Software Metrics and Multi-lingual Code: A Preliminary Study Based on Java and C

Uroš Zarić[1,*], Gordana Rakić[1]

[1]University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics

Abstract

This paper represents a cross-language and cross-tool investigation of tools and techniques for static code analysis focused on basic software complexity metrics such as: McCabe Cyclomatic Complexity, Halstead, and Maintainability Index. The main goal of the research is to determine whether metric differences between multilingual implementation codes exist and how disparate the outcomes of used static tools are. Furthermore aim to learn about the reasons for the discovered differences. To perform this experiment, we use similar multilingual code samples of the RossetaCode project and identical multilingual code samples (SumProd) developed as cloning scenarios for the evaluation of cross-language clone detection. Because of the limited language support by tools and metrics, our experiment uses widespread C and Java programming languages. Our study will benefit from the selection of these two input languages as they have very similar syntax which enables us to write and pick equivalent code snippets in the two languages and their easier comparison. Our research is based on commercial and open-source static analysis tools. The final results provide divided opinions in cross-language analysis and agreement about the existence of absolute inconsistency in cross-tool metric values. More precisely, on the observed heterogeneous multilingual RossetaCode samples C code appears to be more complex, while observed homogeneous multilingual SumProd samples suggest that implementations in Java are more complex. The cross-tool review releases absolute inconsistency of metric values over all covered open-source and commercial static tools. Finally, a manual check of in-tool outcomes signalizes that the roots of inconsistencies are in the tool implementations that do not appropriately follow metric calculation formulas and algorithms.

Keywords

Software metrics, computer languages, cross-language analysis, multi-lingual projects, Java, C

1. Introduction

The design of a software system represents the most challenging and important phase in a software life-cycle. It influences other aspects like the complexity of development organization and necessary maintainability factors. Software metrics provide a countable and comparable way of measuring the quality of software systems. There are two main approaches to calculating metric values, static analysis of written code without its execution and dynamic analysis during program execution [1]. In this paper, we consider the static method with a focus on multilingual codes and cross-tool comparison. Code datasets include chosen and prepared samples from RossetaCode [2] and SumProd snippets used to test cross-language clone detection for LICCA project [3, 4]. We have analyzed three common complexity metrics: McCabe [5], Halstead [6], and Maintainability Index [7]. Our experiment considers many available commercial and open-source tools and also advocates manual checks of their results.

The goals of our research can be formulated in three questions:

- **RQ1.** How two programming language implementations are disparate in terms of metric values?
- **RQ2.** How metric outcomes of different static analysis tools are disparate?
- **RQ3.** Are answers on RQ1 and RQ2 still valid when observing identical code snippets implemented in different languages?

SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9–11, 2024, Novi Sad, Serbia

*Corresponding author.

zaricu22@gmail.com (U. Zarić); gordana.rakic@dmi.uns.ac.rs (G. Rakić)

0000-0001-7366-5159 (G. Rakić)

The paper is organized into the following sections: The second section briefly introduces a reader to the history of this research field and its influence on our motivation in the form of a short related work. The section on the research design introduces the selection of code samples, tools, and metrics, as well as a description of the applied method. The fourth section presents the experimental results and noticed metric relationships. The last two sections state the experimental limitations, conclusions and suggestions for future work.

2. Motivation and related work

One of the open research questions in the software engineering field is related to applicability of software metrics in practise and weaknesses of software metrics techniques and tools for their wider usage [8, 1]. Some of the reasons highlighted in the literature are weak coverage of metrics sets and input computer language by available tools [9].

While source code metrics are well defined, their thresholds among languages are compared and derived [10], and are often used for evaluation and comparison of various aspects of different programming languages (where support for these languages is available) [11, 12, 13, 14, 15, 16], reliability of these measurements and their consistency among languages and tools remains as an unsolved concern.

Early negotiation of differences in interpretation and implementation of software metrics calculation algorithms starts with [17]. Here, the authors showed that different authors implement metrics differently. This results in variation in metric results across tools even within a single language. A similar investigation towards an evaluation of metric tools supporting .NET languages [18] confirmed the variation in results for supported languages.

The problem grows with multilingual and cross-lingual software projects. Our early investigation of code analysis in such projects starts with the identification of problems in the systematic application of static metrics in practice [8] which has been concluded with a need for better tools that would consistently support metric calculation within the same language, as well as across different languages. We have tried to find a solution to the opened problem in [1]. Now we try to systematically investigate (in-)consistency of software metric results across languages and tools.

Identified inconsistencies have opened a completely new space for research and innovation, with numerous reviews, surveys and investigations in the same or a similar direction. One of the most detailed field overview studies is described in [19], while there are also available studies on tools comparison and their precision [20] still without deeper analysis and understanding of reasons for imprecision and discrepancies.

In this study we are interested in observing differences in calculated metric values. Starting from two syntactically similar languages (Java and C) that are, at the same time widely supported by available metric tools, and from small datasets to enable also manual inspection of differences roots we aim for learning about the reasons for inconsistencies between metric values between tools and input languages.

3. Study Description

This section describes the research design by introducing the datasets and methods used. It defines criteria for static tools and code sample selection, as well as a description of the well-known software complexity metrics and experimental setup.

