information from source code. The command-line interface can be used by launching the command `rust-code-analysis-cli`, and feeding the proper parameters. The order of parameters is irrelevant for the proper functioning of the program. The list of parameters supported by the command line interface is shown in Table 2.

rust-code-analysis leverages a Rust library, called Serde,[6] to favor data interchange between applications and to export metrics in a machine-parsable and human-readable format. By default, *rust-code-analysis-cli* supports three textual formats: *JSON*,

[5] https://tree-sitter.github.io/.

[6] https://crates.io/crates/serde Last visited 24/06/2020.

TOML, and *YAML*. In addition, to speed up processing and transfer speeds, a binary data serialization format based on JSON has been added: *CBOR*.

3. Illustrative examples

This section reports illustrative examples of some of the main use cases that can be performed by using the **rust-code-analysis** command-line interface, and the output obtained on simple code examples.

To execute a command on single function it is possible to leverage the `ls` and `le` parameters. For instance, to execute **rust-code-analysis** on a single function which starts at line 5 and ends at line 10:

```
rust-code-analysis-cli command --ls 5 --le 10 -p
    path/to/file/or/directory/containing/the/code
```

To print all computed metrics on the screen, the command is the following one:

```
rust-code-analysis-cli --metrics -p
  /path/to/file/or/directory/containing/the/code
```

To export metrics in a specific output format, it is possible to use the following command:

```
rust-code-analysis-cli --metrics -O format -o /
    output/path -p /path/to/file/or/directory/
    containing/the/code
```

Json and *Toml* formats can be exported as pretty-printed. To do so, it is necessary to set the `pr` parameter. Below an example:

```
rust-code-analysis-cli --metrics -O format --pr -o
    /output/path -p /path/to/file/or/directory/
    containing/the/code
```

To know if there are some syntactic errors in the code, it is possible to use the `--f` parameter, by searching for the occurrence of nodes of the *error* type:

```
rust-code-analysis-cli -I "*.ext" -f -error -p /
    path/to/file/or/directory/containing/the/code
```

In addition, it is also possible to count the number of nodes of a certain type using the **count** option:

```
rust-code-analysis-cli -I "*.ext" --count -error -
    p /path/to/file/or/directory/containing/the/
    code
```

The following command allows to print the AST of a given source file, by using the `-d` parameter:

```
rust-code-analysis-cli -d -p /path/to/file/or/
    directory/containing/the/code
```

To use the *rust-code-analysis-cli* as a server running on port 9090, the following command can be used:

```
rust-code-analysis-cli --serve --port 9090
```

Once the server is started, it is possible to ping it through a GET command at the following URL:

```
http://127.0.0.1:9090/ping
```

Finally, to get the metrics from the server instance of the *rust-code-analysis-cli*, the following POST request can be used. In the POST request, `filename` identifies the path to the source file, while `unit` is a boolean value that indicates the tool to include in the output only top-level metrics (1) or all detailed metrics (0).

```
http://127.0.0.1:9090/metrics?file_name={filename
    }&unit={unit}
```

As an example, we report the JSON output that can be obtained by running the *rust-code-analysis-cli* program on a simple code example. The code example (a simple Rust function) is reported in Listing 2. The JSON output is reported in Listing 2.

The following command writes all the computed metrics in a file called *foo.rs.json*, and outputs it in the *tmp* directory.

```
rust-code-analysis-cli -p foo.rs --metrics -O json
    -o /tmp
```

The JSON output contains the subdivision of a source code in different *spaces*. Looking at the code example below, a space has been generated for the whole unit, for the *impl* Foo, for the `bar` *function* and for the closure inside `bar`.

For each space, the metrics are grouped in different JSON items depending on their structure. For example, the metrics obtained counting the Lines of Code present in a source file (SLOC, PLOC, LLOC, CLOC) have been grouped in a JSON array called LOC.

