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 work 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, a number of tools have been proposed. Some of which include CCFinder, MeCC, CloneDR and Clonetracker. The majority of these tools depend on the syntactic similarities of code clones. However, a large number of code clones (e.g., Type-3 and Type-4 clones) contain a significant amount of syntactic differences. In this work we propose a new tool called Concolic Code Clone Detection (CCCD), which uses *concolic analysis* to detect code clones. The advantage of this approach is that it discovers clones based on the functional not syntactic nature of the software. This means that numerous semantic issues which have plagued previous clone detection techniques do not have any adverse affects on CCCD. Based on our preliminary findings, CCCD has been highly effective in detecting clones of all types.

I. Introduction

Code clones occur in software for a variety of reasons. Developers may knowingly duplicate functionality across the software system. This may be due to an unwillingness to refactor and retest the modified portion of the application, or simply due to laziness on the part of the developer. Clones continue to be extremely widespread in software development. It is estimated that clones comprise between 5-23% of all source code [2] [18].

A significant amount of previous work states that code clones are undesirable [15] [6]. One of the most prominent reasons against code clones is that they substantially raise the maintenance costs associated with an application [10]. Reducing maintenance costs is a very serious matter since the maintenance phase of a project has been found to comprise between 40-80% of the cost of a software project [20] [7] [19] [21]. Finally, unintentionally making inconsistent bug fixes to cloned code across a software system is also likely to lead to further system faults [5].

Clone detection tools have been proposed to assist practitioners in detecting code clones so they may be easily detected. Such code clone tools have helped in discovering clone-related bugs and even security vulnerabilities in software systems [4].

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 [8].

There are numerous tools which discover clones in a variety of ways. While many are able to discover the simpler type-1 and type-2 clones, fewer have the ability to find type-3 and type-4 clones. Type-4 clones are typically the most difficult clones to detect and may be more problematic than type-1, type-2 and type-3 clones [17]. To the best of our knowledge, only two known techniques are able to discover the most complicated types of clones, type-4 [14], [16].

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. Concolic analysis has been traditionally used in software testing to find faults in an application [13]. Concolic analysis forms the basis for a powerful clone detection tool because it only considers the functionality of the source code and not its syntactic properties. This means that things such as comments and naming conventions which have been problematic for existing clone detection techniques will not affect concolic analysis and its discovery of clones. This tool is innovative for several reasons. First, only two other works claim the ability to discover type-4 clones, with each having some drawbacks. Additionally, it represents the only known proposed technique for discovering clones which is based upon concolic analysis.

While CCCD is not perfect, its ability to reliably discover type-4 clones is profound. Using clone benchmarks defined by prior research, CCCD was able to detect the majority of code clones (of all types) in both a single file as well as clones which were injected into several open source software applications.

The rest of the paper is organized as follows. Section II provides an overview of the proposed CCCD tool. Section III discusses the primary components of CCCD and how it was validated. Section IV briefly describes existing clone detection tools and some of the current uses of concolic analysis. Section V summarizes the findings and conveys future work to be done on CCCD, including some future applications of this tool.

II. The CCCD Tool

CCCD follows a traditional pipe-and-filter architecture. Figure 1 shows the basic components of the CCCD tool. 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 [3]. 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. Even though specific tools are described in the implementation of CCCD, these components are all easily interchangeable and newer or more robust technologies may be added to the tool as needed. The final report contains a listing of all code clone candidates as identified by CCCD.

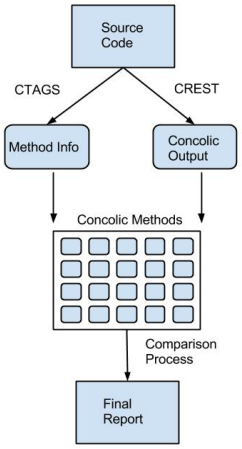

Fig. 1. Overview of the CCCD Tool

Data Generation

The initial step of CCCD is to generate the required information for analysis. An open source tool known as CREST was selected for generating the necessary concolic output. This tool was chosen because it is highly configurable, open source and robust. The bash component of CCCD first loops through the source code of the target application. CREST is executed against all source code files with a .c extension. The generated concolic output is stored in external text files for further analysis.

