How to Empirically Assess the Quality of Software Source Code in The Era of Multicore Architecture and Multithreaded Programming

Quentin Ferguson, Ben Barcaskey, Tyler Goldstein, and Saleh M. Alnaeli*

Mathmatics, Statics and Computer Science Department
University of Wisconsin-Stout
Menomonie, Wisconsin, USA
{fergusonq9852, barcaskeyb2473, goldstein4143}@my.uwstout.edu ,
alnaelis@uwstout.edu*

Abstract

In the general-purpose software computing domain, there are a vast amount of open source systems (OSS) that are being utilized. What can happen is that the developers of these software solutions come from varying backgrounds leading to unused proper coding styles. In this study, presented is four open source systems spanning back to their older versions in 2014. Included with these systems is a static analysis that includes statistics such as the amount of lines of code, files, jump statements, and functions with side effects. Both systems have their C/C++ systems analyzed with over two million lines of code collectively. The results show there are significant barriers that possess problematic behavior for the maintainability of source code as well as the expandability**.**

1. Introduction

Since the popularity of open source projects, we as a community have not only gained deeper insight in how some of the most popular programs run, but also how they are designed and maintained. Since open source projects existed, anyone from the community can edit a copy of the repository and submit changes to the owners. If the owners accept it, they can implement it into the code like a regular push to the repository. This caused the birth of many famous and widely used programs such as Linux, Apache, Firefox, Chromium, jQuery, Python and node. Open source projects also allow us to dissect source code and examine the quality either with prewritten tools or manual examination.

The quality of source code can be impacted by many causes, some of which include the amount of dead code, the age of source code, the amount of developers working on the project at once, the quality of the coders, the quality of design and communication on the project and many more other factors. This often consequently causes the source code to be less compatible, harder to manage, more difficult to maintain, more difficult to implement multithreading, and more difficult to update. The quality of the source code is becoming increasingly more important with the mass utilization of multicore processers and though there is no scientific measurement to gauge the quality of a certain piece of source code, we use factors such as the amount of function side effects, the amount of legacy code utilized by the program, the amount and complexity of recursion, the amount of goto and break statements, the amount of for and while loops, and the amount of classes/structures. This gives us a good idea in observing how source code quality has either improved or declined throughout time.

The purpose of our study is to examine the trends of the quality of various open source projects. Though many studies like this have been done, we decided to focus and specify our work on the certain trends that happen with certain popular repositories of open source projects throughout 4 to 5 years. This allows us to further study the trends of source code maintainability and functionality through the various chosen open source projects and analyze the resources that goes into maintaining source code.

2. Jumps Statement

Jump statements, such as Goto and Break, are typically used to alter the way in which a program runs. They are simple to implement, both logically and literally, and are also simple to recognize. Loop statements are the one of the most common types of statements where Goto and Break statements will be used to bypass lines of code. Loops typically contain lots of lines of code and operations to execute, so adding jump statements to them can be viewed as a shortcut for programmers. While these statements are convenient, they have been marked as being major inhibitors for the parallelization of source code.

System	Lines of code	Files	Number of Functions
Scilab	819,361	4,761	16,111
Otter	146,313	558	3,694
Falkon	432,166	2,945	16,793
Ripple	632,193	2,024	19,088
TOTAL	2,030,033	10,288	55,686

Table 1: The Four Open Source General Purpose Systems Used in The Study and Functions Found in the 4 Systems

Parallelization is a technique in which modern multi-core processors are able to utilize all of their cores for faster, more efficient computing.

3. Recursive Class (direct and indirect)

Recursion is a programming technique in which a function either directly or indirectly calls itself during its execution [17,18]. Recursion is an alternative to continuously performing repetitive task. Often, recursive solutions to problems in programing are ineffective regarding the amount of time and space consumption versus a non-recursive solution [17].

4. Methods and Tools

The source code for engineering systems written in C/C++ was collected, then transformed by a srcML toolkit [16]. The srcML toolkit wraps the statements and structures of the source code syntax with XML elements, allowing tools, such as Source Code Quality Analyzer (srcQ) to use an XML application program interface to find functions that carry recursive calls directly and indirectly, and jump statements (goto and break). All results have been tabulated and recorded. That is, a count of each recursive and jump statement4within the system was saved in new results file.

5. Data Collection

This section consists of the results from our handpicked repositores using our tools. This includes the goto functions, breaks, indirect recursion, direct recursion, lines of code, number of files, and number of programmer defined functions. Data from the four systems is represented in table 1. Each repository is available on either an svn repository or a

GitHub repository for better timeline management and consistency, which is a major factor in our results.

