Comparing the Effectiveness of Concolic Analysis for Code Clone Detection

XXX X XXX
xxxxx
xxxx, xx, xxx
xxxxx@xxx.xxx

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 because they potentially elevate maintenance costs, increase the number of bugs (due to inconsistent changes to cloned code), and decrease programmer comprehensibility (due to the increased size of the code base). This paper will outline a comparison process between a clone detection technique based on concolic analysis and leading clone detection tools.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques;

Keywords

Clone Detection, Concolic Analysis, Software Engineering

1. INTRODUCTION

Code clones occur in software for a variety of reasons including knowingly duplicating functionality, an unwillingness to refactor/retest the modified portion of the application, and simple 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 [1, 9]. They are generally considered to be problematic since they are likely to increase the maintenance costs and may lead to inconsistent bug fixes as the codebase increases in size, leading to continued system faults [3].

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 segments which differ due to altered or removed statements. Type-4 clones, the most difficult to detect, are code segments which considerably differ syntactically, but produce identical results when executed [4]. An example type-2 clone from Roy *et al.* [8] is shown in Table 1.

SAC'15 April 13-17, 2015, Salamanca, Spain.

Code Segment #1	Code Segment #2

`void sumProd(int n) {`
`  double sum=0.0;`
`  double prod =1.0;`
`  int i;`
`  for (i=1; i<=n; i++){`
`  sum=sum + i;`
`  prod = prod * i;`
`  foo2(sum, prod);`
`  }`
`}`
	`void sumProd2(int n) {`
	`  int sum=0; //C1`
	`  int prod =1;`
	`  int i;`
	`  for (i=1; i<=n; i++){`
	`  sum=sum + i;`
	`  prod = prod * i;`
	`  foo2(sum, prod);`
	`  }`
	`}`

Table 1: Example Type-2 clones from Roy

Detecting clones can be extremely difficult since the syntax of two clones may widely vary, and there are numerous techniques and tools available with varying levels of success for each of the clone types (1-4). Basic techniquess include text, tree, and symbolic based methods, and leading tools include Simian[1], Nicad[2], CloneDR[3], MeCC[4], and Simcad[5].

A previous work created a clone detection tool based on concolic analysis [6]. We propose a process for further verifying this technique and conducting a large scale analysis on the capabilities of concolic analysis for discovering code clones.

2. CONCOLIC ANALYSIS

Concolic analysis combines concrete and symbolic values in order to traverse all possible paths (up to a given length) of an application. It has traditionally been used in software testing to find faults in an application [5], but it also forms the basis of a powerful clone detection tool because it only considers the functionality of the source code - not its syntactic properties. This means that comments, naming conventions, and other non-syntactic parts of the sourcecode which are problematic for many existing clone detection techniques do not affect concolic analysis and its discovery of clones. Two well-known concolic analysis tools are Java Path Finder (JPF)[6] and jCUTE[7].

Concolic output from JPF is shown in Table 2. Constant variable types are represented generically by "CONST" and integer variable types are represented by a generic tag "SYMINT". Though not present in the example, other variable types are represented simi-

[1] http://www.harukizaemon.com/simian/
[2] http://www.cs.usask.ca/~croy/
[3] http://www.semdesigns.com/products/clone/
[4] http://ropas.snu.ac.kr/mecc/
[5] http://homepage.usask.ca/~mdu535/tools.html
[6] http://babelfish.arc.nasa.gov/trac/jpf/wiki
[7] http://osl.cs.illinois.edu/software/jcute/

Concolic Segment #1	Concolic Segment #2
sumProd1(a);	sumProd2(a)
PC # 3 = 3	PC # 3 = 3
CONST_3>a_1_SYMINT	CONST_3>a_1_SYMINT
CONST_2<=a_1_SYMINT	CONST_2<=a_1_SYMINT
CONST_1<=a_1_SYMINT	CONST_1<=a_1_SYMINT
SPC # = 0	SPC # = 0
PC # = 2	PC # = 2
CONST_2>a_1_SYMINT	CONST_2>a_1_SYMINT
CONST_1<=a_1_SYMINT	CONST_1<=a_1_SYMINT
SPC # =0	SPC # =0
PC # = 1	PC # = 1
CONST_1>a_1_SYMINT	CONST_1>a_1_SYMINT
SPC # = 0	SPC # = 0

Table 2: Diff of Concolic Output

larly; actual variable names do not appear anywhere in the output and are irrelevant to the concolic analysis technique.

3. CONCOLIC CODE CLONE DETECTION

The first step of concolic analysis for code clone detection is to generate the necessary concolic output at the method level, done using a tool such as jCUTE or JPF. After this is done for each method, the concolic output for each method is compared to all other output in a round robin fashion using the Levenshtein distance formula.

Since the concolic output is being generated at the method level, concolic analysis for code clone detection is only be able to discover clones at this granularity. This is done to ensure that the number of comparisons, which need to be made between each set of concolic output, is manageable. Future work may be done to allow analysis at a more granular level. Table 2 displays identical concolic output from the code clones shown in Table 1.

4. ANALYSIS PROCESS

Although a Concolic Code Clone Detection (CCCD) tool was proposed in previous research, it has not been compared against other concolic analysis tools to fully evaluate its effectiveness in discovering code clones. We will determine how it compares to existing tools in A) Their ability to discover all four types of code clones B) The accuracy, precision, and recall of their ability to find clones, and C) The time it takes for each tool to perform their analysis. We will present our results in the format similar to that shown in Table 3.

		Compared Tools		
	Concolic	**Tool**	**Tool2**	**Tool3**
Type-1	x/x	x/x	x/x	x/x
Type-2	x/x	x/x	x/x	x/x
Type-3	x/x	x/x	x/x	x/x
Type-4	x/x	x/x	x/x	x/x
Accuracy	x	x	x	x
Recall	x	x	x	x
Precision	x	x	x	x
Execution Time	x	x	x	x

Table 3: Results Format

In addition to further evaluation of concolic analysis for code clone detection, we will compare these results against leading clone detection tools including Simian, Nicad, CloneDR, MeCC, and Simcad - a list that is likely to grow in our final analysis. These tools and techniques will be evaluated using a previously created oracle by Murakami *et al.* [7], which is an extension of the oracle created by Bellon *et al.* [2].

All results will be published to our project website for researchers to analyze and use in their own research.

5. CONCLUSION

This paper presents the outline for a thorough analysis of the effectiveness of concolic analysis in code clone detection. We believe is necessary in order to evaluate concolic analysis for code clone detection in comparison with leading existing techniques in terms of accuracy, precision, recall, and execution time.

6. REFERENCES

[1] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, Nov 1998.

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

[3] F. Deissenboeck, B. Hummel, and E. 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.

[4] 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.

[5] Y. Kim, M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1143–1152, Piscataway, NJ, USA, 2012. IEEE Press.

[6] D. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 489–490, Oct 2013.

[7] H. Murakami, Y. Higo, and S. Kusumoto. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 412–415, New York, NY, USA, 2014. ACM.

[8] 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.

[9] S. Schulze, S. Apel, and C. Kästner. Code clones in feature-oriented software product lines. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 103–112, New York, NY, USA, 2010. ACM.