Listing 1: A simple Rust code example

```rust
struct Foo {
  buf: Vec<u8>,
}

impl Foo {
  fn bar(&self) -> usize {
    let s = 0;
    for i in 0..10 {
      s += self
        .buf
        .iter()
        .fold(s, |acc, x| acc + x * i);
    }
    s
  }
}
```

Listing 2: foo.rs.json


```json
{
    "name": "/tmp/foo.rs",
    "start_line": 1,
    "end_line": 16,
    "kind": "unit",
    "spaces": [
        {
            "name": "Foo",
            "start_line": 5,
            "end_line": 16,
            "kind": "impl",
            "spaces": [
                {
                    "name": "bar",
                    "start_line": 6,
                    "end_line": 15,
                    "kind": "function",
                    "spaces": [
                        {
                            "name": "<anonymous>",
                            "start_line": 12,
                            "end_line": 12,
                            "kind": "function",
                            "spaces": [],
                            "metrics": {
                                "nargs": 4.0,
                                "nexits": 0.0,
                                "cyclomatic": 1.0,
                                "halstead": {...},
```

```
                        "loc": {...},
                        "nom": {...},
                        "mi": {...},
                    }
                }
            ],
            "metrics": {
                "nargs": 1.0,
                "nexits": 1.0,
                "cyclomatic": 1.5,
                "halstead": {...},
                "loc": {...},
                "nom": {...},
                "mi": {...},
            }
        }
    ],
    "metrics": {
        "nargs": 0.0,
        "nexits": 1.0,
        "cyclomatic": 1.3333333333333333,
        "halstead": {...},
        "loc": {...},
        "nom": {...},
        "mi": {...},
    }
},
],
"metrics": {
    "nargs": 0.0,
    "nexits": 1.0,
    "cyclomatic": 1.25,
    "halstead": {...},
    "loc": {...},
    "nom": {...},
    "mi": {...},
}
}
```


Table 3
Languages supported by **rust-code-analysis** compared with most popular open source tools according to the literature.

	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco	CBR Insight	Halstead Tool	SonarQube	JSInspect	Escomplex	Eslint	CCFinderX	Ref-Finder	R-C-A
C++				v		v	v	v				v		v
C#						v		v				v		v
CSS								v						v
GO								v						v
HTML								v						v
Java	v	v	v		v	v	v	v				v	v	v
JavaScript						v		v	v	v	v			v
Python						v		v						v
Rust														v
TypeScript								v						v

Table 4
Metrics computed by **rust-code-analysis** compared with most popular open source tools according to the literature.

	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco	CBR Insight	Halstead Tool	SonarQube	JSInspect	Escomplex	Eslint	CCFinderX	Ref-Finder	R-C-A
CC		v	v	v				v		v	v	v		v
LOC		v		v	v	v		v	v		v	v		v
Halstead							v			v				v
MI								v		v				v
NOM				v					v					v
NARGS														v
NEXITS														v

4. Related work

In our previous work we conducted a Systematic Review to find all open-source tools cited in related literature about software metrics. We came up with a set of thirteen open-source tools that were proposed by the software maintenance community and/or used to perform empirical measurements of source codes: CKJM [17], MetricsReloaded [18], CodeMetrics [19], Squale [20], Quamoco Benchmark [21], CBR Insight [22], Halstead Metrics Tool [23], SonarQube [24], JSInspect [25], Escomplex [26], Eslint [27], CCFinderX [28], Ref-Finder [29].

In Table 3, we report the language support offered by **rust-code-analysis**and the languages among those supported by the open-source tools available in the literature. It can be seen that, albeit tools like SonarQube and CBR Insight cover many of the languages with which **rust-code-analysis** works, no tool is compatible with the Rust language.

In Table 4, we report the metrics computed by **rust-code-analysis** and the ones computed by the open-source tools available in the literature. For the sake of readability, we compacted under the *LOC* name all metrics related to the count of the lines of code in a source file, namely SLOC, PLOC, LLOC, CLOC, and BLANK. As shown in the Table, there are no tools that can offer the same combination of metrics computed by **rust-code-analysis**. In particular, two of them do not cover any of the considered metrics. Considering the most complex metric suites (i.e., the Halstead suite and the Maintainability Index), they are both considered by only two open-source tools (and only Escomplex can compute both). As expected, no open-source tools allow the computation of the *NARGS* and *NEXITS* metrics since they have been defined in the scope of the **rust-code-analysis** tool.

