Evolution of Code Clone Ratios throughout Development History of Open-Source C and C++ Programs

Anfernee Goon*, Yuhao Wu†, Makoto Matsushita†, and Katsuro Inoue†
*Department of Computer Science and Engineering, University of California San Diego
La Jolla, United States of America
agoon@ucsd.edu
†Graduate School of Information Science and Technology, Osaka University
Osaka, Japan
{wuyuhao, matusita, inoue}@ist.osaka-u.ac.jp

***Abstract*—A code clone is a fragment of code which is duplicated throughout the source code of a project. Code clones have been shown to make a project less maintainable because all code clones will share potential bugs and problems. Unlike other code clone research, this study analyzes the code clone ratios over the entire development lifetime of three open-source projects written in C/C++ to understand code clone growth in software over development and potential developer habits which could affect this growth. The study utilizes CCFinderX and Git to detect clone metrics across development history. The results from each project show very low, stable ratios across development history, with the code clone ratios only fluctuating greatly during the beginning of development mostly and very little refactoring occurring. This study goes further into the potential cause of low ratios and different fluctuations at different periods of development.**

***Index Terms*—code clone ratio, refactoring, code clone maintenance, open source software**

I. Introduction

A code clone is a duplicated fragment of code. Having many code clones in a project makes it much less maintainable because all of these code clones will share potential bugs and problems [1]. These problems will propagate throughout the software with continued use of the problematic code clone fragment, and a fix for this bug may have to be applied to every one of these fragments present in the code. Detecting code clones and retrieving code clone metrics for a target software can aid in bug fixing and general code clone maintenance of the detected clones.

An abundance of code clone research focuses on the code clone metrics of software at one particular snapshot. Instead of focusing on a specific point in time, we analyze the code clone metrics of software over the entire development process. In this paper, we seek to understand code clone growth and maintenance through the analysis of code clone metrics throughout the entire development process of three different open source projects primarily written in C/C++. More specifically, we will investigate the following research questions in our analysis:

- **RQ1.** *What patterns can be observed of code clone growth based on the ratios gathered from each project?*
- **RQ2.** *How do the code clone ratios throughout development characterize code clone maintenance habits?.*

By analyzing code clone growth within developing software, it is possible to better understand the developer's efforts in code clone maintenance, specifically efforts in refactoring. Although the general consensus is that code clones make software less maintainable and propagate bugs, some studies have shown that some clones can actually be beneficial [2]. By understanding the effectiveness of certain code clone maintenance techniques, it will be more clear what research needs to be done to increase the effectiveness of code clone maintenance.

II. Approach

A. Target Source Dataset

When deciding on open-source projects to use, we took into account a couple factors. These factors were how well studied the project is, how large the project is, and the type of the project itself. We wanted to use three projects which varied by these metrics in order to analyze the effects of these factors on the code clone ratios and what they imply about the development process. We analyzed the following three open-source projects which are hosted on GitHub:

1) libcurl: libcurl is a library which curl, a command-line tool for transferring data, uses. This library is fairly well studied as seen in Kawamitsu et al. [3]. It is the smallest project we studied, starting at around 2,500 lines of code, eventually growing to around 12,500 lines of code over a series of about 20,000 commits.

2) Skynet: Skynet is a lightweight online game framework which is slightly larger than libcurl, growing from around 2,500 lines of code to around 40,000 lines of code over a series of about 1,000 commits (we analyze about 800 of those commits).

TABLE I
DESCRIPTION OF METRICS USED

Metric	Description
LOC	Total lines of code, given by CCFinderX
SLOC	Total lines of code not including whitespace or comments, given by CCFinderX
CLOC	Total code clone lines, given by CCFinderX
CCR	Ratio of code clone lines to total lines of code, or CLOC divided by LOC
SCCR	Ratio of code clone lines to total lines of code not including whitespace or comments, or CLOC divided by SLOC

While not as well studied as the other two projects, it is as popular as libcurl on Github based on forks and stars.

