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). CCCD is the first known clone detection tool that 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 often raise the maintenance costs of a software project since alterations may need to be done several times [4]. 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. 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 significantly differ syntactically, but produce identical results when executed [3].

In this paper, we propose Concolic Code Clone Detection (CCCD), a tool 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 [6]. CCCD is innovative for several reasons. First, only two other works [5] [7] are able to effectively discover type-4 clones. Additionally, it represents the only known proposed technique for discovering clones which is based on 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 proven to be problematic for existing clone detection systems will have no adverse affect on CCCD.

II. Tool Overview

Figure 1 shows the basic components of CCCD. As shown, CCCD is comprised of two primary phases. The first step is to generate the necessary concolic output for analysis and is accomplished using two components which are invoked from a Unix bash script. The first component is an open source tool for generating the concolic output known as CREST [1]. CREST was selected since it was able to analyze a variety of function types, regardless of the signature. The second major component of the bash script is CTAGS [1] which is used to identify the functions in the source code of the target application.

The only modification which is done to CREST is with how it generates the necessary concolic output. Other than this, all default values are used and CREST is used in its native form. No details regarding the maximum path exploration length, main function for exploration or any other application details are needed. Only C programs are compatible with CCCD since CREST is only capable of analyzing C code.

The analysis phase is conducted using a component written in Java. This component separates the generated concolic output into individual function files using the list of functions generated by CTAGS as a guideline. Since the concolic output is separated at the function level, only function level clones are identified by CCCD. Code segments within functions or clones which partially traverse multiple functions will likely not be identified by CCCD.

Once the concolic information has been split into individual function files, the comparison process may begin. The concolic output for each function is compared to one another in a round robin fashion using the Levenshtein distance algorithm. Comparisons with a lower Levenshtein distance means the concolic output is more closely related, and thus indicative of a code clone candidate. The final report contains a listing of all code clone candidates as identified by CCCD.

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

III. Evaluation

In order to evaluate CCCD, we first compare its performance on the code clone benchmarks provided by Krawitz [7] and Roy *et al.* [8]. 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 *et al.* and four as defined by Krawitz.

Several functions were inserted into this class which were not clones of any other functions. The purpose of this was to

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

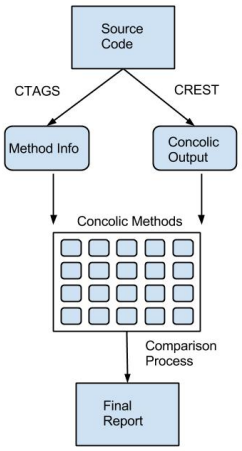

Fig. 1. Overview of the CCCD Tool

help ensure that CCCD did not incorrectly identify functions 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 functions which represented clones while 165 comparisons were manually determined to represent code clones. CCCD was then run 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. Higher Levenshtein scores were found to include too many false positives, while lower scores ignored a large number of clones. All results were manually verified by two researchers.

CCCD was able to determine whether or not two functions 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. These all interact with *Krawitz_type4* clones. This is due to CREST's inability to traverse all paths of the code, thus creating incomplete concolic output and therefore hindering the ability of CCCD to detect clones. Since CCCD was able to identify the remaining type-4 clones presented in the work by Roy *et al.*, this is not considered to be a concern for CCCD. There were not false positives, meaning that all clone candidates identified by CCCD were manually verified to be actual clones. These results are shown in Table I.

TABLE I
CLONE DETECTION RESULTS

Total Comparisons	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

The next step was to ensure that these clones could be discovered in several open source applications. These included FileZilla [2], VLC [3] and MySQL [4]. Each of the predefined clones taken from the works of Roy *et al.* and Krawitz were randomly inserted into the source code of these applications with their locations being noted. These results are shown in Table III.

TABLE II
RESULTS OF THE INJECTED CLONES BY CCCD

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%)

IV. CONCLUSION AND FUTURE WORK

This paper presented CCCD, a tool which uses concolic analysis to discover code clones. Preliminary work 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.

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] 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.

[5] 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.

[6] 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.

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

[8] 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.

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