A goal of this clone detection tool is to discover code clone candidates at the method level. The next step is to separate the generated concolic output for each targeted class and create a separate file containing the concolic output of each method. In order to note the methods in the source code for each class, an application known as CTAGS [1] is used. This tool is able to quickly and efficiently note the methods in the source code for the target application. This method information is stored in an external file and is used to break the concolic output for each class into individual files for all methods inside of the class.

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

Comparison Process

Once all of the necessary data has been created (i.e., the conconlic analysis is run on the code), the comparison portion of the tool is then invoked to process this information and ultimately identify any code clone candidates. Using the generated information from CTAGS, the concolic output for each class is broken up for each method and is stored in a newly created file. Each method in the entire application will have an external file created for it.

After all of the concolic method files have been created, the comparison process begins. The goal is to record similar concolic output for any of the methods. This is accomplished by comparing each concolic output method file against others in a round robin fashion. In order to measure the similarity between two files, the Levenshtein distance measurement is used. The Levenshtein distance is defined as the minimal number of characters that would need to be replaced in order to convert one string to another [1]. While there are other string similarity techniques which may be viable options, the Levenshtein distance metric was found to be well suited for this clone detection task. In order to help normalize the similarity results based on length, the final Levenshtein score (ALV) is computed by dividing the Levenshtein distance between two files (LD) by the longest string length of the two strings being compared (LSL) and then multiplying by 100. The ALV formula is shown below in Equation 1.

$$ALV = (LD/LSL) \times 100 \tag{1}$$

The results of each comparison are stored in a final report .csv file. In order to make the generated information more manageable for the user, only comparisons with a Levenshtein score of 35 or less are added to the report. This is because our preliminary empirical analysis showed that comparisons with a distance of 35 or less yielded higher quality code clone candidates. More analysis is needed to determine the impact of the Levenshtein score on the performance of the CCCD tool. A brief representation of the output is shown in the Appendix (Table III).

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

III. Implementation and Evaluation

In order to evaluate CCCD, we first compare its performance on the code clone benchmarks provided by Krawitz [14] and Roy [16]. In their work, each of Krawitz and Roy defined benchmarks containing clones of all four types and laid out several explicit examples of each type. 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.

Another interesting finding is the wide gap between methods which were clones and methods that were not clones. Methods which were manually identified as clones had an average Levenshtein similarity score of 12.7 while non-clones had a much higher average score of 58.4. A listing of the results is shown in Table III. A more thorough list of the results may be found on the project website.

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 phase 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.

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.

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

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

IV. RELATED WORK

There are numerous clone detection tools which utilize a variety of methods for discovering clones. Some of which include text, lexical, semantic, symbolic and behavioral based approaches [16]. Only two known works claim the ability to reliably detect type-4 clones. MeCC discovers clones based on the ability to compare a program's abstract memory states. While this work was successful in finding type-4 clones, this tool suffers from several drawbacks including the ability to only analyze pre-processed C programs and an excessive clone detection time which is likely caused by the exploration of an unreasonably large number of possible program paths [12]. Krawitz [14] proposed a clone discovery technique based on functional analysis. This process was shown to detect clones of all types, but was never implemented into a reasonably functional tool. Additionally, this technique requires a substantial amount of random data which may be a difficult and time consuming process to produce.

V. 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 purport the ability to locate.

While a tool that employs concolic analysis as its driving force and is able to discover type-4 code clones is profound, there are significant opportunities for future work. CCCD will first be compared against existing leading clone detection techniques. Likely candidates include MeCC, CCFinder [11] and Deckard [9]. CCCD is likely able to find more type-4 clones than Deckard and CCFinder since previous research has noted these tools to struggle at finding the more complicated types of clones. Some of the reasons for the struggles include the inability to overcome semantic differences in the source code or problems with normalization [17].