3) Git: Git is a version control system which is widely used; even our analysis relies on Git. Although starting relatively small at 950 lines of code, it grows to around 200,000 lines of code over a series of about 40,000 commits (we analyze about 14,000 of those commits) which is significantly larger than both libcurl and Skynet.

Each of the three open-source projects present a different range in size which may provide valuable information on whether size affects code clone growth. Having software which is popular in use and analysis lessens the chance of abnormal data coming from one of the projects. The varying functions of each project ensure that the analysis covers the development of various types of software.

B. Clone Detection

To detect clones and clone metrics in the source code, we used CCFinderX, a token-based clone detector [4] [1]. Although CCFinderX is a multilinguistic clone detector, we only use it's C/C++ clone detection capabilities. Using bash scripting, we automated the use of CCFinderX on every important commit in the master branch of each project by using git log with the –first-parent flag. The commits retrieved with this flag are important because they represent the most linear development history available by following the first parent down Git's history ensuring that parallel development on separate branches are limited to the merge commit into master. Similarly, we automate the retrieval of clone metrics from CCFinderX's results to streamline collecting the results from each commit. For each commit, we collected number of C/C++ files, the metrics found in Table 1, as well as the tag of the commit if applicable. The scripts were designed to exclude test and example files whenever possible in order to keep analysis limited to functionality related files, and only includes .c and .cpp files. Header files are not included because most header files are naturally similar to each other, and may be picked up as false positives by CCFinderX. The minimum number of tokens that a fragment needs to be considered a clone is 50 in our study.

C. Qualitative Approach

Our qualitative approach to analyzing each project utilizes a tool called GemX, which provides GUI options along with CCFinderX's normal clone detection. It is an improved version of the program Gemini [5]. GemX's biggest asset is it's ability to show the clone pairs found in the source code. Using this tool, we were able to analyze a big increase or decrease in SCCR and see what caused to fluctuation. This was done by first running GemX on the commit previous to the fluctuation in SCCR and noting what clone pairs existed at that point. We then ran GemX on the commit which caused the fluctuation, and looked at whether the clone pairs that existed previously still remained, or if they were refactored out of the code base.

III. RESULTS

For our quantitative analysis, we make use of several graphs containing the metrics SCCR, SLOC, and CLOC discussed in Section 2.2 displayed by commits in chronological order. The main metric we focus on is SCCR, which we initially expected to have mostly gradual increases with periodic sharp declines. The gradual increases would be a result of functionality being added over time, which naturally increases SCCR because more code is being written [6]. Refactoring would be the cause of the sharp declines, because the initial additions of functionality may not be clean and would be in need for maintenance to ensure maintainability before the next round of functionality is added. The results from all three projects did not quite follow this trend, and in fact all projects displayed various different patterns over development.

A. libcurl

Our analysis of libcurl is confined to the files in libcurl's src folder in addition to the test and example constraints previously mentioned. The src folder is the primary folder for development for libcurl's C/C++ modules, so it contains the most relevant information about libcurl's development process.

1) Quantitative Analysis: Fig. 1 is libcurl's clone metrics graph, detailing a series of sharp increases followed by gradual decreases in SCCR which is the opposite of expectation. In hindsight, this finding makes sense because the number of consecutive commits which add functionality will be significantly less than the total number of commits resulting in sharp increases in SCCR on the graph when functionality is added. Although decreases are still occurring, the gradual nature of these decreases indicates that refactoring is not necessarily the cause. From a perspective focused on clones, refactoring would be indicated by a decrease in CLOC and SLOC, because this means that code clone fragments are being removed. Looking at the SLOC and CLOC metrics during these

