Applying Symbolic Execution to Semantic Code Clone Detection

Kazusa Takemoto
Dept. of Info. and Comp. Sci., Keio University
kazusa@doi.ics.keio.ac.jp

Shingo Takada
Dept. of Info. and Comp. Sci., Keio University
michigan@ics.keio.ac.jp

Abstract

Many approaches have been proposed to detect code clones, which are basically similar code fragments. Most approaches are based on textual similarity. These approaches cannot detect semantic code clones, which are clones that have the same functionality but implemented with different syntax. Two functions can be considered to have the same functionality, when the output is the same given the same input. In order to appropriately generate inputs, we propose applying symbolic execution to semantic code clone detection. These functions are executed to obtain outputs, which are compared to determine if function pairs are clones. Our approach also does not limit output to return values; we also handle arrays and pointers as output, as the execution of the function may cause changes in their values. Furthermore, we classify types to enable cases where the types of inputs and/or outputs are not exactly the same. We evaluate our approach on SemanticCloneBench.

1. INTRODUCTION

Software developers will make what are called *code clones*, which are basically similar code fragments. There are four types of code clones, where the first three are focused on syntax [3], and the last one is focused on semantics [8]. Type 1 clone is an exact copy without modifications (except for white space and comments). Type 2 clone is a syntactically identical copy, where only variable, type, or function identifiers are changed. Type 3 clone is copy with further modifications, such as changing, adding, or removing statements. Type 4 clone is two or more code fragments that perform the same computation but are implemented by different syntactic variants. Type 4 code clone is also called semantic code clone, and is the focus of this paper.

The presence of code clones incurs unnecessary costs in maintenance. Manually detecting code clones is time-consuming, so much work has been done on automatically detecting them [1][9]. Many of these approaches compare lexical units or graphically represent the structure of the code and compare the graphs. Such approaches often fail to find semantic code clones. Semantic code clones have also been confirmed in actual software development[5], and need to be addressed.

We focus on the fact that the same input returns the same output when the functions are the same. We detect semantic code clones by comparing the inputs and outputs. In order to generate an appropriate set of inputs, we consider input generation as test case generation and use symbolic execution to generate them. It is also necessary to deal with code clones with different types and to deal with functions other than return values in the comparison. We thus propose an approach for detecting semantic code clones by comparing input/output for each execution path using symbolic execution, taking into account differences in types and outputs having effects other than return values.

Section 2 describes related work and issues. Section 3 describes our proposed approach. Section 4 evaluates our approach, and section 5 makes concluding remarks.

2. RELATED WORK

Several approaches have been proposed for detecting semantic code clones by representing the structure of codes using graphs, e.g., PDG or CFG, and determining whether the code is a semantic code clone or not based on their similarity [1][11][10]. These approaches assume that PDG (or CFG) show the semantics of the code, and do not consider input/output. However, if two functions use a completely different algorithm, their respective PDGs will be different, and thus these approaches cannot find them.

On the other hand, Li et al [6] uses a test-based approach to detect clones in Java. They first filter out APIs that have differing input (parameter) and output (return) types. They further limit their search to methods with similar method names. In addition, the test execution uses branch coverage-based testing to obtain input-output pairs for a single target code, executes each method using its inputs, and compares the outputs of the approaches. The comparison is terminated when test cases with different outputs are found, in which case that method would not be a clone. The issue

DOI reference number: 10.18293/SEKE23-070

with this approach is that (1) the branch coverage based approach may not obtain all possible execution paths, (2) for each pair of methods under consideration, the approach only executes the test cases derived from the given target method, (3) it does not consider outputs other than the return value, and (4) it cannot detect cases where the parameter and return value types between methods are different. These issues can be summarized as follows:

Issue A: All execution paths are not considered.

Issue B: Test cases are generated only for the given target method and not for the candidate method.

Issue C: Side effects other than explicit outputs (returns) are not considered.

Issue D: Methods with differing parameter types and return types are not considered.

3. PROPOSED APPROACH

3.1. Overview of the proposed approach

We propose an approach where we apply symbolic execution to semantic code clone detection. Our proposed approach targets the C language. Given a set of functions, our approach outputs pairs of semantic code clones.

Our approach first uses symbolic execution to generate test cases (input-output pairs) for each execution path. It then executes each pair of functions against each other with the generated test cases. The output results for each function pair given the same input is checked. The percentage of matching results is given, where 100% indicates that the pair is a semantic code clone. Note that each function pair executes each other's test cases. In some cases, our approach also compares functions having differing input (parameter) and output (return) types. All test cases are executed and results are compared without terminating, even when test cases executions have different outputs. Because of this, our approach is expected to have a longer execution time compared to Li's approach [6].

3.2. Features of the proposed approach

Our proposed approach has the following three features to overcome the four issues given in the previous section[1].

Feature α To address Issues A and B, our approach compares the execution paths of function pairs to cover the full range of execution paths, thus detecting whether the input/output corresponding to execution path is an exact match or not, as well as similar but not exact match.

[1]Note that the previous section discussed Java methods, but we consider C functions.