Currently, CCCD only analyzes files with a .c extension. Source code located in other files, including .h, were not examined. In the future, CCCD will be expanded to explore these file types in order to discover clones in a wider spectrum of file types. Furthermore, future work will be done to analyze much larger data sets using CCCD and compare this new and innovative technique against leading existing clone detection tools. CCCD will also be applied in answering questions of how clones affect the software development and maintenance process.

REFERENCES

[1] Gregory V. Bard. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *Proceedings of the fifth Australasian symposium on ACSW frontiers - Volume 68*, ACSW '07, pages 117–124, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine 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] 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.

[4] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao 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] 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.

[6] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1):3:1–3:31, July 2010.

[7] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] 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.

[9] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[10] 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.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

[12] 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.

[13] 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.

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

[15] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *SIGAPP Appl. Comput. Rev.*, 12(3):20–36, September 2012.

[16] 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.

[17] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 115, 2007.

[18] Sandro Schulze, Sven Apel, and Christian Kästner. Code clones in feature-oriented software product lines. *SIGPLAN Not.*, 46(2):103–112, October 2010.

[19] Carolyn B. Seaman. Software maintenance: Concepts and practice. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(2):143–147, 2001.

[20] Ruchi Shukla and Arun Kumar Misra. Estimating software maintenance effort: a neural network approach. In *Proceedings of the 1st India software engineering conference*, ISEC '08, pages 107–112, New York, NY, USA, 2008. ACM.

[21] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 67–, Washington, DC, USA, 2002. IEEE Computer Society.

APPENDIX

CCCD is an open source tool and may be downloaded in a Fedora 14 Virtual Machine at http://www.se.rit.edu/~dkrutz/CCCD/. The complete source code for the tool may be found in a public SVN repository at http://code.google.com/p/dk-crest-java/. The following section describes the steps required to run CCCD.

A. Installation

The simplest way to attain CCCD is to download the Fedora 14 VM which CCCD is already configured and installed on. If the user wants to install CCCD in their own environment, they should download and install the following into their Linux environment:

1) CREST
2) CTAGS
3) Java 1.6
4) CCCD

Users taking this approach should refer to the component's respective website for installation assistance. Other packages required by these components may also need to be installed depending on the specific environment chosen by the user. CIL is a required component of CREST and will need to be slightly modified. In the cil/src directory, the file cil.ml will need to replaced by a modified version for CCCD. This modified version of the file, along with instructions, may be found in the public SVN under bashScripts/customfiles. CCCD may be installed by merely retrieving its latest build from its public SVN repository and placing the attained file structure under the user's HOME directory.

While CCCD will run in a variety of Unix and Unix-like environments, it has only been tested in Fedora 14 running Java 1.6. The primary environmental limiting factor for CCCD is the environments that CREST and CTAGS are able to properly run in.

B. Data Population

CCCD is able to process C based source code in its search for clones. No pre-processing is required. To begin the clone detection process, the user should place the raw source code in the sourceFiles/input/{appname} directory. Examples of this are represented in Figure 2 and Figure 3 assuming that "kraw" was the source file to be analyzed.

Since the CREST component of CCCD will attempt to analyze each C file individually, all necessary header files which are not located in the same directory as the file being analyzed need to be included as environmental variables. This may be done by using