3.1. Multilingual Code Samples

The research includes equivalent implementations of the selected standard problems and toy examples implemented in Java and C programming languages [21]. This enables a mutual comparison of the code implementation languages themselves. The selected codebases provide multilingual implementations of defined problems, which enables, among other things, a mutual comparison of the code implementation languages themselves.

An overview of alternative datasets of multilingual implementations can be found on the RossetaCode webpage [22]. Many available online codebases don't provide enough examples or consistency in solving the same problem in multiple programming languages. Considering the previously mentioned, the RossetaCode database was singled out as the only adequate bigger source for this research. Multilingual examples in RossetaCode semantically solve the same problems, but their syntax doesn't have to be identical. The mentioned code bases are a good starting point for this study.

- RosettaCode is a collection of solutions for standard problems usually implemented by contributors who are developers in the language in which they contribute to the collection. An overview of alternative datasets of multilingual implementations can be found on the RossetaCode webpage [22]. Many available online code bases do not provide enough examples or consistency in solving the same problem in multiple programming languages. Considering the previously mentioned, the RossetaCode database was singled out as the only adequate bigger source for this research. Examples in RossetaCode solve the same problems in a semantically equivalent way, but their syntax is different.
- SumProd is a collection of cloning scenarios, for all four types of clones, going from renaming a variable, across a reordering statement, to replacing *while* loop with a *do* one or with an *if* branching. It is used with small modifications, on the original code version, to achieve maximum uniformity of codes between languages. It should provide a small set of identical codes to be used as a control point to conclusions of bigger and divergent codes of the RossetaCode sample.

It should also be noted that the datasets don't provide specific information about the program code. This means that we cannot gain insight into whether the code was written by the same developer, a different one, or the level of their knowledge of the implementation language. The initial step of the research included selecting adequate examples from the mentioned datasets, which implies their availability in both included programming languages and their implementation in one file (to avoid complicated compensation of program code modularity and implementation versions).

3.2. Choosing Software Metrics

The covered metrics in this research are fundamental metrics of software code design complexity and quality such as McCabe, Halstead, and Maintainability Index. They include other basic metrics such as lines of code, and the number of operators and operands. The primary criteria for selecting metrics are their availability in considered tools (Section 3.3).

A short explanation of caught metrics [1]:

- **LOC: Lines of code** that measures the size of source code is available in different variants. E.g.
  Physical lines of code: count every single line of code
  Logical lines of code: consider different aspects of the code which overcome coding style dependency
- **McCabe: Cyclomatic Complexity (CC)** of control-flow structures as number of linearly independent paths (V) through a control-flow graph (CFG) representation of a method/function[1] source code.

$$V(CFG) = (No.ofEdges)\breve{}(No.ofNodes) + 2$$

- **Halstead**: Product size and complexity metrics set based on the number of operators and operands.

$$n1 : No.ofUniqueOperators$$

$$n2 : No.ofUniqueOperands$$

[1]The limitation to method/function represented by a CFG is based on the limitation the the graph should be strongly connected and have one entry and exit point (see [1] for the formal definitions.

$$N1 : No.ofTotalOperators$$

$$N2 : No.ofTotalOperands$$

$$Length : N = N1 + N2$$

$$Vocabulary : n = n1 + n2$$

$$Volume : V = N * log_2 n$$

$$Level : L = \frac{n1}{n2}$$

$$Difficulty : D = \frac{n1}{2} * \frac{N2}{n2}$$

$$Effort : E = D * V$$

- **Maintainability Index**: Relative ease of maintaining the code.
  Original formula (0-100+):

  $$MI = 171 - 5.2 * \ln HalsteadVolume - 0.23 * CyclomaticComplexity - 16.2 * \ln LinesofCode$$

  Microsoft version (0-100):

  $$MI_{microsoft} = MAX(0, MI * \frac{100}{171})$$

3.3. Static Analyzer Tools selection

A further step requires the selection of tools known as static analyzers which will automatically analyze the code examples. We initiated a search for existing tools used and recommended by real-life developers. Among numerous lists, we identified the most complete source[23].

Only open-source tools and commercial tools that provide a "free" or "trial edition" license are included in this survey. Popular commercial tools like SonarQube [24], Embold [25], PVS-Studio [26], DeepSource [27], and some others were excluded from the research because they did not provide adequate language or metrics coverage in the mentioned releases.

The review of the static analyzers market concludes that tools mainly focus on spotting and reporting code errors, security flaws, and code design irregularities rather than providing quantifiable metric values. For example, the SonarQube tool in the "free plan" does not support the C programming language, while Embold and DeepSource in the mentioned edition provide only descriptive metrics of their arrangement. Among the commercial tools, the research has included FrontEndART SourceMeter in the form of the SonarQube plug-in [28], CoderGears CppDepend/Jarchitect [29], and M Squared Technologies's Resource Standard Metrics [30].

Some of the commercial tools are available in the form of SonarQube plugins like SourceMeter, CppDepend, and JArchitect. We used SourceMeter in the mentioned plugin form because it provides easier access to output results.

Extensive research into the open-source static analysis tools, via the GitHub portal, has been done in multiple revisions to achieve consistent and comparable metric values as mentioned between the languages and the tools. Certain tools could not successfully report metrics on the source code and therefore were not included in this research. On the other hand, some tools required minor corrections to display the appropriate parameters of the results or considerable upgrades of the logic due to its incompleteness.

