Concolic Analysis for Code Clone Detection

Daniel E. Krutz, Emad Shihab, and Samuel A. Malachowsky
Software Engineering Department
Rochester Institute of Technology, USA
{dxkvse, emad.shihab, samvse}@rit.edu

ABSTRACT

Source code is often duplicated throughout software projects. These duplicated functionality equivalent portions of an application are known as code clones. Clones may be problematic for several reasons, some of which include elevated maintenance costs and increased system faults. While there are a wide range of powerful techniques for finding these clones, most struggle at detecting the most complicated types of code clones, type-4 clones. In this work, we propose a new type of clone detection technique, which uses concolic analysis. Through a case study, we demonstrate the effectiveness of concolic analysis in clone discovery and its ability at identifying type-4 clones. We discuss our preliminary results and describe the future work we plan on conducting.

1. INTRODUCTION

Research has shown that between 7% and 23% of a software system is comprised of clones [16] [1]. This is likely very problematic since clones, and their inconsistent maintenance, have been shown to widely lead to a high level of system faults [7]. Clones also increase the maintenance costs of an application since the same alteration may need to be done multiple times in all of the cloned elements [12]. While there are four types of code clones, only two known techniques state that they are able to reliably detect type-4 clones. These clones are typically the most difficult to detect and may be much more problematic than the simpler types of clones [20] [15].

In a previous work, we created a clone detection tool based upon concolic analysis, called Concolic Code Clone Detection (CCCD) [11]. This tool demonstrated the effectiveness of concolic analysis in discovering clones in a small, controlled environment.

In this paper, we propose using concolic analysis to discover clones. This research is innovative because to our knowledge, no previous attempts have been made in using concolic analysis in clone discovery and any techniques which can effectively discover all four types of code clones are important since so few clone detection processes are able to do so. We want to present our early findings using concolic analysis for clone discovery and demonstrate our plan for future work. *[Emad says: We do not say what is new over the tools paper here.]*

ICSE '14 Hyderabad, India

1.1 Code Clones

There are four types of code clones which are generally recognized by the research community. Type-1 clones are the simplest types of clones and represent identical code except for variations in whitespace, comments and layout [3]. Type-2 clones contain variations in identifier types, but are otherwise syntactically similar. Type-3 clones contain altered or removed statements between two functionally equivalent code segments. Type-4 clones are two segments which are considerably different syntactically, but produce identical results when executed. Type-4 clones also represent the most difficult type of clone to detect [6] [4].

Code clones are usually viewed as being problematic for several reasons. Clones will likely elevate maintenance efforts since bug fixes and code adaptation may need to be done multiple times, along with the need for extra testing and verification activities. Program comprehension, code quality analysis and aspect mining efforts may also be increased due to the existence of code clones. Other uses for clone detection techniques include plagiarism analysis, software evolution analysis and virus detection [17].

1.2 Concolic Analysis

Concolic analysis combines concrete and symbolic values in order to traverse all possible paths of an application (up to a given length). Concolic analysis is a variation of symbolic analysis were concrete executions are simultaneously run with symbolic analysis. In order to examine application paths, solvers are generated which are used to generate new test input to direct the application along the various execution paths. This process is continued until all possible distinct paths have been reached using a depth search strategy [18]. The use of concrete values represents the primary advantage of using concolic analysis instead of symbolic analysis since constraints may be simplified which may assist in the precise reasoning of complex data structures. Concolic analysis has been traditionally used for testing due to its ability to traverse a large number of application paths [14]. *[Emad says: This paragraph sounds similar to the tools paper. Are you sure it is different?]*

2. DETECTING CLONES USING CONCOLIC ANALSYSIS

The initial step is to perform concolic analysis on the chosen application. This may be accomplished through the use of several existing concolic analysis tools such as CREST [1], Java Path Finder (JPF) [2], or CATG [3]. The generated concolic output is then separated at the method level, and then compared to other concolic method output in a round robin fashion so that all methods are compared to one another. Figure 1 presents the basic structure of CCCD.