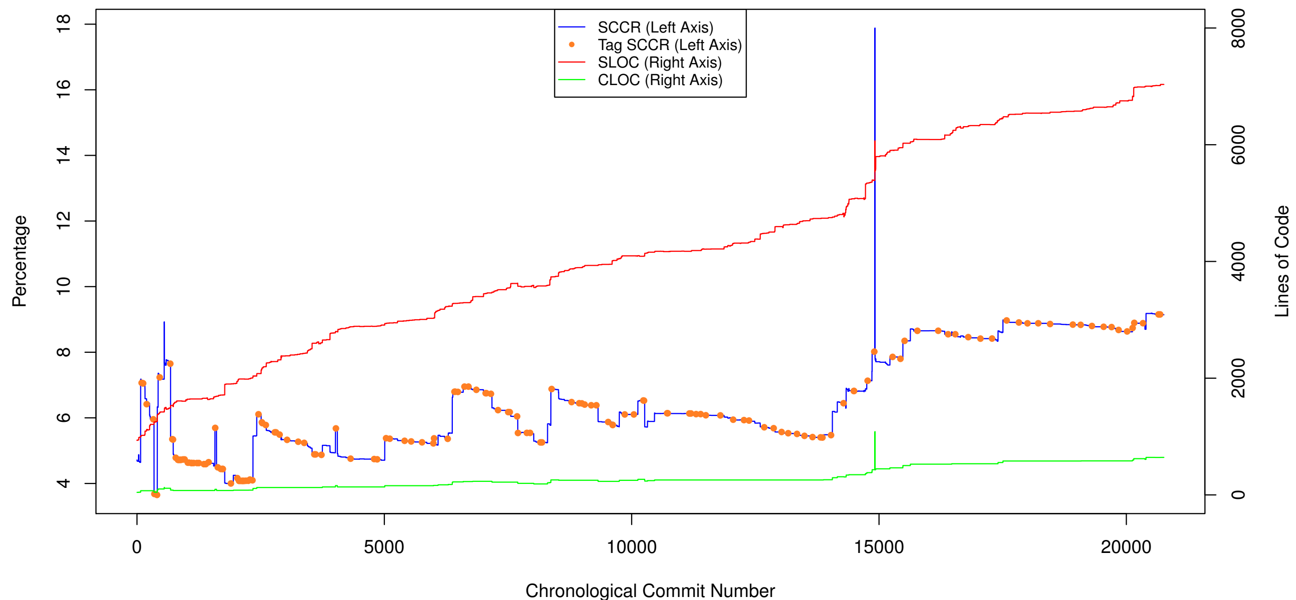

Fig. 1. SCCR, SLOC, and CLOC changes over all commits of libcurl in chronological order.

periods of decrease, we observe that the SLOC is continually increasing while the CLOC does not decrease. Since SCCR is a ratio between CLOC and SLOC, it is clear that it is the growth in SLOC which is causing the decreases and not a result of refactoring.

While this trend holds true for the majority of libcurl's development, the beginning of libcurl's development does have some sharper decreases which are a result of CLOC decreasing. These decreases are not due to refactoring, which is discussed in more detail in the next section. Contrary to the sharp increases depicted on the SCCR graph, the SCCR is actually quite stable with the sharp increases actually only being about a 1% difference. Throughout the entire lifetime of development, the SCCR mostly falls between 3% and 9%. The stable, low SCCR of libcurl throughout development may indicate a good development habit of refactoring before commits. This practice ensures a more stable clone rate with less clones being introduced each commit, leading to a cleaner git history and consistent maintainability. Unfortunately we cannot confirm this technique is in fact in use as we are only analyzing code that has been committed. Although libcurl does have relative stability throughout its development, it is still consistently growing especially towards the end of development, where a 1,000 commit sequence adds about 2% to the SCCR.

2) Qualitative Analysis: For libcurl's qualitative analysis, we studied key rises and falls on the SCCR graph attributed to CLOC changes in further depth using the qualitative approach described above. There are four commit points which are looked at in libcurl: 6562caf, 22d8aa3, b5fdbe8, a0d7a26 (shortened commit hashes). On the clone metrics graph, each of these commits represent a point before a rise or fall.