We collected the data by first downloading four years' worth of source code from a particular repository. For each year, we used a self-made program to transcode the source folder to an xml file. From there we used another self-made program to read the xml file and write the results onto a text file. This process happens individually for each annual release and every repository.

6. Results

This section contains the results from the tools previously mentioned. The prioritized results observed was the amount of functions with side effects, recurrision, and jump statements (goto and break Statements). Between all four systems, scilab has the largest ratio of funtions with side effects standing at 69%. The total amount of side effects between all four systems is a concerning 31,279 with an average of aroud 7820 per system. All systems except Ripple have seen an increase in the amount of side effects over the course of four years. Falkon has seen a drastic jump pertaining to the frequency of side effects within functions with about a 506% increase over four years. Goto statements are not being used very frequently with an average of 176 due to the large amont of goto used by scilab. Despite the large amount, there has been a decrease in their usage over their years with all of the systems.

Based off of figure 1, break statements have been in high usage between all systems. Over the years of development they have commonly increased except for ripple which has a decrease. When compared with the size of each system, scilab has the smallest number of break statements at around .24% and otter having the most at around .98%. Recurision usage has been moderatly used with otter using a significant amount with about 83% of the functions having recurision calls within them. Otherwise, all systems have around the same usage when based off of size.

System	Side Effects	Goto	Break	Function with Indirect Recursive Calls	Functions with Direct Recursive Calls
Scilab	11,108	644	6,273	145	285
Otter	1,937	6	1,434	638	132
Falkon	8,261	15	2,159	16	271
Ripple	9,973	39	1,498	0	0
TOTAL	31,279	704	11364	799	688

Table 2: Software Quality Metrics of Studied Systems

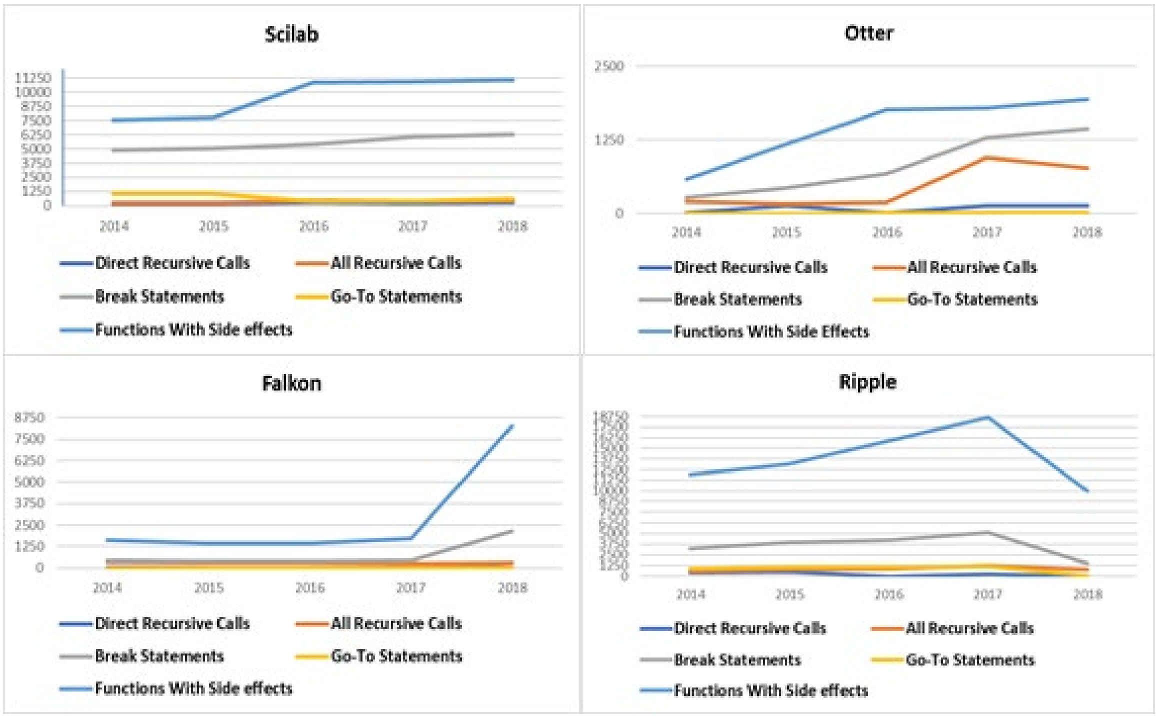

Figure 1: The Evolution of The Number of Metrics Over a Four-Year Period.

7. Conclusion and Future Work

The purpose of this study was to observe the frequency of particular metrics that played a role in making general purpose open source systems harder to maintain, develop, and support. The study was conducted by using a system developed by one of the authors that quantified each metric and then comparing the utilization of different systems. The goal of this study was to first show the frequency of concerning metrics, and additionally, examine the shift of said metrics.

