A Code Clone Oracle

Daniel E. Krutz and Wei Le
Rochester Institute of Technology
1 Lomb Memorial Drive
Rochester, NY 14623, USA
{dxkvse,wei.le}@rit.edu

ABSTRACT

Code clones are functionally equivalent code segments. Detecting code clones is important for determining bugs, fixes and software reuse. Code clone detection is also essential for developing fast and precise code search algorithms. However, the challenge of such research is to evaluate that the clones detected are indeed functionally equivalent, considering the majority of clones are not textual or even syntactically identical. The goal of this work is to generate a set of method level code clones with a high confidence to help to evaluate future code clone detection and code search tools to evaluate their techniques. We selected three open source programs, *Apache*, *Python* and *PostgreSQL*, and randomly sampled a total of 1536 function pairs. To confirm whether or not these function pairs indicate a clone and what types of clones they belong to, we recruited three *experts* who have experience in code clone research and four students who have experience in programming for manual inspection. For confidence of the data, the experts consulted multiple code clone detection tools to make the consensus. To assist manual inspection, we built a tool to automatically load function pairs of interest and record the manual inspection results. We found that none of the 66 pairs are textual identical *type-1* clones, and 9 pairs are *type-4* clones. Our data is available at: http://phd.gccis.rit.edu/weile/data/cloneoracle/.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Maintaining software, Reusability, Software Evolution

Keywords

Code Clone Detection, Software Engineering, Clone Oracle

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597127

1. INTRODUCTION

Code clones are functionally equivalent code segments. The clones may differ in whitespace, comments and layout (which we call *type-1* clones), in identifiers and types (*type-2*), or in altered and removed statements (*type-3*). There are also *type-4* clones, where the only conditions are to ensure the same output for the same given input [10]. Detecting clones may also help to find bugs, determine inconsistent bug fixes and locate redundancies in code search results [8, 10].

There are many techniques to detect code clones [10]. When measuring the effectiveness of new or existing tools in terms of precision, recall, or their abilities to discover different types of clones, we need an *oracle* to automatically answer whether a clone found is indeed functionally equivalent and which type the clone belongs to. Currently, researchers either manually confirm results or compare the clones found with the results reported by an existing tool [3,6]. The problem of such ad-hoc approaches is that different tools and manual inspection processes may report inconsistent clones and as a result, lead to imprecision in the evaluation. Even worse, it can take researchers a significant amount of time to perform such an evaluation.

The goal of this work is to create a clone oracle, which consists of a set of code clones created using a mixture of human and tool verification to increase confidence. We recruited three *experts* who have experience in code clone detection and four students who have experience in programming to manually determine code clones for a set of function pairs. The results from a set of leading clone detection tools are used for helping make the consensus during manual analysis. To facilitate manual inspection, we built a tool to automatically load function pairs of interest and record the manual inspection results. This tool can be used by future researchers to inspect the clones.

We selected three open-source programs *Apache*, *Python* and *PostgreSQL* for our studies. We constructed 1536 function pairs from randomly selected classes. We identified a total of 66 pairs of clones from the three programs, among which none is type-1 clones, 43 are type-2 clones, 14 are type-3 clones and 9 are type-4 clones.

The rest of the paper is organized as follows. Section 2 describes the methodology of creating the clone oracle. Section 3 presents the data we generated, including the distributions of the code clones in the three programs and clone types. In Section 4, we discuss the limitations of the data. Section 5 provides an overview of related work, followed by a conclusion in Section 6.

2. ORACLE CREATION

In this section, we describe our approach of creating the clone oracle.

2.1 Selecting Function Pairs

We aim to create an oracle of clones at the method level. We initially constructed a set of function pairs as candidates of clones. We chose three real-world programs *Apache 2.2.14*[1], *Python 2.5.1*[2] and *PostgreSQL 8.5*[3]. Ideally, we would compare each method from a program against all other methods in the same program for clones. However, such approach requires implausible manual effort to confirm clones, as the number of function pairs generated is $MethodCount * (MethodCount - 1)/2$. Another option is to randomly select any two functions from the program; but the chance where two randomly selected functions are clones is low.

Therefore, our approach is to randomly select 3-6 classes from each application and enumerate all the possible function pairs for the classes. Using this approach, the total number of function pairs generated for the three applications is 45,109. We applied code clone detection tools for all the function pairs. We then randomly selected a statistically significant number (a confidence level of 99% and a confidence interval of 5) of clones for manual studies, which resulted in a total of 1536 function pairs, 357 for *Apache*, 545 for *Python*, and 634 for *PostgreSQL*.

2.2 Determining Clones

Once the function pairs are established, we ran a set of publicly available clone detection tools, including Simcad [12], Nicad [9], MeCC [3] and CCCD [5]. We found that these tools are inconsistent for many of the clones detected.

We next perform manual inspection on the function pairs to determine clones and their types. We recruited three *experts* who have research experience with code clones and four students with programming experience. The experts discussed results during studies, and the clones are confirmed only when the consensus is made. Students inspect clones independently. We present all the manual inspection results in our dataset. Our goal for having the two groups is to improve the confidence of the data and present more information for future researchers to consider. A secondary goal is to compare how much the findings of the expert and student groups differed from one another.