The first two commits saw a SCCR decrease of about 2% and a CLOC decrease of about 30 lines. In both these instances, the logic of the clone fragments were changed with a few additions or deletions. These fragments can still be considered code clones, but are now Type-3 code clones instead of Type-2 code clones, where Type-3 clones have some line additions and deletions which do not occur in the other fragment [1]. Since CCFinderX cannot detect Type-3 code clones, this was seen as a decrease in CLOC resulting in the SCCR decreases [1]. It is clear that the decreases were not due to refactoring in this case, but rather something more akin to a bug fix to make the fragments work correctly by changing a few parts of the fragments (and even still the fragments are Type-3 clones). This case illustrates the importance of also looking at the change in SLOC to determine if refactoring occurred, because if SLOC also decreases by a similar margin then there is a good chance the clone was removed. This phenomenon of clones becoming undetectable due to change in type is a threat to validity discussed in Section 3.5.

Unlike the first two commits, the last two commits show the most refactoring seen in libcurl. The first of the two commits, b5fdbe8, shows a huge increase in CLOC (from around 4% to around 18%) while the other commit, a0d7a26, shows a symmetrical decrease (from around 18% to around 4% again). The first commit adds a new experimental functionality through a new file which is an exact copy of another source file that is about 600 lines, increasing the CLOC by that margin. The

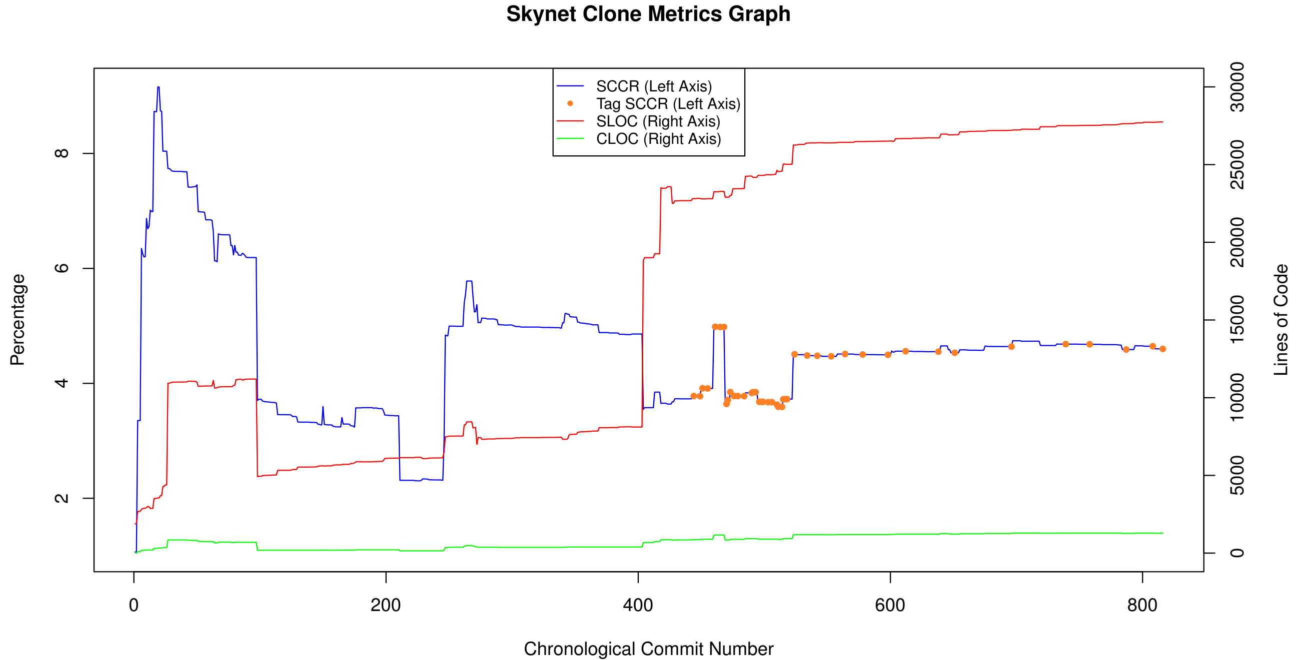

Fig. 2. SCCR, SLOC, and CLOC changes over all commits of Skynet in chronological order.

second commit significantly changes the new source file to reuse parts of the original source file along with new features, this eliminating the entire clone fragment as it was in the previous commit and reducing the CLOC down once more. This shows the only clear case of refactoring we have found libcurl, with the developer implementing reuse of source files indirectly rather than keeping directly copied pasted code.