Based off our analysis, break statements as well as usage of side effects, have both been heavily utilized and have been continually increasing with the expansion of the systems. With the average percentage of functions with side effects being 56% while in 2014, it was 60%. This improvement is encouraging but shows that today's practices have changed very little over the last few years. Systems are continuing to grow and with them, the negative side effects that come with particular practices currently used in software development.

The concerns brought up with this research need to be known by those developing any form of a software system. Failure to do so may result in unnecessary challenges later in development as well as loss of performance due to circumstances such as parallelization.

One potential solution would be to provide training that expressed the difficulties with using certain statements as well as guidelines that further enforce the training.

More work can be done in the future by analyzing more systems with more broad domains then showed here. Additionally, a more detailed solution to the problem could be addressed that would outline a plan.

8. Acknowledgment

The authors are grateful to the University of Wisconsin-Stout and the Office of Research and Sponsored Programs for their support.

REFERENCES

[1] J.E. Hannay, C. MacLeod, J. Singer, H.P. Langtangen, D. Pfahl and G.Wilson, "How do scientists develop and use scientific software?", Software Engineering for Computational Science and Engineering, 2009, pp 1-8.

[2] S.M. Alnaeli, J.I. Maletic, and M.L. Collard, "An empirical examination of the prevalence of inhibitors to the parallelizability of open source software systems", Empirical software engineering, 2016, vol 21, pp 1272-1301.

[3] M. Young, "The technical writer's handbook", Mill Valley, CA: University Science, 1989.

[4] S.M. Alnaeili, M. Sarnowski, C. Meier, M. Hall, "Empirically Identifying the Challenges in Parallelizing Scientific Software Systems", 25th International Conference on Software Engineering and Data Engineering, 2016, pp 1-6.

[5] S.M. Alnaeili, M. Sarnowski, "Examining the Prevalence and the Historical Trends of Indirect Function Calls in Open Source Systems: A Case Study". The Midwest Instruction and Computing Symposium, 2016, pp 1-15.

[6] S.M. Alnaeili, M. Sarnowski, S. Aman, K. Yelamarthi, A. Abdelgawad, H. Jiang, "On the Evolution of Mobile Computing Software Systems and C/C++ Vulnerable Code." Ubiquitous Computing, Electronics & Mobile Communication Conference, 2016, pp 1-7.

[7] S.M. Alnaeli, M. Sarnowski, "Historical Trends of the Multicore Adaptability and Parallelizability of Scientific Software Systems".

[8] V. Starkiovicius, R. Ciegis, A.Bugajev "On efficiency analysis of the OpenFOAM-Based Parallel Solver for Simulation of Heat Transfer in and Around the Electric Power Cables "Informatica. 2016, vol. 27, issue 1, p161-178. 18p.

[9] W.A. Bhat, S.M.K. Quadri, "Open Source Code Doesn't Always Help Case of File System Development", Trends in Information Management, 2011, vol. 7, issue 2, pp 135-144.

[10] A. Khandelwal and A. K. Mohaparta, “An insight into the security issues and their solutions for Android phones.” Computing for Sustainable Global Development (INDIACOM), 2015 2nd International Conference, 2015, pp. 106-109

[11] E.Erturk, “A case study in open source software security and privacy; Andriod adware.” Internet Security (wroldCIS), 2012 World Congress, 2012, pp. 189-191.

[12] J. Viega, J.T. Bloch, Y. Kohno and G. McGraw, “ITS4: a static vulnerability scanner for C and C++ code.” Computer Security Applications, 200. ACSAC ’00. 16th Annual Conference, 2000, pp.257-267.

[13] B. Barney. (2012) Introduction to Parallel Computing. Available: https://computing.llnl.gov/tutorials/parallel_comp/#Models

[14] D.A.P.J.L. Hennessy “Computing Architecture: A Quantitative Approach.” Morgan Kaufman Publishers, San Francisco, 2006.

[15] Y. Joonseok, R. Duskan, and B. Jongmoon, “improving vulnerability prediction accuracy with Secure Coding Standard violation measures.” 2016 International Conference on Big Data and Smart Computing (BigComp), 2016, pp. 115-122.

[16] M.L. Collard, M.J. Decker, and J.I. Maletic, “Lightweight Transformation and Fact Extraction with the srcML Toolkit.” Presented at the Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation, 2011.

[17] Michael T. Goodrich and Roberto Tamassia, Data Structures and Algorithms in Java, Book, WILEY, ISBN: 9781118771334, 2015.

[18] Nell Dale Daniel T. Joyce Chip Weems, Object-Oriented Data Structures Using Java: Edition 4, Book, Jones & Bartlett Learning, ISBN: 9781284125818, 2016.