In the following, we highlight some **rust-code-analysis** advantages compared with the other tools described in Table 4. In particular, **rust-code-analysis**:

- is modular, so it is easy to create new modules that implement new metrics or add support for more programming languages;
- is written in Rust, so it takes advantage of this programming language's features, described in Section 1.
- is actively maintained by Mozilla. External contributors can implement new features, fix bugs, and then submit pull requests to include their changes in the project.
- is currently used for analyzing *mozilla-central* (the Firefox source code), so it can exploit a big testbed;
- supports multiple programming languages. Implementing metrics for projects written in different programming languages is problematic because developers are required to learn the structure of each language, wasting a significant amount of time;
- provides the binaries for Windows and Linux in a facilitated way by using a continuous integration and deployment service. There is no need to build the code from scratch, making things simpler for end-users. Indeed, building a project sometimes requires struggling with complex configurations, depending on the operating system (and its version) in use, and the libraries installed in the system;
- provides a CLI and a library, which makes it easily integrable into third-party software.

5. Impact and conclusions

In this paper, we presented **rust-code-analysis**, a Rust library to analyze and extract maintainability information from source codes.

rust-code-analysis is a novel software package with two newly defined software maintainability metrics (NARGS and NEXITS). It offers a command-line interface that can be easily called by other software projects to monitor and inspect the quality of their code, either in production or under development. The tool is already in use for Firefox Nightly to evaluate the risk of the codebase changes.[7]

We also expect the following impacts for the **rust-code-analysis** library:

- For software engineers and coders, the tool can evaluate the code maintainability of running projects, quantify its complexity, and inspect possible parsing errors before compile time through an AST analysis.
- For integration in developer workflows, the tool can be integrated to identify risky changes and surface the need for more testing or refactoring.
- For researchers in software engineering, the tool can be used to either conduct empirical studies and compare maintainability properties between implementations of the same algorithm in different languages or implement different algorithms in the same language.

Both use cases can be easily carried out, leveraging the command line interface of the tool. As an example of using the tool for empirical experiments, we already developed a tool in Python that exploits **rust-code-analysis** to compare diverse algorithm implementations written each in a different language.[8]

As an immediate future work, we plan to add CHANGE and Cognitive Complexity metrics and to implement all the metrics for the C# and Java languages.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Mozilla Research, United States of America funded this project with the research grant 2018 H2. The project title is: “Algorithms clarity in Rust: advanced rate control and multi-thread support in rav1e". This project aimed to understand how the Rust programming language improves code maintainability while implementing complex algorithms.