Observation 1 - Overall, libcurl's SCCR fluctuations display opposite trends to our original expectations. The SLOC and SCCR follows a general trend of increase occurring mostly with sharp increases over a small number of commits, but there are gradual decreases among the fluctuations. Our manual analysis shows that code refactoring is not the cause of most decreases but rather from a lack of CLOC increase. Even without refactoring, SCCR is still very low indicating that refactoring may be occurring before commits or that the developers are simply able to write good code with a low amount of clones.

B. Skynet

Unlike libcurl's analysis, our analysis of Skynet takes account all C/C++ files that the project contains excluding files in the test folder. There was no single folder consolidating all of Skynet's C/C++ modules, so analyzing all the files in the project was a necessity.

1) Quantitative Analysis: Skynet's clone metrics graph, shown in Fig. 2, shows SCCR fluctuations that partially resemble what was initially expected, but with some significant differences. The resemblance is in the graph's sharp declines, which occur only a few times throughout the entire commit history. The unexpected differences are the sharp increases which mirror the sharp declines, as well as the extreme stability which characterizes the second half of the development history. Similar to libcurl, Skynet's SCCR graph's sharp increases most likely indicate an addition in functionality. These only occur during the first half of development, but are considerable increases, with the largest increase going from 6% to around 9%. The mirrored sharp decreases may indicate refactoring after large functionality additions. Most of these sharp decreases have corresponding CLOC decreases, but only a few have corresponding SLOC decreases as well indicating refactoring is not necessarily the case for every instance. The qualitative analysis of Skynet in the next section confirms refactoring efforts, showing that refactoring does occur in some of these instances.

Although the development of Skynet has fairly sharp fluctuations during the beginning of its development, it actually shows greater stability than libcurl after its initial fluctuations, occurring directly after the first release. The stability after release seems to be logical, as a polished product should be delivered at release, reducing the additions to mostly bug fixes which are less likely to introduce new clone additions.

2) Qualitative Analysis: Skynet's qualitative analysis used the same approach as libcurl's, where we looked at big fluctuations on the SCCR graph that resulted from CLOC changes in detail using GemX. The main commits that we looked at in Skynet are 28dc840 and 58aa755. Again as in