[Dan says: Did I do a good enough job describing the concolic analysis process?] *[Emad says: I feel like you need a little more of a description here.]*

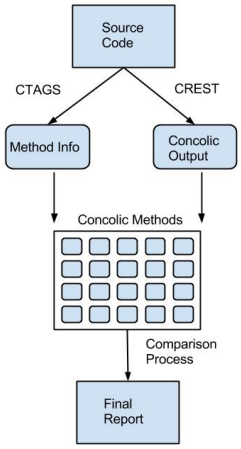

Figure 1: Overview of the CCCD Tool [11]

The Levenshtein distance algorithm *[Emad says: Levenshtein is a distance metric, not algorithm. Or is it?]* is used to evaluate the similarity of two sets of compared concolic output. We chose to use the Levenshtein algorithm to measure the similarity of the concolic output because of its ability to work with strings of different lengths and its restriction of upper and lower bounds in the calculated distances. In order to assist in normalizing the differences for strings of different lengths, the final calculated Average Levenshtein Value (ALV) is calculated by dividing the Levenshtein distance between the two compared methods (LD) by the longer of the two strings being compared (LSL) and then multiplying this value by 100. This formula is demonstrated in Equation 1. For our initial work, we chose to state that any two compared methods with a Levenshtein distance of 30 or less would be considered a clone. However, this value may be altered in future use. *[Emad says: Also, this seems very familiar from the tools paper. Ot am I just on crack?]*

$$ALV = (LD/LSL) \times 100 \quad (1)$$

Concolic analysis forms the basis of a powerful clone detection system because it only considers the functional paths of the examined source code. Similar functional paths represent a code clone candidate. Things such as comments, spacing and naming conventions with have been problematic for existing clone detection techniques have little effect on the ability of concolic analysis to detect clones. Since these items are irreverent, the amount of normalization which must be done using concolic analysis is minimal, especially when compared with many clone detection tools such as CCFinder and NICAD [17].

Our proposed technique does have two significant deficiencies. While future work may be done to mitigate this issue, our proposed method will only discover clones at the method level since this is where the concolic output is divided. This means that our technique will not discover a large number of clones that occur at a more granular level. Secondly, like all techniques, concolic analysis is not capable of discovering all types of clones in all situations. There are a large number of scenarios where concolic analysis will not discover clones. We do not believe this is an overly problematic issue for concolic analysis since all clone detection techniques suffer from this issue.

3. CASE STUDY RESULTS

In order to demonstrate the capability of concolic analysis for discovering code clones, we conducted an initial assessment using a self-created oracle. We then ran concolic analysis against this oracle recording types of clones discovered, false positives and false negatives.

3.1 Data

The initial phase of our case study was to select several open source applications to examine for code clones. We chose to examine Apache 2.2.14, Python 2.5.1 and PostgreSQL 8.5 since they have already been used in previous clone detection research [9]. These applications were selected as-is and had no alterations performed on their source code. *[Dan says: I feel like this section is a bit weak]*

3.2 Clone Oracle

The first step was to identify existing clones in these systems. In order to make this process more manageable, we chose to analyze a small, statistically significant, subset of each application selecting several random class to examine. Even though only a small subset of each application was analyzed, since every method would be compared against each other, this made the number of possible clones to check for exponentially large. For the three applications, there were a total of 45,109 possible clones to check for, which was not reasonable to inspect through manual analysis. *[Dan says: Actually, more were examined due to how each application had a stat sig value selected from them]* To make this number more manageable, we selected a statistically significant number of random clone combinations to examine with a goal of having a confidence level of 99% and confidence interval of 5. We then used an open source tool we created known as GraphicDiff [4] to automatically load the selected possible clone comparisons for manual analysis. This tool also allowed the user to record if the comparison was not a clone, and if so, what type of clone it was. Two researchers familiar with code clones independently analyzed the applications for clones. Any discrepancies with the findings were discussed until an agreement could be made.