The previously mentioned is shown in Table 1. The original names of the tools have been changed in the research (clearly indicated by their links) for easier distinction due to existing overlaps. Adoption of open-source tools has been involved following changes:

- C-Cyclomatic-Halstead - working with a whole directory of source files instead of file-by-file
- C-Static-Analyzer - display total no. of operators/operands for provided source file

- Java-4-Metrics - debug of runtime-errors during the execution of RossetaCode examples The survey did not include popular languages like JavaScript and Python, due to the small number of static analyzer tools dedicated to them that could provide consistent and comparable results.

Table 1
Open-source tools available on the GitHub portal, and commercial tools with corresponding website, all supporting the requested metrics and programming languages

Open-Source Tool	**GitHub Links**
	Java
Java_complexity _and_organization	https://github.com/Taha248/Java-Code-Analyzer/tree/b30e240
Java_halstead _cyclomatic	https://github.com/MohamedSaidSallam/halstead_cyclomatic
Java_4_metrics (logic adopted [31])	https://github.com/AccuType-911/Java-code-analyzer/tree/fc82210
	C
C-Static-Analyzer (output adopted [32])	https://github.com/sajedjalil/C-Static-Analyzer/tree/cbc5e5b
C_cyclomatic _halstead (output adopted [33])	https://github.com/AhmedEssam17/ CyclomaticComplexity-and-HalsteadMetrics/tree/1c2b9de
Commercial Tool	**Website**
CppDepend (2023.1)	https://www.cppdepend.com
JArchitect (2023.1)	https://www.jarchitect.com
SourceMeter (10.0.0)	https://sourcemeter.com
ResourceStandard Metrics (7.70)	https://msquaredtechnologies.com

3.4. Experimental setup

The research provides a comparison of static analysis tools over multilingual datasets in terms of software metric values. To make this experiment more replicable, we provide all used code samples online available in the repository at [21]. Some tools (like SourceMeter, CppDepend, and JArchitect) have been required to create whole project structures of given source code. The necessary steps for running the specific tool on a particular dataset have also been issued in detail in the GitHub repository. As earlier mentioned, our research includes a manual check (based on previously listed metric formulas) of each tool's output results. Manual checking of the results proved especially necessary with open-source tools, but also missing metrics on some tools are manually calculated to make their outcomes mutually more comparable.

4. Results

This section presents the metric results of static analysis on the selected codebase. Metric results are available on our GitHub repository [21]. It should be noted again that the given RossetaCode examples do not represent identical codes for different languages but rather they represent similar implementations of the same problems in different languages. For the analysis of identical multilingual samples, we used the SumProd dataset. The whole experiment is based on Java and C programming languages.

4.1. Cross-language comparison on RossetaCode samples

The results of the Resource Standard Metrics tool, shown in Table 2 and Table 3, indicate that the examples written in the C programming language contain more extensive implementation in terms of the

lines of code and program control flow, as expected based on nature of the language [34]. Considering previous facts, they express greater code complexity as measured by Cyclomatic Complexity (Cyclo Vg). They also encourage the appearance of more "defects in the code which are not covered by the compiler" (Notice).

Table 2
Summary of basic metrics in Resource Standard Metrics tool
Colors: red - less favorable value, blue - uniform values

Metrics	RossetaCode		SumProd	
	C	Java	C	Java
Quality				
Funcs	55	58.5	36	36
Notice	825	642.5	388	558
QN/Func	15	11.4	10.8	15.5
QN/KeLoc	1346.5	1146.5	2380	3382
Complexity				
Param	75	76	54	54
Return	62.5	70.5	36	36
CycloVg	192.5	171.5	56	56
Total	330	318	146	146
Lines Of Code				
LOC	785	735	219	221
eLoc	630.5	586	163	165
lLoc	450.5	408.5	124	108
Comment	46.5	20.5	20	19
Lines	965.5	861	220	257
Function Points				
FP(LOC)	6.1	13.85	1.7	4.2
FP(eLOC)	4.95	11.1	1.3	3.1
FP(lLOC)	3.5	7.7	1	2

Despiting a larger number of methods, parameters, and return values, the examples written in the Java language reflect lower complexity but higher values of Functional Points (derived from the number of lines of code).

The same static tool in Table 3 reveals that C language dominates in several used program constructs (keywords). Also, it demonstrates inequality between two language implementations of RossetaCode samples.

The values in Table 4 of the SourceMeter tool demonstrate a similar trend in terms of number of Lines of Code and Cyclomatic Complexity. The remaining two Halstead Length & Vocabulary parameters (based on the total and unique number of operators and operands) are, as expected, higher for Java language examples.

Contrary to the above observations, the CppDepend and JArchitect cross-tool report on RossetaCode (Table 5) indicates that C examples express higher values of Halstead metrics. Other metric values like LOC, McCabe, and Maintainability Index are compatible with the previously mentioned tools.

Averages of multiple open-source tools in the same languages are concordant with previous CppDepend and JArchitect conclusions except for Halstead Vocabulary Size and Maintainability Index. It must be noticed that outcome values are dependable on the selection of the covered open-source tools.

Despite well-known metric formulas, static code analyzers provide uneven results. Because of that, this experiment doesn't provide judgment about the amount of cross-language differences. This is the reason that induces us to perform further cross-tool analysis.