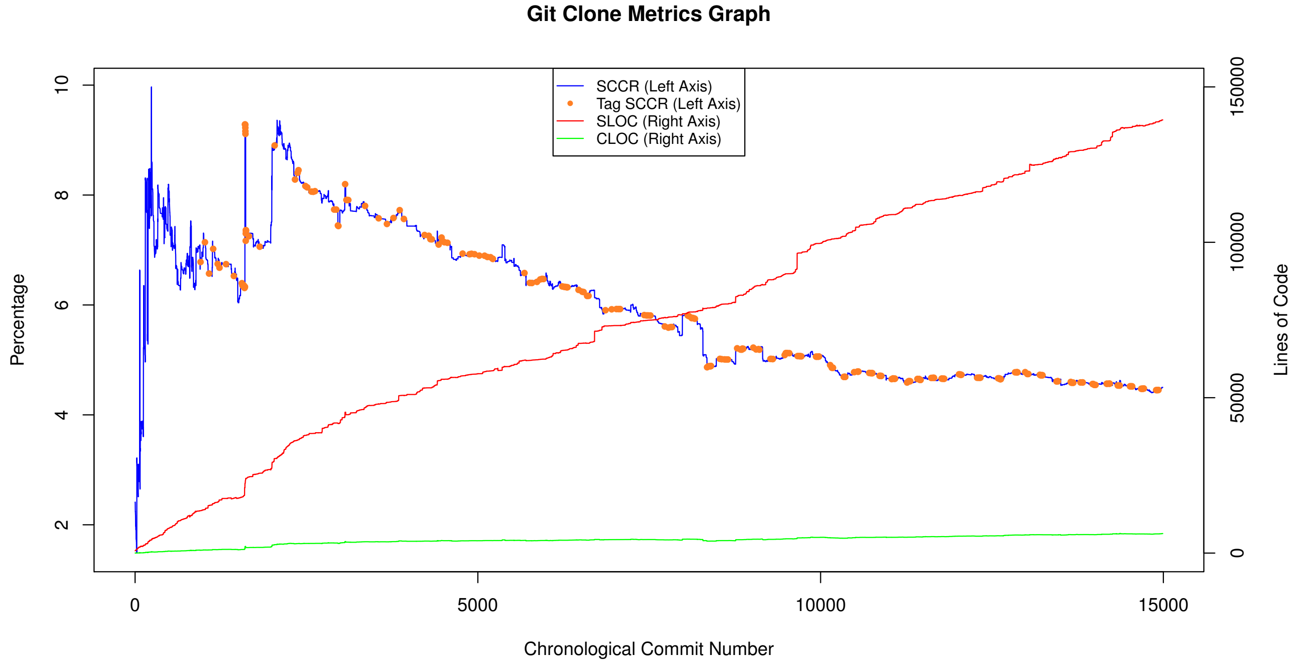

Fig. 3. SCCR, SLOC, and CLOC changes over all commits of Git in chronological order.

libcurl, these commits represent a point before a rise or fall on the clone metrics graph.

The two commits show decreases in SCCR of about 1% and 2% respectively. The cause of these decreases can both be attributed to changes in libraries which Skynet was using. The first commit sees a removal of a part of a library which contains 7,000 lines of code, 500 of those being clone lines. Although not a refactoring of their own modules, this library removal is in fact an instance of refactoring because it removes unnecessary code, potentially opting for reuse like in libcurl's instance of refactoring which in turn reduces CLOC. The second commit adds on a large library of code to the project which proportionally has very little code. Through this addition, the CLOC only increases slightly, by about 260 lines, while the SLOC increases significantly, by about 10,000 lines. Unlike the first commit, this is not a result of refactoring as none of the originally existing clone pairs were removed.

Observation 2 - The fluctuations in Skynet's SCCR during the beginning of development display similarity to the original hypothesis, but fluctuations become less significant and frequent during the later stages of development, showing that instability occurs at the start of development. Upon qualitative code analysis, it is evident that refactoring efforts on libraries are the cause of most sharp decreases in SCCR. A lack of growth in code during the second half of development indicate that most functionality was added during the beginning of development. If this is the case, it is unlikely that Skynet's developers employ refactoring before commits because the beginning of development would be more stable and cases of refactoring were shown during this time although this is not provable unless source code prior to commits is available for analysis.

C. Git

Similar to the analysis of Skynet, our analysis of Git takes into account all C/C++ files that the project contains excluding test and example files. Git's C/C++ modules were dispersed all throughout the project which is why all files in the project needed to be analyzed. Unfortunately all analysis of Git was done quantitatively, as a qualitative analysis was difficult given the size of the project. We hope to perform a qualitative analysis on Git in the future to reinforce our quantitative analysis.

1) Quantitative Analysis: The results of Git's SCCR analysis is shown in Fig. 3 and follows a similar trend as Skynet. Unlike our initial expectation, Git's SCCR sees a large growth towards the beginning of development, but after a certain point sees a gradual but consistent decrease up to the present state of development. After its large growth over around 2,000 commits, the SCCR is around 9%. The gradual decrease sees the SCCR decrease to 4% over the course of about 8,000 commits, and afterwards there is stability near 4% until the present state of development. Like in Skynet, the initial growth can be attributed to many additions of functionality at the beginning of development. After that initial stage, the CLOC barely increases while the SLOC continues to grow at a fast rate, which is what causes the gradual decline in SCCR, which indicates functionality is still being added unlike in Skynet's stable period. The gradual decrease may be due to better code

being written or code being refactored before being committed, thus having the SCCR shrink and the CLOC grow only to maintain the 4% SCCR during the stability period.