Expert Group. The expert group performs the manual analysis without knowing the results from clone detection tools. Then the compared results from tools to make final decisions. There were numerous discrepancies between the experts during the manual analysis phase. In order to help mitigate these disagreements as to whether or not two methods are clones, or even what type of clones they represent, results from the various tools were used by the researchers to assist with the decision making process. Using these results as input, discrepancies were discussed until an agreement is made. There were cases however, where no agreement could be made with these results being recorded as *unsure* in the final result set.

[1]http://www.apache.org
[2]http://www.python.org
[3]http://www.postgresql.org

If several of the clone detection tools indicated a clone, where none was noted during manual analysis, the reviewers re-evaluated the candidate clone pair to ensure they had not overlooked anything. The same process was used in reverse when the tools indicated no clone, but manual analysis had found a clone pair. Ultimately, the final decision on a possible clone pair was made by the researchers, not by any tool.

Student Group. A group of four students examined the same set of function pairs to provide a larger, more diverse set of results for consideration. These students are upper division software engineering students who had no prior experience with code clones. To help familiarize students with code clones, they were asked to read papers by Roy *et al.* [10], Kim *et al.* [3], and Lavoie and Merlo [6]. Finally, all students were independently interviewed to ensure that they understood code clones at an acceptable level.

The students did not discuss their results or come to a conclusion with other student examiners. They were not provided with any results from clone detection tools or from the expert group and were only asked to identify clones but not their types.

2.3 Tool to Help Manually Inspect Clones

We developed an open source tool *CloneInspection* [4] to assist with the manual clone identification process for both the expert and student groups. This tool automatically displayed each of the methods to be compared and allowed the user to select if the comparison represented a clone, and if so, what type. Once the user finished examining a function pair, they can easily navigate to the next function pair. In Figure 1, we show a screen shot of the *CloneInspection* tool.

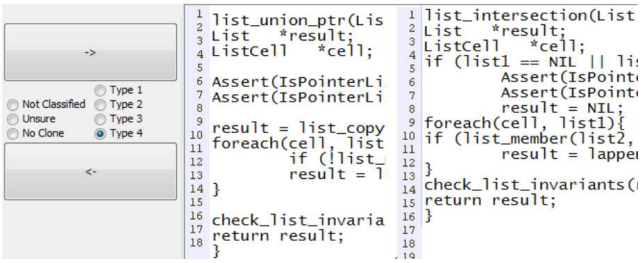

Figure 1: the *CloneInspection* tool

3. CLONES DISCOVERED

In this section, we first report the number of code clones discovered in this study and their types. We then provide an explanation on how the data available on our website should be interpreted.

3.1 Clones and Their Types

In Table 1 under *Expert*, we show the number of clones and clone types where the expert group makes consensuses. Under *Student*, we show the number of function pairs that at least half of the students identified as clones. Under *Agree*, we list the number where the experts and 50% or above students agree they are clones.

[4]https://github.com/cloneoracle/

As shown in Table 1, we discovered clones from all of the three programs. We found 66 pairs of clones, 4.3% of the total of 1536 function pairs examined. The majority clones found are type-2; however, we also find a total of 9 type-4 clones. No type-1 clones were noted, which is not surprising since developers can usually recognize exact duplications of code and would have removed them from the software.

Under *Student*, we show that individual students disagree significantly on what is a clone. If we increase the threshold from 50% to 75% and report a clone if 75% students agree, we would see much less clones identified by the students. Under *Agree*, we see that the students agreed with 12 out of 15 pairs in *Python* and 33 out of 33 pairs in *PostgreSQL*, but only 7 of the 18 clones in *Apache*. Interestingly, *Python* and *PostgreSQL* contain type-3 and type-4 while *Apache* only contains type-2 clones, which typically are considered easier to confirm compared to type-3 and type-4 clones.

Table 1: Clones Identified in Oracle

Application	Clone	Expert	Student	Agree
Apache	**T1**	0		
	T2	18		
	T3	0		
	T4	0		
	Total	18	23	7
	Not Clone	339	334	303
Python	**T1**	0		
	T2	7		
	T3	4		
	T4	4		
	Total	15	34	12
	Not Clone	530	522	473
PostgreSQL	**T1**	0		
	T2	18		
	T3	10		
	T4	5		
	Total	33	107	33
	Not Clone	601	550	459
Total	**Clone**	66	164	46
	Not Clone	1470	1406	1208

In Table 2, we display an example of a type-4 clone in *PostgreSQL* reported by an expert group and also classified by 50% of the students as a clone. The code segments both implement the union of the two lists. The two segments use different function names and invoked different calls, e.g., *list_member_ptr* in the first segment, and *list_member* in the second segment. The majority statements in the two functions are different. None of the existing code clone detection tools reported it as a clone.