Table 3
Construct differences in C and Java of RossetaCode examples (Resource Standard Metrics tool)
Colors: red - less favorable value, blue - uniform values

Code Construct	Programming Language			
	C		Java	
Lines				
Blank Lines	160		118	
Comment Lines	46.5		20.5	
Total Log | Phy Lines	991.5	965.5	873.5	861
Key Words				
Literal Strings	112.5		120.5	
Preprocessor Lines	45		0	
#include / import	30		22.5	
const, enum	8	0.5	0	0.5
do, while	1	11	0.5	9
switch, default	1	0.5	0.5	0
for	47		40.5	
case, break	3	4	2	2
if, else	45	9	47	10.5
Funcs	55		59	
goto, return	0.5	50	0	41
exit, _exit, abort	0.5	0	0	0
macros, struct	17.5	0	9	0
class, interface	5.5	0	17	0
Analysis				
Paren Count (,)	475.5		518	
Brace Count {,}	137		175	
Bracket Count [,]	127.5		134.5	
Char/NBLine, Notices	17	825	21	642.5
Code, eCode Lines	78.90%	63.40%	84.10%	66.90%
Comment, Blank Lines	4.80%	16.25%	2.35%	13.60%
Characters, Spaces	72.65%	27.35%	65.55%	34.45%

Table 4
Results of Available Parameters in SourceMeter Tool.
* - manually calculated; ** - Microsoft version (manually calculated)
Colors: red - less favorable value, blue - uniform values

Metrics	RossetaCode		SumProd	
	C	Java	C	Java
Halstead Length	68.34	78.33	25.67	25.78
Halstead Vocabulary	25.54	30.10	12.72	13.75
Halstead Volume*	325.00	382.26	94.22	97.52
McCabe's CC	3.25	2.75	1.56	1.56
Maintainability Index**	47.42	49.67	69.65	69.59
Number of Invocation	2.74	2.23	0.50	0.50
Logical LOC	44.29	40.90	5.58	5.58
Physical LOC	52.61	37.13	5.61	5.58
Total Logical LOC	44.29	40.90	5.58	5.58
Total Physical LOC	52.61	37.13	5.61	5.38

Answer to RQ1: Based on agreement between covered static tools outcomes, we can evidence that C language RossetaCode samples are larger in terms of LOC and more complex according to McCabe values. The majority of tools also agree that Halstead metrics and Maintainability Index are worse for C examples as covered in Figure 1.

Table 5
Results of CppDepend and JArchitect tools on RossetaCode examples Written in C and Java
* - Logical lines of code; ** - Microsoft version; [] - manually recalculated
Colors: red - less favorable value

Metrics	C (119 Methods)		Java(116 Methods)	
	Avg	StDev.P	Avg	StDev.P
Construct Details				
Unique Operands	12.02	7.15	19.39	19.82
Total Operands	32.13	30.54	8.37	5.10
Unique Operators	9.60	4.84	12.47	10.72
Total Operators	32.22	33.83	5.61	3.86
LOC*	8.49	8.97	7.38	6.86
Paramters	1.29	1.09	1.28	0.94
Variables	1.81	1.97	2.09	2.14
Methods Called	1.72	2.00	2.90	3.08
Methods Calling Me	0.90	0.78	0.72	0.61
Basic Complexity				
McCabe Complexity	3.12	2.62	2.92	2.21
Maintainability Index**	68.76	14.06	72.58	11.61
Halstead Complexity				
Program Length	64.34	63.90	13.98	7.49
Vocabulary Size	21.61	10.86	31.86	29.35
Program Volume	309.11	360.97	71.12	52.37
Difficulty Level	[13.04]	9.29	[3.03]	1.53
Program Level	[0.87]	0.17	[0.74]	0.20

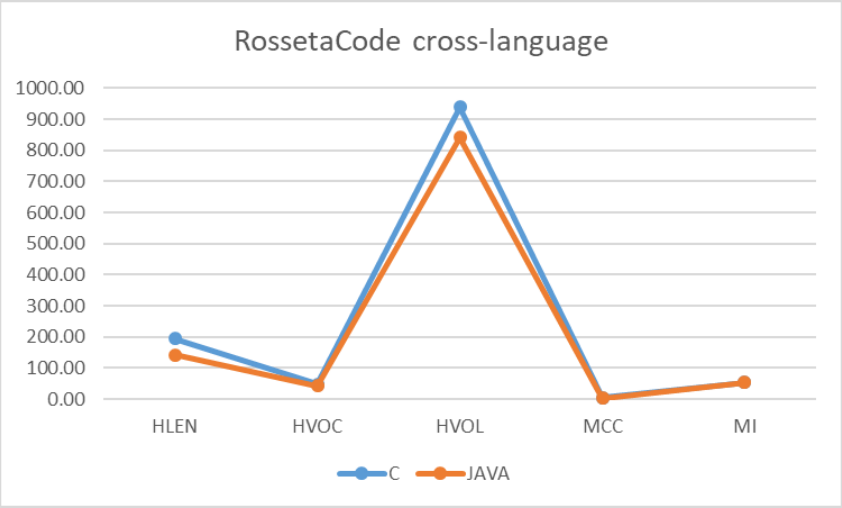

Figure 1: Cross-Language comparison of RossetaCode samples.
HLEN: Halstead Length, HVOC: Halstead Vocabulary, HVOL: Halstead Volume, MCC: McCabe Cyclomatic Complexity, MI: Maintainability Index

4.2. Cross-tool comparison on RossetaCode samples

The standard deviation among the results of the open-source tools (Table 6) directly demonstrates the mentioned deviations in the "cross-tool" values of the same parameter. Comparing outcomes in Tables 2, 4, 5, 6, and 9, it becomes obvious that all tools provide disparate values. Among all parameters, the McCabe seems to be the most uniform metric, but also it doesn't manifest identical values across all considered static tools.