Fig. 2. Inital Input Folder Example #1

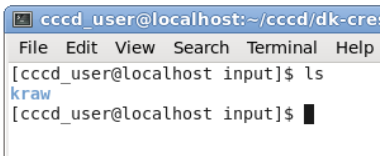

Fig. 3. Inital Input Folder Example #2

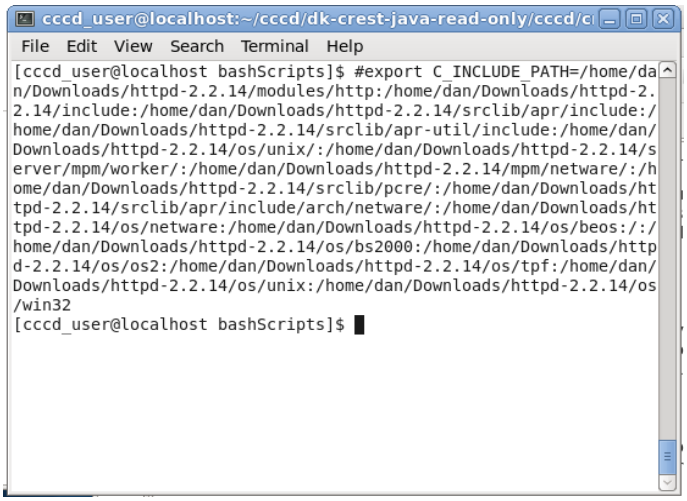

Fig. 4. Export_C Command

the ”export C_INCLUDE_PATH” command in Unix. More information on this command may be found at: http://www.network-theory.co.uk/docs/gccintro/gccintro_23.html. Another example of this command is shown in Figure 4

C. Running CCCD

Once the application to be examined has been copied to the appropriate input directory, CCCD may be executed. This is done by executing the following command from the bashScripts directory:

```
bashScripts$ ./cccd \{appname\}
```

An example of this command may be viewed in Figure 5

The appname parameter should exactly match the name of the folder which the source code was placed into in the sourceFiles/input directory. Once this command is invoked, the application will begin. Screen output will include information regarding file directories being generated,

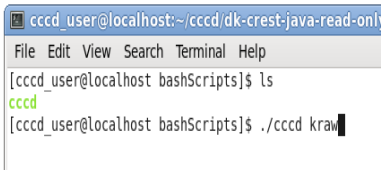

Fig. 5. CCCD Execute Command

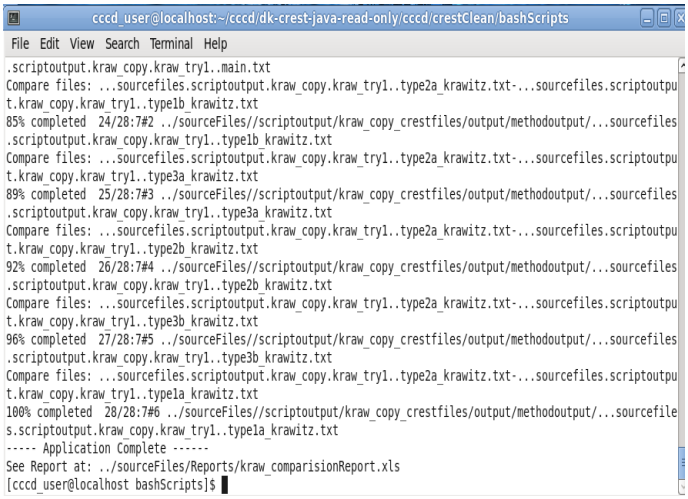

Fig. 6. CCCD Execution Window

Files	Leven Avg. Distance
kraw/type1a_krawitz()-kraw/type1b_krawitz()	0
kraw/type2a_krawitz()-kraw/type1a_krawitz()	11
kraw/type2a_krawitz()-kraw/type1b_krawitz()	11
kraw/type2b_krawitz()-kraw/type1a_krawitz()	1
kraw/type2b_krawitz()-kraw/type1b_krawitz()	1

TABLE III
EXAMPLE RESULTS SET

CREST and CTAGs information being created, the method comparisons and ultimately with the location of the final report. Depending on the size of the source code being analyzed, this entire process may take a few seconds to a few hours. An example of CCCD during execution may be viewed in Figure 6

D. Viewing Reports

The final report in .csv format may be viewed at sourceFiles/Reports/appname.csv . The report contains all method comparisons which were conducted using the computed Levenshtein average. Comparisons over this average are not shown in order to significantly reduce the amount of information displayed to the user and are comparisons which are highly unlikely to be code clones. The report is shown in the following format:

Table III represents the final result set created by CCCD. The first column displays the two methods which are compared with each other. The final column displays the calculated average Levenshtein distance.