3.3 Performance Measurement

[1] http://code.google.com/p/crest/
[2] http://babelfish.arc.nasa.gov/trac/jpf/
[3] https://github.com/ksen007/janala2
[4] https://code.google.com/p/dek-graphicdiff/

Once the clones were identified, the concolic analysis for clone detection process was ran on the selected subsections of the code. All comparisons with a Levenshtein distance under 30 were stated to be clones and were compared against the manual analysis. These results are shown in Table 3.3*[Dan says: fix table number]*.

[Dan says: double check these numbers to make sure they add up]

Table 1: Preliminary Results

	Apache	Python	P-SQL	Total
Type-1	1/1	0/0	0/0	1/1
Type-2	17/17	7/8	18/21	42/46
Type-3	0/0	8/12	14/19	22/31
Type-4	0/0	0/0	1/2	1/2
Clones Found	18	15	33	66/80
Clones Missed	0	5	9	14
False Positives	2	2	1	5
True Negatives	339	523	613	1475
Examined	359	545	656	1560
Accuracy	.99	.99	.98	.99
Recall	1	.75	.79	.85
Precision	.9	.88	.97	.92

These initial results are very encouraging. Concolic analysis was able to identify 66/80 or 82.5% of all identified clones in the system. Accuracy, recall and precision are also very high, especially for initial results. These values are all very favorable, especially when compared with results from existing clone detection studies [20]. For example, MeCC was reported to have a false positive ration of about 14.7% [9], while our technique was found to have an initial ratio of about 7.5%. *[Dan says: build on this? Find more apps to get values against?]*

The lack of discovered type-4 clones is not concerning since so few were found during manual analysis. This is largely due to the difficulty of manually discovering or identifying type-4 clones [19].

4. RELATED WORKS

There have been numerous works which have discussed methods of discovering code clones and their impact on software development. CCFinder [8] is a token-based clone detection tool which has served as the basis of numerous other clone detection tools, but is unable to reliably detect type-4 clones. Baxter *et al.* created a tree-based clone detection tool known as CloneDR which has been extensively used in research [2].

While most of proposed techniques have little difficulty in discovering simpler type-1 and type-2 clones, far fewer are able to discover more complicated type-3 and type-4 clones. Only two known works state the ability to discover the most complicated type of clones, type-4. Krawitz [10] proposed a clone detection process based upon functional analysis, but never implemented this technique into a functional tool. Kim *et al.* [9] created a tool known as Memory Comparison-based Clone Detector (MeCC) which was the first tool to explicitly demonstrate the ability of finding type-4 clones.

The impact of code clones on software development has also been examined. Juergens *et al.* [7] posed the question of if code clones mattered. This work found that clones are often inconsistently changed, are often created unintentionally and that inconsistent*[Dan says: inconsistent again?]* clones are often the source of a high level of system faults. However, this work only examined the impact of type-1, type-2 and type-3 clones on software development and did not explicitly address or separate type-4 clones.

This study is significantly different from that in our previous work introducing CCCD [11]. In this work, CCCD was discussed at a more technical level, and the amount of analysis on the tool was fairly minimal. Several clones of all four types as previously defined by Krawitz [10] and Roy *et al.* [17] were inserted into several open source applications, with their locations recorded. This work did not attempt to locate existing clones in any applications.

5. THREATS TO VALIDITY

[Dan says: proofread extra]

Even though our initial findings are promising, there are several threats to our results. Only three open source applications were selected for analysis. While we feel that these offer a good initial testbed, the ability to test and analyze more applications would have provided more confidence in our results. Additionally, since the clone detection technique only examines source code written in C, it is unproven how this process would work with other languages. The choice of the Levenshtein distance algorithm may also be problematic. While it has helped us to achieve promising results, it may not be the most appropriate similarity measurement method for the task.

