2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Challenges in Behavioral Code Clone Detection

Fang-Hsiang Su, Jonathan Bell, and Gail Kaiser
Department of Computer Science
Columbia University
New York, NY 10027
Email: {mikefhsu, jbell, kaiser}@cs.columbia.edu

***Abstract*—When software engineering researchers discuss "similar" code, we often mean code determined by static analysis to be textually, syntactically or structurally similar, known as code clones (looks alike). Ideally, we would like to also include code that is behaviorally or functionally similar, even if it looks completely different. The state of the art in detecting these behavioral clones focuses on checking the functional equivalence of the inputs and outputs of code fragments, regardless of its internal behavior (focusing only on input and output states). We argue that with an advance in dynamic code clone detection towards detecting behavioral clones (i.e., those with similar execution behavior), we can greatly increase the applications of behavioral clones as a whole for general program understanding tasks.**

I. Introduction

Software developers are often tasked to work with unfamiliar code that they do not understand. A developer might not know what the code is supposed to do, how it works, how it is supposed to work, how to fix it to make it work, but documentation might be poor, outdated, or non-existent. Furthermore, knowledgeable co-workers are not always immediately available to help. We believe that if the developer can be presented with "similar" code – code that is known (by the IDE) to execute in the same way or compute the same function, the developer might 1) find the similar code more readable; 2) identify familiar code to transfer her existing knowledge to the code at hand; 3) gain insights from a different structure that (the IDE tells her) implements the same behavior.

Numerous researchers have investigated code that exhibits *static* – textual, syntactic or structural – similarity. That is, code that looks alike. Visual differences might range from whitespace, layout and comments to identifiers, literals and types to changed, added and removed statements (sometimes referred to as type 1 to type 3 *"code clones"*, resp. [3]). Type 4 is a catch-all for all semantically similar clones, but it lacks scientific formulations to classify them. Static detection techniques attempt to match the source code text [11] or some static internal representation such as tokens [1], abstract syntax trees [2], [9], program dependence graphs [15], [16], or a combination of multiple program representations [4]. [7] found static similarity at the assembly code level. But static techniques cannot always detect code that **behaves alike**, i.e., that exhibits similar *dynamic* (runtime) behavior, but does not **look alike** [10], [12]. When the developer does not understand the code she needs to work with, showing her more code that looks about the same may not be helpful. But explaining that certain other code that looks quite different actually behaves very similarly could provide clues to what her own code does and how it works. Thus tools that detect and manage statically similar code may miss opportunities to help developers understand execution behavior [14].

Some researchers have investigated how to detect code that exhibits *dynamic* – executional, functional or behavioral – similarity. Dynamically similar code might perform the same computation from a functional input/output perspective [5], [10], [17], perform the same transformation of application state [8], or produce outputs satisfying the same constraints [13]. The corresponding source code may be expressed in structurally or conceptually different ways. If a developer is having difficulty understanding what the subject code is supposed to do or how it works, it could help to examine dynamically similar code: A different structural or conceptual approach might provide exactly the insight needed.

While previous work has equated dynamic similarity with some conception of functional equivalence, we propose a broader view. Our key insight is that any behavioral representation with an appropriate comparison metric might be used to detect code whose dynamic characteristics would be considered "similar" by human developers. Therefore, we raise the question: what makes two code fragments "behaviorally similar?"

II. Discussion

While most previous work considers code fragments to be behaviorally similar if they are functionally equivalent, we argue that this is an overly-strong definition. Code may perform a similar computation, but with dissimilar outputs. To support developers in understanding how code works, we seek to detect code that behaves similarly. We see static similarity as $Sim(Code_1, Code_2) \geq thresh$ and define dynamic similarity as $Sim(Exec(Code_1), Exec(Code_2)) \geq thresh.$

Consider for example the three code samples shown in Figure 1. Listings 1 and 3 take an array as input and return its sum; Listing 2 returns two times the sum. Note that Listing 2 is very similar in terms of apparent behavior, although it is not functionally equivalent. We believe that all three listings represent a behavioral clone group, although with existing definitions, either 1 and 3 are clones (for techniques that consider inputs and outputs), or only 1 and 2 (for techniques that consider abstract syntax trees).

```
int magic1(int[] ar){
  int sum = 0;
  for(int i=0;i<ar.length
    ;i++)
    sum += ar[i];
  return sum;
}
```