Unlike in Skynet, the first release does not correspond to the beginning of the gradual decrease, but happens during the large growth period. Despite this, the SCCR is still very low. This may simply be due to good developer habits, such as code being refactored before being committed as we have mentioned about libcurl. As stated before, we did not do a qualitative analysis on Git, but we reserve the right to look into this in the future. Despite this, the quantitative data still provides us with the ability to characterize the development of Git with the help of the analyses on libcurl and Skynet.

Observation 3 - The SCCR of Git exhibits a trend of increase during the beginning of development and a gradual trend of decrease thereafter, implying that the early development stages are more volatile than the rest of development. The extremely low average SCCR suggests refactoring before commits, but a qualitative analysis is needed to confirm this notion.

IV. Discussion

Based on the SCCR graphs of each open-source project, it is clear that every project will most likely follow it's own trends. In spite of this, there is a very broad trend which is apparent from the results for each project. All three projects have an initial period of instability and fluctuations, followed by a period of stability. This overall trend suggests a general workflow of code clone maintenance. The beginning of development has a lot of fluctuations because by nature not everything is very concrete and many large design choices are most likely being made or still in discussion. As a result of this, SCCR will change as design choices are made since code will need to be changed or refactored to accommodate for new design choices and general code clone maintenance may be initially neglected to focus on functionality implementation alone. Once design has been established, the period of stability begins where additions to the project can be refactored easily beforehand to fit with certain design principles, reducing the code clones added with each commit. While the period of stability may differ depending on what kind of development occurs after the design principles are established, these periods are still mostly stable. For example, libcurl is still developing on functionality, which results in a gradual increase during the period of stability. Meanwhile, Skynet has stopped major development of functionality resulting in an almost completely stable SCCR.

Developers can analyze their own projects in similar fashion in order to understand their own software's code clone growth and how certain maintenance techniques affect this growth. Looking at instability periods of code clone growth and analyzing what code clone maintenance techniques were employed during these times can help the developer understand the effectiveness of these techniques, and make inquiries on why these techniques may not have been effective at the time. As mentioned before, changing design choices could increase the amount of code clones added due to difficulty in refactoring or a simple neglect of code clone maintenance due to functionality concerns. If software design is not the issue, the frequency of refactoring itself could be a large factor. Looking at the clone ratios over development history can show how often refactoring occurs and whether this frequency is enough to keep the software maintained. Although ideally refactoring occurs before commits and a SCCR graph similar to the stable part of Git's graph is produced, time constraints may not allow for such intense refactoring practices, which is where understanding how frequently refactoring should occur can be vital to a project's code clone maintenance.

V. Threats to Validity

Since our analysis relies heavily on the output of CCFinderX, it's limitations pose a threat to our data. CCFinderX's inability to detect Type-3 clones could possibly allow for a misinterpretation of the SCCR graph to see more refactoring than actually occurred. This would mostly affect our analysis of Git because we did not have the time to qualitatively analyze it and could not confirm notions on refactoring efforts. The other two projects saw thorough analysis of pivotal commit points which may have had this problem. Despite this, the data we gathered should still hold weight because there were not many refactoring points to consider in the first place, especially in Git. As mentioned earlier carefully analyzing the SLOC changes could still help distinguish these occurrences. Since our data can still potentially distinguish these occurrences and there are a means of confirming such suspicions, CCFinderX's limitations on detecting Type-3 clones should not be an issue. It should be noted that conducting a similar analysis again may benefit from the use of a clone detector which can detect Type-3 clones to mitigate the issue completely.

Although mostly being composed of C/C++ files, each project did contain files of significance to development from different languages, which could mean that development with different languages could be occurring simultaneously, making certain periods more stable because functionality is being added through different undetected means. Since all projects had in fact a majority of C/C++ files, this issue should not be very prevalent as the developers should be more likely to continue development in the language used most frequently for ease of compatibility.

Due to the differing organization structures of each project, we were not able to exclude libraries in our analysis in every project analyzed. Depending on the project, this could drastically change the clone ratios. Whether the libraries should or should not be included are up for debate, as they still do serve a purpose in development, but are not necessarily written by the developer themselves. Along with this, our analysis only cover 3 different projects which in hindsight seem to be very well developed based on their relatively low clone ratios. In order to fully understand the anomaly which is software development, an analysis of code clone ratios over the version evolution of more projects is necessary, in particular projects which have higher clone rates than our three. This will help us

distinguish different development patterns as well as understand exactly how libraries affect clone ratios.