References

[1] IEEE. Standard glossary of software engineering terminology. IEEE Std 610.12-1990 1990;1–84. http://dx.doi.org/10.1109/IEEESTD.1990.101064.
[2] Kaur A, Kaur K, Pathak K. Software maintainability prediction by data mining of software code metrics. In: 2014 international conference on data mining and intelligent computing (ICDMIC). 2014, p. 1–6. http://dx.doi.org/10.1109/ICDMIC.2014.6954262.
[3] Lanza M, Marinescu R. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media; 2007.
[4] Nuñez-Varela AS, Pérez-Gonzalez HG, Martínez-Perez FE, Soubervielle-Montalvo C. Source code metrics: A systematic mapping study. J Syst Softw 2017;128:164–97. http://dx.doi.org/10.1016/j.jss.2017.03.044, http://www.sciencedirect.com/science/article/pii/S0164121217300663.
[5] Ardito L, Coppola R, Barbato L, Verga D. A tool-based perspective on software code maintainability metrics: a systematic literature review. Sci Program 2020;2020:5284645. http://dx.doi.org/10.1155/2020/8840389.
[6] Sarwar M, Tanveer W, Sarwar I, Mahmood W. A comparative study of MI tools: Defining the roadmap to MI tools standardization. In: 2008 IEEE international multitopic conference. 2008, p. 379–85. http://dx.doi.org/10.1109/INMIC.2008.4777767.
[7] Lincke R, Lundberg J, Löwe W. Comparing software metrics tools. In: Proceedings of the 2008 international symposium on software testing and analysis. ACM; 2008, p. 131–42.
[8] Matsakis ND, Klock FS. The rust language. ACM SIGAda Ada Lett 2014;34(3):103–4.
[9] Balasubramanian A, Baranowski MS, Burtsev A, Panda A, Rakamarić Z, Ryzhyk L. System programming in rust: Beyond safety. In: Proceedings of the 16th workshop on hot topics in operating systems. 2017, p. 156–61.
[10] Xu H, Chen Z, Sun M, Zhou Y. Memory-safety challenge considered solved? An empirical study with all rust CVEs. 2020, arXiv preprint arXiv:2003.03296.
[11] Nayrolles M, Hamou-Lhadj A. CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In: Proceedings of the 15th international conference on mining software repositories. New York, NY, USA: Association for Computing Machinery; 2018, p. 153–64. http://dx.doi.org/10.1145/3196398.3196438.
[12] Castelluccio M, An L, Khomh F. Is it Safe to Uplift this Patch?: An Empirical Study on Mozilla Firefox. In: 2017 IEEE international conference on software maintenance and evolution (ICSME), 2017. p. 411–21.
[13] Castelluccio M, An L, Khomh F. An empirical study of patch uplift in rapid release development pipelines. Empir Softw Eng 2018;1–37.
[14] Ebert C, Cain J, Antoniol G, Counsell S, Laplante P. Cyclomatic complexity. IEEE Softw 2016;33(6):27–9.
[15] Halstead MH. Elements of software science. Operating and programming systems series, New York, NY, USA: Elsevier Science Inc.; 1977.
[16] Welker KD. The software maintainability index revisited. CrossTalk 2001;14:18–21.
[17] Kaur A, Kaur K, Pathak K. A proposed new model for maintainability index of open source software. In: Proceedings of 3rd international conference on reliability, infocom technologies and optimization. IEEE; 2014, p. 1–6.
[18] Saifan AA, Alsghaier H, Alkhateeb K. Evaluating the understandability of android applications. Int J Soft Innov (IJSI) 2018;6(1):44–57.
[19] Kaur A, Kaur K, Pathak K. Software maintainability prediction by data mining of software code metrics. In: 2014 international conference on data mining and intelligent computing (ICDMIC). IEEE; 2014, p. 1–6.
[20] Ludwig J, Xu S, Webber F. Compiling static software metrics for reliability and maintainability from github repositories. In: 2017 IEEE international conference on systems, man, and cybernetics (SMC). IEEE; 2017, p. 5–9.
[21] Wagner S, Lochmann K, Heinemann L, Kläs M, Trendowicz A, Plösch R, et al. The quamoco product quality modelling and assessment approach. In: 2012 34th international conference on software engineering (ICSE). IEEE; 2012, p. 1133–42.
[22] Ludwig J, Cline D. CBR insight: measure and visualize source code quality. In: 2019 IEEE/ACM international conference on technical debt (TechDebt). IEEE; 2019, p. 57–8.
[23] Hariprasad T, Vidhyagaran G, Seenu K, Thirumalai C. Software complexity analysis using halstead metrics. In: 2017 international conference on trends in electronics and informatics (ICEI). IEEE; 2017, p. 1109–13.
[24] Sarwar MI, Tanveer W, Sarwar I, Mahmood W. A comparative study of MI tools: Defining the roadmap to MI tools standardization. In: 2008 IEEE international multitopic conference. IEEE; 2008, p. 379–85.
[25] Chatzidimitriou K, Papamichail M, Diamantopoulos T, Tsapanos M, Symeonidis A. Npm-miner: An infrastructure for measuring the quality of the npm registry. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR). IEEE; 2018, p. 42–5.
[26] GitHub I. escomplex. 2017, https://github.com/escomplex/escomplex,
[27] Tómasdóttir K, Aniche M, Van Deursen A. The adoption of javascript linters in practice: A case study on eslint. IEEE Trans Softw Eng 2020;46(8):863–91. http://dx.doi.org/10.1109/TSE.2018.2871058.
[28] Matsushita T, Sasano I. Detecting code clones with gaps by function applications. In: Proceedings of the 2017 ACM SIGPLAN workshop on partial evaluation and program manipulation; 2017. p. 12–22.
[29] Kádár I, Hegedus P, Ferenc R, Gyimóthy T. A code refactoring dataset and its assessment regarding software maintainability. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol. 1. IEEE; 2016, p. 599–603.

[7] https://github.com/mozilla/gecko-dev.
[8] https://github.com/SoftengPoliTo/SoftwareMetrics.