3.2 Data Available on the Project Website

Both our tool and data are available on our website[5] in several formats including html, csv, xml and xls. Table 3 shows an example of the publicly available data. Under *Comparison*, we list function pairs under study. The *Expert* column displays the type of clone agreed upon by the experts and a *No* if a clone was not found. The *Student* column shows the percentage of students who determined the pair is a clone. Finally, columns *Tool 1* and *Tool 2* report whether a specific tool determines if the pair is a clone. The table includes the results for all the 45,109 function pairs collected. Since only a subset of function pairs are selected for manual studies, the function pairs not included in the manual studies are left blank under *Expert* and *Student*. In addition to the detailed results shown in Table 3, we also report 66 pairs of code clones on the website. This value is sufficient since most previous works have only used 4-20 clones in evaluating clone detection tools [4, 10]

[5]`http://phd.gccis.rit.edu/weile/data/cloneoracle/`

Table 3: Example Results Output

Comparison	Expert	Student	Tool 1	Tool 2
MethA-MethB	Type-1	100 %	No	Yes
MethB-MethC			No	No
MethA-MethB	Type-3	75 %	Yes	Yes
MethD-MethE	No	0	No	No

4. LIMITATIONS OF THE DATA

Although our decision making process was guided in a variety of ways, there may be clones in our dataset that are not actually code clones, as manually determining code clones is not a trivial task [13].

The oracle we created only identified clones at the method level. While many clone detection tools are capable of only identifying clones at the method level, others may find them at a more granular level [10]. Work may be done to create an oracle at the sub-method level.

In our studies, we did not find type-1 clones in the real-world code. However, such data are easily generated by adding white space and comments in the code and changing the code layout. On the other hand, type-1 clones can be determined by a compiler parser, and it is the types 2-4 clones that are valuable in the code clone oracle.

Finally, our oracle is C-based, meaning that the it will be of no use to clone detection tools which analyze code of other languages. Our technique is still very useful since a large portion of existing detection tools are C-based [10]. We can extend our methodology to create oracles for other languages in the future.

5. RELATED WORK

Existing code clone data either are produced using only one tool, contain a small number of code clones, or do not include real-world type-4 clones. Our data set is produced with the considerations of a diverse set of clone detection tools and manual analysis, and thus have a higher confidence level. In addition, we studied over 1000 function pairs and produced 66 pairs of clones, including real-world type-4 clones, and thus our data set is also more complete.

Krawitz [4] and Roy *et al.* [10] both explicitly defined clones of all four types in a small controlled environment. However, these works only specified a small number of clones which were artificially created. In 2002, Bailey and Burd [2] formed a manually verified clone data set which was used to compare three of the leading clone techniques at the time. This data has been criticized due to its validation subjectivity and its relatively small size. Bellon *et al.* [1] compared a set of code clone detection tools using a single researcher

Table 2: An Example of Type-4 Clones from PostgreSQL

Code Segment #1

```
list_union_ptr(List *list1,List *list2)
  List    *result;
  ListCell    *cell;

  Assert(IsPointerList(list1));
  Assert(IsPointerList(list2));

  result = list_copy(list1);
  foreach(cell, list2){
    if(!list_member_ptr(result,lfirst(cell)))
      result=lappend(result,lfirst(cell));
    }
  check_list_invariants(result);
  return result;
}
```

Code Segment #2

```
list_intersection(List *list1, List *list2)
  List    *result;
  ListCell    *cell;
  if (list1 == NIL || list2 == NIL)
    return NIL;

  Assert(IsPointerList(list1));
  Assert(IsPointerList(list2));

  result = NIL;
  foreach(cell, list1){
    if(list_member(list2,lfirst(cell)))
      result=lappend(result,lfirst(cell));
  }
  check_list_invariants(result);
  return result;
}
```

to manually verify the clones, but never publicly released all the discovered code clones [10].

Li et al. [7] and Saebjornsen *et al.* [11] created a clone data set using clones identified by software developers. However, it is possible that developers only reported a small portion of the clones in the system since manually identifying code clones in a real world software system without the assistance of a tool is an extremely difficult and imprecise process. Lavoie and Merlo [6] described an automated technique of constructing a clone data set based on the Levenshtein metric. While this is a powerful, automated technique for producing clones in large data sets, this is the only process used to create the oracle and but does not generate any verified data. Their approaches do not handle type-4 clones.

6. CONCLUSIONS

Code clone detection is important for bug finding, fixes and code search. In this paper, we describe the methodology and data for a code clone oracle which may be used by future researchers. The data are agreed by a group of experts with an assistant of a set of leading code detection tools. The 66 pairs of discovered code clones, the experimental data and tool are available on our website http://phd.gccis.rit.edu/weile/data/cloneoracle/.

7. REFERENCES

[1] 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.

[2] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 36–, Washington, DC, USA, 2002. IEEE Computer Society.

[3] 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.

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

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

[6] T. Lavoie and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, 2011.

[7] J. Li and M. D. Ernst. Cbcd: cloned buggy code detector. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 310–320, Piscataway, NJ, USA, 2012. IEEE Press.

[8] 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.

[9] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, 2008.

[10] 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.

[11] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[12] M. Uddin, C. Roy, and K. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238, 2013.

[13] 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*, 2003.