Observing the above defect and considering earlier mentioned changes in open-source tools, it suggests that the way of interpretation and parsing program constructs should have the biggest impact

Table 6
Averages of open-source tools.
* - Java_complexity_and_organization (total not available);
** - Microsoft version (manually calculated);
*** - Java projects are manually calculated;
[] - standard deviation between tools results on a population level.
Colors: red – less favorable value.

Metrics	RossetaCode		SumProd	
	C	Java	C	Java
Construct Details				
Unique Operands	57.08[4.66]	72.57[53.91]	6.78[4.77]	11.00[3.32]
Total Operands *	170.99[7.59]	117.74[21.65]	15.58[5.11]	21.94[2.22]
Unique Operators	40.03[15.77]	84.61[84.30]	10.11[13.58]	15.15[2.19]
Total Operators *	196.36[39.10]	207.55[46.36]	23.47[17.47]	48.39[8.28]
McCabe's Complexity				
Cyclomatic Complexity	6.10[1.80]	4.46 [2.06]	1.56[0.56]	2.74[0.45]
Maintainability Index **	43.39[4.99]	39.65 [0.91]	63.32[4.17]	57.94[0.66]
Halstead Complexity				
Program Length	449.27[43.95]	334.17[56.92]	41.86[29.25]	63.52[9.68]
Vocabulary Size	97.11[20.43]	64.61[12.87]	16.89[9.89]	25.94[4.83]
Program Volume ***	2185.61[481.80]	2069.89[405.64]	250.28[227.82]	355.55[119.57]
Difficulty Level ***	55.05[16.92]	39.24[6.24]	10.90[8.40]	16.74[2.61]

on the final outcomes of tools. There is no consensus among tools about the uniform way of treatment program code constructs. Comparison in Table 7 showed that open-source tools are proven as very divergent in the possibility of distinguishing between program constructs. Further, previous observations and comparisons with commercial tools in Tables 5 and 9 bring the same conclusion.

Table 7
Dinstinction of Detected Program Constructs over Open-Source Tools

Tools	RossetaCode		SumProd	
	uniqopnd	uniqoptr	uniqopnd	uniqoptr
Java				
Java_complexity_and_organization	148.29	203.65	15.33	16.17
Java_halstead_cyclomatic	42.39	30.81	10.39	17.17
Java_4_metrics	27.03	19.39	7.28	12.11
C				
C-Static-Analyzer	61.73	55.80	2.00	5.00
C_cyclomatic_halstead	52.42	24.26	11.56	15.22

Interestingly, manual checking of the results of the open-source tools (Table 8) and commercial tools (Table 5) also demonstrates "in-tool" inconsistencies between Halstead values provided by the tool and those manually recalculated with well-known metric formulas. Considered recalculations are based on the information available in the results of the tool about the total and unique number of operators and operands (Table 7 and 5).

Parallel examination of Table 4, 5, and 6, release that open-source tools output higher values than commercial tools. Again, open-source tools utilize less restrictive approaches to program constructs assessment which is clearly shown in Table 5, 7, and 9.

Table 8
Inconsistency in Final Results of Open-Source Tools
[] - manually recalculated

Tools	RossetaCode		SumProd	
	hvol	hdiff	hvol	hdiff
Java				
Java_complexity _and_organization	same		same	
Java_halstead_cyclomatic	2448[2535]	47.84[48.39]	506[339]	same
Java_4_metrics	not provided		not provided	
C				
C-Static-Analyzer	1704[2408]	71.97[71.05]	same	same
C_cyclomatic_halstead	2667[2629]	same	478[333]	same

Table 9
Results of CppDepend and JArchitect tools on SumProd examples Written in C and Java
* - Logical lines of code
** - Microsoft version
[] - manually recalculated
Colors: red - less favorable value, blue - uniform values

Metrics	C (34 Methods)		Java(34 Methods)	
	Avg	StDev.P	Avg	StDev.P
Construct Details				
Unique Operands	9.00	3.07	9.97	7.02
Total Operands	16.56	9.64	5.59	2.65
Unique Operators	5.59	4.65	4.59	4.68
Total Operators	9.62	8.77	3.62	3.69
LOC *	4.44	4.47	3.56	3.57
Paramters	1.50	0.56	1.50	0.56
Variables	1.50	1.50	1.50	1.50
Methods Called	0.50	0.50	0.50	0.50
Methods Calling Me	0.50	0.50	0.50	0.50
Basic Complexity				
McCabe Complexity	1.56	0.60	1.56	0.6
Maintainability Index **	81.50	18.52	83.97	16.04
Halstead Complexity				
Program Length	26.18	18.38	9.21	6.33
Vocabulary Size	14.59	7.68	14.56	11.68
Program Volume	110.47	89.07	38.69	34.61
Difficulty Level	[5.85]	4.97	/	/
Program Level	[0.51]	0.06	/	/

Answer to RQ2: "Cross-tool" analysis exhibits disparity in the majority of output values. Moreover, there is no one metric value that is consistent between all static tools. Also, "in-tool" analysis reveals that some tools don't correctly follow well-known metric formulas. This distinction characterizes both open-source and commercial tools as demonstrated in Figure 2.