VI. Related Work

Software clones has been a very popular topic of research in recent years, and many different studies have reported findings about code clone ratios as well as the effect of clones on software maintainability. Koshchke et al. report an average clone rate of 12% for open-source C and C++ programs [7]. The average clone rate has been reported differently among different studies, with Zibran et al. reporting a clone rate between 9% and 17% for programs written in C, C#, and Java [8]. Chen et al. reports a range of 4.6% and 24.9% when analyzing open-source games written in C, Java, and Python [9]. Clearly ratios differ among studies for different reasons, including the clone detection tool used, the languages that the analyzed software is written in, and the type of software which was analyzed. More research needs to be done to understand what these different metrics indicate about the presence of clone ratios in software.

As stated previously, the general consensus is that code clones can be harmful to software development due to maintainability problems and bug propagation, and this is the assumption used when qualitatively analyzing each of the three projects in this study. Juergens et al. observed software faults being induced from cloned code in various commercial and open-source programs [10]. Lague et al. looked specifically at a large telecommunication system and found several cases where cloned code has propagated to software bugs which the customer had experienced, and has proposed methods to prevent these faults from reaching the customers [11]. Lozano et al. analyzed method changeability with clones present, and reported higher costs in changing methods with code clones present [12]. Although several studies have shown detriments that come with the presence of code clones, some studies have observed that code clones can actually be beneficial to software, with Kapser et al. describing several benefits such as preventing instabilities in code through the use of code clones to introduce new features [2]. They also report that as many as 71% of clones have a positive impact on software maintainability. With the ongoing debate about how code clones actually effect software maintainability, it is important that further research is done to truly understand what harms or benefits cloned code can have on software to allow proper qualitative analysis of clones and to develop efficient software maintenance practices which make use of clone awareness [2].

VII. Conclusion

In this paper, we analyzed code clone ratios over the version evolution of three very different open-source projects. Our analysis primarily focused on the SCCR throughout version evolution in conjunction with the SLOC and CLOC at each particular commit point to understand different parts of the development process. With this data it was possible to determine the role which code clone refactoring played during development, as well as make inferences on code clone maintenance techniques. Each project displayed very different short term trends, but overall all projects showed a period of instability followed by a period of relative stability which may be attributed to project design not being concrete during the initial phase of development. Similar analysis of a developer's own software can help the developer understand the effectiveness of their code clone maintenance. From there, the developer can adjust their code clone maintenance techniques in order to better control code clone growth in their software.

Acknowledgment

This work was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) JP25220003, and also by Osaka University Program for Promoting International Joint Research.

References

[1] A. Sheneamer and J. Kalita, "Article: A survey of software clone detection techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, March 2016.

[2] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9076-6

[3] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. D. Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, Sept 2014, pp. 305–314.

[4] T. Kamiya, "Ccfinderx," http://www.ccfinder.net/.

[5] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Code clone analysis tool," in *Proceedings 1st International Symposium on Empirical Software Engineering*, vol. 2, 2002, pp. 31–32.

[6] M. Dagenais, E. Merlo, B. Laguë, and D. Proulx, "Clones occurence in large object oriented software packages," in *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '98. IBM Press, 1998, pp. 10–. [Online]. Available: http://dl.acm.org/citation.cfm?id=783160.783170

[7] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in c or c++," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, March 2016, pp. 1–7.

[8] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and forecasting near-miss clones in evolving software: An empirical study," in *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, April 2011, pp. 295–304.

[9] Y. Chen, I. Keivanloo, and C. K. Roy, "Near-miss software clones in open source games: An empirical study," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2014, pp. 1–7.

[10] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070547

[11] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *1997 Proceedings International Conference on Software Maintenance*, Oct 1997, pp. 314–321.

[12] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*, Sept 2008, pp. 227–236.