Listing 1. Code Sample 1

```
int magic2(int[] ar){
  int sum = 0;
  for(int i=0;i<ar.length
    ;i++)
    sum += 2 * ar[i];
  return sum;
}
```

Listing 2. Code Sample 2

```
int magic3(int[] ar){
  int sum = 0;
  int i = 0;
  try{
    while(true)
      sum += ar[i++];
  }
  catch(ArrayIndexOutOfBoundsException ex){}
  return sum;
}
```

Listing 3. Code Sample 3

Fig. 1: **Three code samples, all which behave similarly.**

We propose similarity detection techniques based on *runtime behavior* (*i.e.,* instructions executed) as an alternative model for identifying code that is behaviorally similar. Our idea is rooted in the relative success of techniques that introduce *characteristic vectors* as signatures to represent the static properties of code fragments [9], [19]. However, no purely static approach could detect all behavioral clones, since in the general case, code's behavior is unknowable until its execution. Instead, we propose to use dynamic signatures such as feature vectors of execution traces [6] enable detection of code that results in similar execution behavior regardless of its static composition.

III. Applications and Conclusions

Maalej et al. [18] conducted a comprehensive qualitative and quantitative study of how developers practice program comprehension. Four of the eight comprehension strategies identified either directly involve seeking "similar" code or potentially could be enhanced by similar code from the same codebase and/or an open-source repository: Interact with UI to test expected program behavior; Debug application to elicit runtime information; Clone to avoid comprehension and minimize effort; Identify starting point for comprehension and filter irrelevant code based on experience. Three of their six knowledge needs and channels seem relevant to similar code: Source code is more trusted than documentation; Standards facilitate comprehension; Cryptic, meaningless names hamper comprehension. Maalej et al. did not observe any use of comprehension tools and said developers seem unaware of them, and concluded by calling for reconsidering research agendas towards context-aware tool support.

We believe that dynamic code clone tools can answer this call. For example, detecting functionally or behaviorally similar code could augment UI interactions with the subject code or debugger elicitation of its runtime information. Functionally or behaviorally similar code could provide additional starting points for comprehension besides the subject code. The similar source code might be trusted more than documentation, better comply with standards than the subject code, or use less cryptic more meaningful identifiers. We believe advances in dynamic clone detection will open doors for applying code clones to program understanding in ways previously not possible. Showing the developer code that looks similar may not be helpful for understanding the code at hand, but showing code that behaves similarly and/or computes a similar function could provide valuable clues to what her code does and how it works.

IV. Acknowledgements

The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1302269 and CCF-1161079.

References

[1] B. S. Baker. On Finding Duplication and Near-duplication in Large Software Systems. WCRE '95.
[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. ICSM '98.
[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, Sept. 2007.
[4] C. De Roover, T. D'Hondt, J. Brichau, C. Noguera, and L. Duchien. Behavioral similarity matching using concrete source code templates in logic queries. PEPM '07.
[5] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. CSMR '12.
[6] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM TACO*, Jan. 2012.
[7] S. Ding. Kam1n0-Plugin-IDA-Pro. https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro.
[8] R. Elva and G. T. Leavens. Semantic Clone Detection Using Method IOE-behavior. IWSC '12.
[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. ICSE '07.
[10] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. ISSTA '09.
[11] J. H. Johnson. Identifying Redundancy in Source Code Using Fingerprints. CASCON '93.
[12] E. Juergens, F. Deissenboeck, and B. Hummel. Code Similarities Beyond Copy & Paste. CSMR '10.
[13] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing Programs with Semantic Code Search. ASE '15.
[14] Ko, Andrew J. and DeLine, Robert and Venolia, Gina. Information needs in collocated software development teams. ICSE '07.
[15] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *8th International Symposium on Static Analysis*. Springer-Verlag, 2001.
[16] J. Krinke. Identifying Similar Code with Program Dependence Graphs. WCRE '01.
[17] D. E. Krutz and E. Shihab. CCCD: Concolic Code Clone Detection. WCRE '13.
[18] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the Comprehension of Program Comprehension. *ACM TOSEM*, Sept. 2014.
[19] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting Similar Java Classes Using Tree Algorithms. MSR '06.