4.3. Consistency comparison on SumProd identical code samples

This section focuses on strictly comparing the results of identical program codes, to the mentioned shortcomings of the RossetaCode example. For this purpose, the SumProd examples were used.

The file changes themselves and the intermediate results of the static tool showed the expected absence of the impact of gaps and comments in the code on outcome metrics value. The confirmation of

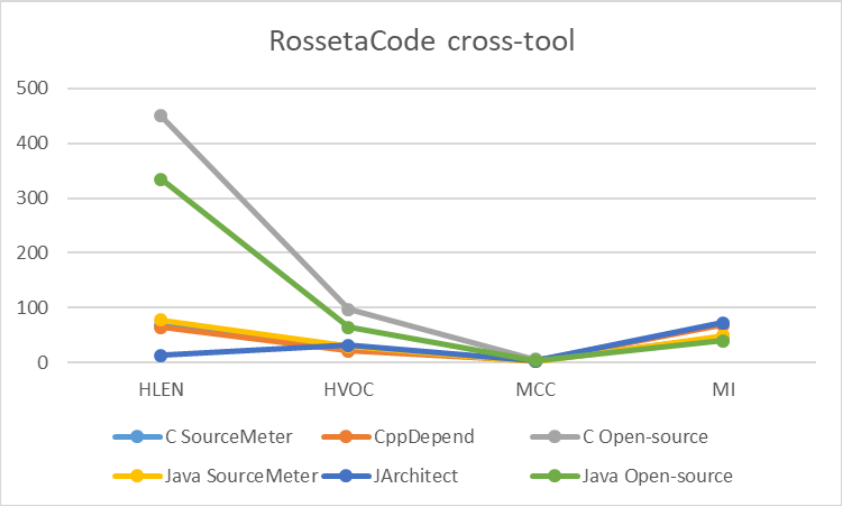

Figure 2: Cross-Tool comparison of RossetaCode samples.

the identity of the constructs of the program codes discussed in this section is clearly shown in Tables 2, 9, and 10.

Table 10
Construct difference in C and Java codes of SumProd examples (Resource Standard Metrics tool)
Colors: red - less favorable value, blue - uniform values

Code Construct	Programming Language			
	C		Java	
Lines				
Blank Lines	0		36	
Comment Lines	20		19	
Total Log | Phy Lines	239	220	276	257
Key Words				
do, while	1		1	
for	16		16	
if, else	3		3	
Funcs	36		36	
class, interface	0		18	
Analysis				
Paren Count (,)	76		76	
Brace Count {,}	55		73	
Bracket Count [,]	0		0	
Char/NBLine, Notices	9	388	10	558
Code, eCode Lines	91.60%	68.20%	80.10%	59.80%
Comment, Blank Lines	8.40%	0.00%	6.90%	13.00%
Characters, Spaces	68.80%	31.20%	63.50%	36.50%

On the other hand in Table 2, the Resource Standard Metrics tool indicates higher function point values, line of code metrics, and notices (errors not covered by the compiler) in Java examples. Considering that the tool doesn't include the Halstead metrics of code complexity, identical program codes can't show differences in terms of the cyclomatic complexity itself. Higher values of function points for Java examples are only concordant observations between SumProd and RossetaCode datasets.

The results of the SourceMeter tool on the SumProd dataset in Table 4 indicate that the implementation of Java examples is more complex in terms of Halstead and Maintainability Index. Examples of both languages are identical concerning McCabe and Physical LOC metrics, but C language is slightly complicated regarding only Logical LOC. Again, we can see that Halstead complexity measures (opera-

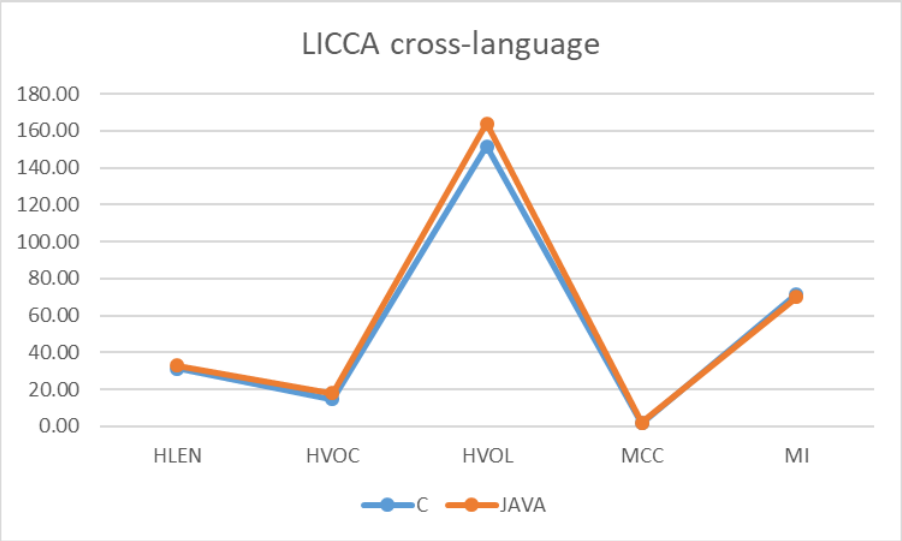

Figure 3: Cross-Language comparison of SumProd samples (taken from LICCA cloning scenarios).