While we did our best to create a robust and fair clone oracle, manually creating such an oracle is a difficult and imprecise task, even for the most experienced researchers [19]. We attempted to created an unbiased oracle, creating it before an analysis was run on the source code, however we are confident that it contains imperfections and clones that other researchers may disagree with. This is not a new problem as there is no clear consensus in the research community over the precise rules of what defines type-3 and type-4 clones [17] [5] [13] [9].

6. CONCLUSION

Code clones are likely to increase software development costs while concurrently reducing the quality of the final product. While various approaches have been demonstrated to be effective in discovering simpler types of clones, very few are able to reliably detect more complicated types. We see concolic analysis forming the basis of a new and powerful clone detection technique which is able to reliably detect all types of code clones. Our work is profound for two primary reasons. The first is that no other known techniques have attempted to use concolic analysis in discovering clones. Secondly, any technique which is able to reliably detect type-4 clones import since so few other techniques are able to accomplish this. Future work is required to help to further demonstrate this assertion as well show how this technique directly compares to leading existing tools. We would also like to find more type-4 clones in our oracle which concolic analysis and other tools may be evaluated against.

7. FUTURE WORK

While we have achieved very positive initial results, there is still a large portion of analysis to be conducted. The next action will be to compare concolic analysis for clone detection against other leading clone detection tools. Current candidates include CloneDR [5], Simian [6],iClones [7], and MeCC [8]. Performing this comparison will help to demonstrate the effectiveness of concolic analysis for clone detection in relation to leading existing tools. Some of the methods of comparison will include, but not be limited to accuracy, precision, recall, analysis time and number of each type of clone discovered.

The use of the Levenshtein distance metric for comparing the out of concolic analysis will also be examined. While this distance metric has helped us to produce encouraging initial results, we would like to see if there is a more appropriate metric for measuring the similarity of concolic output. Additionally, for the initial analysis we used a Levenshtein distance of 30 to determine if an examined method was a clone or not. In the future, we would like to explore the effects that different Levenshtein distance values would have on the clone discovery process.

Since we are unaware of any large, open source clone oracles which explicitly define all four types of code clones, we decided to create our own clone oracle. While we are confident this oracle helped us to achieve promising initial results, it can be improved upon. We were only able to initially have two researchers independently identify the methods that represented clones. We would like to include more researchers in this process, especially since identifying clones using a manual process is difficult even for experienced researchers [19]. Additionally, we would like to work to discover more type-4 clones. Once concolic analysis has been further demonstrated to be an effective clone detection process capable of reliably discovering type-4 clones, we would like to extend upon the work of Juergens *et al.* [7]and analyze how type-4 clones affect software development

8. REFERENCES

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[4] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. Xiao: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 369–378, New York, NY, USA, 2012. ACM.

[5]http://www.semdesigns.com/products/clone/
[6]http://www.harukizaemon.com/simian/
[7]http://softwareclones.org/iclones.php
[8]http://ropas.snu.ac.kr/mecc/

[5] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[6] N. Gold, J. Krinke, M. Harman, and D. Binkley. Issues in clone classification for dataflow languages. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 83–84, New York, NY, USA, 2010. ACM.

[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[9] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[10] R. M. Krawitz. *Code Clone Discovery Based on Functional Behavior*. PhD thesis, Nova Southeastern University, 2012.

[11] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, 2013.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, Mar. 2006.

[13] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDDÃŢ06*, pages 872–881. ACM Press, 2006.

[14] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[15] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENÕS UNIVERSITY*, 115, 2007.

[16] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 81–90, Washington, DC, USA, 2008. IEEE Computer Society.

[17] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[18] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[19] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 285–, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Y. Yuan and Y. Guo. Cmcd: Count matrix based code clone detection. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference*, APSEC '11, pages 250–257, Washington, DC, USA, 2011. IEEE Computer Society.