CCCD: Concolic Code Clone Detection

Daniel E. Krutz and Emad Shihab
Rochester Institute of Technology
{dxkvse,emad.shihab}@rit.edu

***Abstract*—Code clones are multiple code fragments that produce similar results when provided the same input. Prior research has shown that clones can be harmful since they elevate maintenance costs, increase the number of bugs caused by inconsistent changes to cloned code and may decrease programmer comprehensibility due to the increased size of the code base.**

To assist in the detection of code clones, we propose a new tool known as Concolic Code Clone Discovery (CCCD). This is the first known clone detection tool which uses concolic analysis as its primary component and is one of only three known techniques which are able to reliably detect the most complicated kind of clones, type-4 clones.

I. Introduction

Code clones may adversely affect the software development process for several reasons. Clones have the tendency to raise the maintenance costs of a software project since alterations may need to be done several times [5]. Additionally, unintentionally making inconsistent bug fixes to cloned code across a software system is also likely to lead to further system faults [2].

There are four types of code clones which are generally recognized by the research community. Type-1 clones are the simplest and represent identical code except for variations in whitespace, comments and layout. Type-2 clones are syntactically similar except for variations in identifiers and types. Type-3 clones are two segments which differ due to altered or removed statements. Type-4 clones are the most difficult to detect and represent two code segments which considerably differ syntactically, but produce identical results when executed [3].

In this paper, we propose Concolic Code Clone Detection (CCCD), a tool for finding code clones which uses *concolic analysis* as a driving force for discovering clones. Concolic analysis combines concrete and symbolic values in order to traverse all possible paths (up to a given length) of an application [7]. CCCD is innovative for several reasons. First, only two other works [6] [8] are able to effectively discover type-4 clones. Additionally, it represents the only known proposed technique for discovering clones which is based upon concolic analysis.

Concolic analysis assists in creating a powerful clone detection tool because it does not consider the syntactic properties of the source code of an application. Only the functionality is analyzed. This means that issues such as naming conventions and comments which have proved to be problematic for existing clone detection systems will have no adverse affect on CCCD.

The rest of the paper is organized as follows. Section II provides an overview of the proposed CCCD tool. Section III discusses the results of an evaluation of the tool and and section IV summarizes the findings and conveys future applications of the tool.

[make sure that the tool section does not feel like a bandaid of a

II. Tool Overview

CCCD is comprised of two primary phases. First a Unix bash script generates the necessary information for analysis. The concolic output is generated using an open source tool known as CREST [1]. CREST was selected since it was able to completely examine the benchmark applications and was able to analyze a variety of method types, regardless of the signature. Another open source tool known as CTAGS [1] assists in preparing the concolic output for analysis. The process of identifying code clone candidates is done using a comparison component written in Java and is invoked automatically from the Unix bash script. The concolic output for each method in the target application is compared to one another in a round robin fashion using the Levenshtein distance algorithm. The final report contains a listing of all code clone candidates as identified by CCCD. Figure 1 shows the basic components of the CCCD tool.

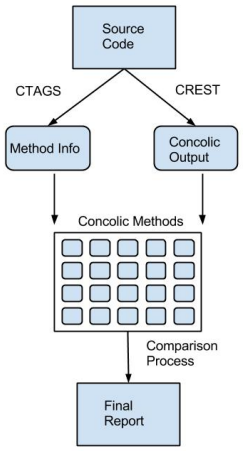

Fig. 1. Overview of the CCCD Tool

The tool and complete results may be found by visiting the main project website at http://www.se.rit.edu/~dkrutz/CCCD/.

[1] http://ctags.sourceforge.net

III. Evaluation

In order to evaluate CCCD, we first compare its performance on the code clone benchmarks provided by Krawitz [8] and Roy [9]. These works provided several explicit examples of all four types of clones. The initial step was to ensure that CCCD would be able to detect all of these predefined clones individually. A simple C application was created which contained the sixteen clones as defined by Roy and four as defined by Krawitz.

Several methods were inserted into this class which were not clones of any other methods in the class. The purpose of this was to help ensure that CCCD did not incorrectly identify methods to be clones which were not. This class was then analyzed by CCCD. Out of 465 comparisons, 296 were manually determined not to be comparisons between two methods which represented clones while 165 comparisons were manually determined to represent code clones. CCCD was then ran against the target source code. Comparisons with a Levenshtein similarity score of under 35 were deemed to be code clone candidates. These values were selected after several previous test runs with this source code, along with the source code from other applications.

CCCD was able to determine whether or not two methods were clones with an accuracy of 93%. An additional, 17 comparisons were recommended for further manual analysis (i.e., had a Levenshtein score close to 35). Another 14 comparisons should have been identified as clones, but were not. This is not considered to be overly concerning for CCCD. All of these non-identified clones were type-4 clones from the work by Krawitz. This was likely a problem that CCCD had with the specific type of clone method as laid out by Krawitz. Additionally, CCCD was able to identify the remaining type-4 clones as presented in the work by Roy. There were no false positives, meaning that all clone candidates identified by CCCD were manually verified to be actual clones.

Total Comparisions	465
Not clones	296 (65%)
Clones	165 (35%)
Correctly Identified	434 (93%)
Not Identified	14 (3%)
Recommended	17 (3.5%)
False Positive	0 (0%)
Avg. Leven Clones	12.7
Avg. Leven Non-Clones	58.4

TABLE I
Clone Detection Results

The next step was to ensure that these clones could be discovered in existing systems. Several open source applications were selected for this analysis. These included FileZilla [2], VLC [3] and MySQL [4]. Each of the predefined clones taken from the works of Roy and Krawitz were randomly inserted into the source code of these applications with their locations being noted. A complete listing of the results is shown in Table II.

Application	Type-1	Type-2	Type-3	Type-4	Total
VLC	5/5	6/6	7/7	6/8	24/26 (92%)
MySQL	5/5	6/6	7/7	6/8	24/26 (92%)
FileZilla	5/5	6/6	7/7	6/8	24/26 (92%)

TABLE II
Results of the Injected Clones by CCCD

CCCD was able to discover these clones with similar results as in the previously described control class. This analysis shows that concolic analysis used in CCCD is the same for each method, regardless of where it resides and its surrounding methods. The comparison process will therefore return similar results for these methods, regardless of what application they reside in.

IV. Conclusion and Future Work

CCCD has been shown to be a new, robust and effective technique for clone discovery. Preliminary work has demonstrated its effectiveness in discovering clones of all four types. This includes type-4 clones, which only two other techniques are able to reliably locate. In the future, CCCD will be applied to researching several other areas of computing including malware detection. Since new malware is only a slight variation of existing malware and significant portions of malware code are reused [4], a clone detection mechanism such as CCCD would likely prove to be a valuable asset in battling this malicious software.

[2]https://filezilla-project.org
[3]http://www.videolan.org
[4]http://www.mysql.com

References

[1] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Florian Deissenboeck, Benjamin Hummel, and Elmar Juergens. Code clone detection in practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 499–500, New York, NY, USA, 2010. ACM.

[3] Nicolas Gold, Jens Krinke, Mark Harman, and David 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.

[4] Mathur Idika. A survey of malware detection techniques. 2 2007.

[5] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan 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.

[6] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun 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.

[7] Yunho Kim, Moonzoo Kim, YoungJoo Kim, and Yoonkyu Jang. Industrial application of concolic testing approach: a case study on libexif by using crest-bv and klee. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1143–1152, Piscataway, NJ, USA, 2012. IEEE Press.

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

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