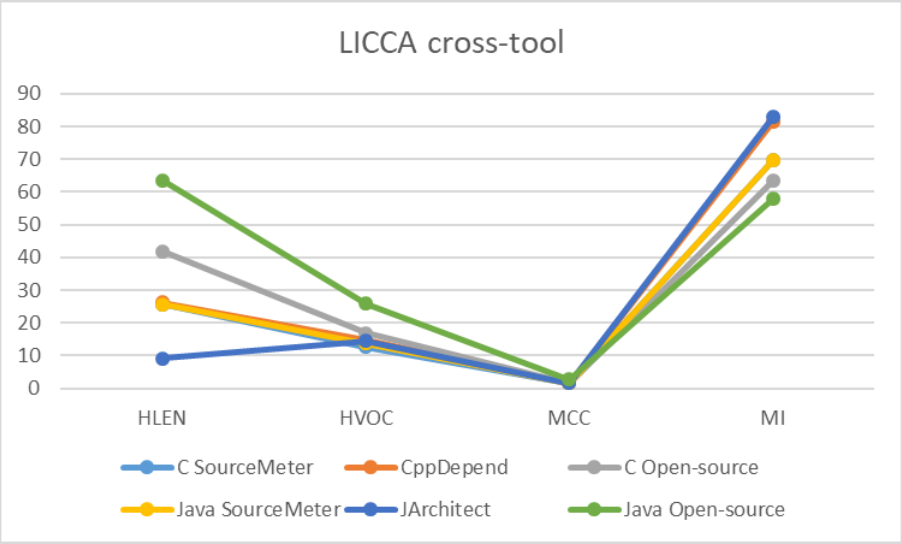

Figure 4: Cross-Tool comparison of SumProd samples (taken from LICCA cloning scenarios).

tors/operands quantity) prove more sensitive against McCabe metric (control-flow paths). Comparing with RossetaCode results of the SourceMeter tool, we can generally conclude that Java language is more complex regarding Halstead values while C language tends to be robust in terms of Logical LOC. In the context of average open-source tool results (Table 6), complexity metrics except the Maintainability Index are higher for Java examples. The outcomes of SumProd and RossetaCode code samples are exactly the opposite for open-source tools. Although the code samples are almost identical, outcomes in Table 9 suggest that the CppDepend tool has higher values of operators and operands followed by LOC, Halstead, and Maintainability Index. Higher values of LOC, Maintainability Index, and Halstead metrics match between SumProd and RossetaCode datasets. Its results on the SumProd case are opposite to SourceMeter and open-source outcomes.

Answer to RQ3: Contrasting to RQ1, the majority of static tools on the SumProd code suggest that Java examples manifest more complexity compared to C code samples as shown in Figure 3. Cross-tool observations on the SumProd case in Figure 4 reveal the same conclusions as RQ2.

5. Limitations

Our research exposes some restrictions of its scope. The analysis doesn't provide cross-platform testing between tools on different operating systems. Also, our experiment only covers two programming languages because of the availability of static analysis tools which are included only in its "free-form". As discussed earlier, the size and diverseability of used code datasets and software metrics are also bounded. Because of subjectivity and the unavailability of used metric definitions by tool designers, research doesn't deeply consider approaches to their calculating.

As another limitation one may consider selection of small datasets based on classical algorithms and toy examples of cloning scenarios. However, this selection enables us to manually investigate calculated and expected values and learn about the reasons for imprecision and discrepancies.

6. Conclusion and Future Work

The paper investigates the inconsistency between different static analysis tools for calculation of traditional and widely used LOC, Halsetad and McCabe CC metric, on similar and identical code samples implemented in Java and C programming languages.

Opinions are divided over the cross-language static analysis. The results on heterogeneous multilingual RossetaCode samples reveal that C codes are more complex implementations as expected. On the other hand, metric observations on homogeneous multilingual SumProd samples suggest that implementations of Java codes are more demanding. The reason for this inconsistency is hidden in the fact that SumProd implementation does not involve object-oriented design and avoiding complexity on code level based on design complexity. Based on these conclusions we can discuss a need for a complexity metric to consider complexity of an implementation taking into account both: source code and the implementation design inbuilt in it (as statically as possible having in mind dynamic nature of the design).

The cross-tool review releases absolute inconsistency of metric values over all covered open-source and commercial static tools. Moreover, a manual check of in-tool outcomes reveals that some static analyzers do not coherently follow well-known metric formulas. The primary reason for the result distinction may be the well-known subjectivity in metric definition and program constructs interpretation involved by the tool author. This is an obvious open space for future work - filling the gap between metric algorithm implementations to consistently support different input languages as proposed by [1].

References

[1] G. Rakić, Extendable and adaptable framework for input language independent static analysis, Ph.D. thesis, University of Novi Sad (Serbia), 2015.

[2] RosettaCode, Accessed on 1.7.2024. URL: https://rosettacode.org/wiki/Rosetta_Code.

[3] T. Vislavski, G. Rakić, N. Cardozo, Z. Budimac, Licca: A tool for cross-language clone detection, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 512–516. doi:10.1109/SANER.2018.8330250.

[4] SumProd, Cloning scenarios provided by LICCA tool, Accessed on 1.7.2024. URL: https://github.com/gocko/licca/tree/master/test/sumProd.

[5] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.

[6] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

[7] T. Heričko, B. Šumak, Exploring maintainability index variants for software maintainability measurement in object-oriented systems, Applied Sciences 13 (2023). URL: https://www.mdpi.com/2076-3417/13/5/2972. doi:10.3390/app13052972.

[8] G. Rakic, Z. Budimac, Problems in systematic application of software metrics and possible solution, arXiv preprint arXiv:1311.3852 (2013).

[9] L. Ardito, R. Coppola, L. Barbato, D. Verga, A tool-based perspective on software code maintainability metrics: A systematic literature review, Scientific Programming 2020 (2020) 8840389.
[10] T. Beranič, M. Heričko, Comparison of systematically derived software metrics thresholds for object-oriented programming languages, Computer Science and Information Systems 17 (2020) 181–203.
[11] N. Govil, Applying halstead software science on different programming languages for analyzing software complexity, in: 2020 4th international conference on trends in electronics and informatics (ICOEI)(48184), IEEE, 2020, pp. 939–943.
[12] N. Govil, Analyzing software complexities by applying data structure metrics on different programming languages, in: 2020 5th International Conference on Communication and Electronics Systems (ICCES), IEEE, 2020, pp. 833–838.
[13] S. A. Abdulkareem, A. J. Abboud, Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics), in: IOP Conference Series: Materials Science and Engineering, volume 1076, IOP Publishing, 2021, p. 012046.
[14] M. Shoaib, M. S. Naveed, A. A. Sanjrani, A. Ahmed, et al., A comparative study of contemporary programming languages in implementation of classical algorithms, Journal of Information & Communication Technology (JICT) 14 (2021).
[15] M. Balogun, Comparative analysis of complexity of c++ and python programming languages, Asian J. Soc. Sci. Manag. Technol 4 (2022) 1–12.
[16] K. A. Onyango, J. Kamiri, G. M. Muketha, A comparative study of the lexicographical complexity of java, python and c languages based on program characteristics, Journal of Innovation, Technology and Sustainability 1 (2023) 42–67.
[17] R. Lincke, J. Lundberg, W. Löwe, Comparing software metrics tools, in: Proceedings of the 2008 international symposium on Software testing and analysis, 2008, pp. 131–142.
[18] J. Novak, G. Rakić, Comparison of software metrics tools for .net, in: Proc. of 13th International Multiconference Information Society-IS, volume A, 2010, pp. 231–234.
[19] Z. Mushtaq, G. Rasool, B. Shehzad, Multilingual source code analysis: A systematic literature review, IEEE Access 5 (2017) 11307–11336.
[20] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, F. Palomba, A critical comparison on six static analysis tools: Detection, agreement, and precision, Journal of Systems and Software 198 (2023) 111575. URL: https://www.sciencedirect.com/science/article/pii/S0164121222002515. doi:https://doi.org/10.1016/j.jss.2022.111575.
[21] Repositorium of the study samples and results, Accessed on 1.7.2024. URL: https://github.com/zaricu22/Multi-Lang_Metrics/.
[22] Alternative collections, Accessed on 1.7.2024. URL: https://rosettacode.org/wiki/Help:Similar_Sites.
[23] Top 40 static code analysis tools, Accessed on 1.7.2024. URL: https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools.
[24] SonarQube, Accessed on 1.7.2024. URL: https://www.sonarsource.com/products/sonarqube/.
[25] Embold, Accessed on 1.7.2024. URL: https://embold.io/.
[26] PVS-Studio, Accessed on 1.7.2024. URL: https://pvs-studio.com/en/pvs-studio/.
[27] DeepSource, Accessed on 1.7.2024. URL: https://deepsource.com/.
[28] FrontEndArt SourceMetter, Accessed on 1.7.2024. URL: https://github.com/FrontEndART/SonarQube-plug-in.
[29] JArchitect, Accessed on 1.7.2024. URL: https://www.jarchitect.com/.
[30] Resource Standard Metrics, Accessed on 1.7.2024. URL: https://msquaredtechnologies.com/Resource-Standard-Metrics.html.
[31] AcuType, Java Code Analyzer, Accessed on 1.7.2024. URL: https://github.com/AccuType-911/Java-code-analyzer/pull/1.
[32] C Static Analyzer, Accessed on 1.7.2024. URL: https://github.com/sajedjalil/C-Static-Analyzer/pull/1.
[33] CC and Halsead Metrics, Accessed on 1.7.2024. URL: https://github.com/AhmedEssam17/CyclomaticComplexity-and-HalsteadMetrics/pull/1.

[34] N. Patakia, Z. Porkolába, E. Csizmásb, Why code complexity metrics fail on the c++ standard template library, in: Proc. of the 7th International Conference on Applied Informatics (ICAI 2007), Eger, Hungary, volume 2, Citeseer, 2007, pp. 271–276.