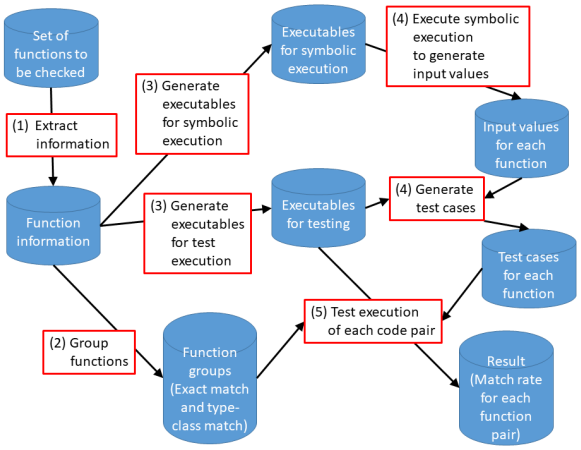

Figure 1. Process flow of our approach

Our proposed approach uses symbolic execution to generate test cases for each function, taking into account the execution path. We use symbolic execution rather than other test generation approaches, as it can theoretically cover all possible execution paths. In addition, when comparing the test cases, the function pair exchanges test cases and perform test execution with each other's test cases to compare the outputs. The outputs are compared to ensure an exhaustive comparison of input/output.

Feature β To address Issue C, if the parameter is an array or pointer, its value is also considered as output.

Although the return value is the primary output for a C function, the execution of a function may result in side effects. For example, if a parameter is an array or a pointer, its value may change as a result of operations performed on it within the function. So, if this array or pointer is accessed after this function call, its contents may have been modified, causing a side effect on the program. Thus, we also consider these as output.

Feature γ To address Issue D, we incorporate type classification to enable the detection of semantic code clones with different input/output types.

Two functions may be the same except for the parameter types and return types. We also consider these as semantic code clones. In order to handle the difference in types, we classify types.

3.3. Process Flow

The process flow of our approach is divided into the following five main parts (Figure 1):

1. Information extraction
2. Function grouping

3. Executable file generation for symbolic and test execution
4. Test case generation through symbolic execution and test execution
5. Test execution of each pair and output of match rates

Information extraction This part extracts and saves the basic information used in subsequent parts. Specifically, signature information of functions (function name, parameter types, and return type), structs (struct name, and each member's name and type), and typedefs (type name and type) are extracted using the *ctags* command [2].

Function grouping The second part groups the functions using the extracted parameter types and return type. The order of the parameters are not considered in our current implementation. This results in the following two groups:

- Exact match group: Each exact match group contains functions whose return value types, number of parameters, and parameter types exactly matches.
- Type-class match group: Types are classified and grouped to detect semantic code clones with different parameter and return types. We shall call such classified types as *type-class*. Even if two types are technically different, if they belong to the same type-class, then they will be treated as if they were the same type. Currently, we employ three type-classes as follows:
  - Numeric type-class: int, long, float, etc.
  - Character type-class: char, signed char, unsigned char
  - Set type-class: array, pointer

Note that functions that belong to the same exact match group will also belong to the same type-class match group.

Executable file generation for symbolic and test execution The information that were extracted in the first part is used to generate executable files for symbolic execution and test execution of each function. The former is used to generate inputs, that will be used by the latter to obtain the outputs. The corresponding input-output pair will be used as test cases. In our current implementation, the length of the area pointed to by arrays and pointers is set to 100.

For symbolic execution, a C function is generated and compiled that uses the library of the symbolic execution tool. In our implementation, we use KLEE-Float [7], which is a symbolic execution tool based on KLEE [4]. The key points of the generated C file is to set variables that are parameters of functions to be analyzed during symbolic execution, and then to call those functions.

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

For test execution, a C function is generated and compiled that takes inputs from standard input, calls the function to be checked, and outputs the execution result to standard output. Note that the inputs are the values that are generated through symbolic execution.

Test case generation through symbolic execution and test execution The generated executables in the previous part is used to conduct symbolic execution and test execution.

First, symbolic execution is done with the executable generated in the previous part. This results in execution paths which are further analyzed by KLEE resulting in input values corresponding to the execution paths. The number of generated input values is limited to a maximum of 100 for each function to prevent the overall execution time from becoming too long.

Next, test execution is done using the input values that were generated with symbolic execution. For each function, a test run is performed with the generated input values as inputs, and the input-output pairs are saved as test cases. Our current implementation limits the test execution time to two seconds.

Test execution of each pair and output of match rates Tests are now conducted on functions that are to be checked for the possibility of being semantic code clones. Specifically, all pairs within groups (exact match group and type-class match group) are checked. Note that the check is done two-ways. For example, if a developer wants to check if `functionA` and `functionB` are clones, the previous part is conducted to obtain `testCasesA` and `testCasesB`, respectively. The inputs in `testCasesA` are given to `functionB`, and the inputs in `testCasesB` are given to `functionA`. The outputs are checked against the corresponding test cases, and the following formula is used to calculate the match rate (MR):

$$MR[\%] = \frac{Number\ of\ test\ cases\ with\ matching\ output}{Number\ of\ test\ cases\ used\ for\ comparison} \times 100 \quad (1)$$

A semantic code clone should have a 100% match rate.

4. EVALUATION

4.1. Research Questions

We pose the following three research questions:

- RQ1: How well can semantic code clones be detected?
- RQ2: Can outputs other than return values be considered?

- RQ3: Can our approach support semantic code clones of different input/output types?

4.2. Evaluation Dataset

The evaluation uses SemanticCloneBench [2]. This is a benchmark consisting of semantic code clones from code that exists on Stack Overflow, and was collected based on human judgment. For C language, it contains 1000 pairs of semantic code clones. Since it includes pairs that are out of scope of our tool, we extracted 91 pairs for our evaluation based on a set of conditions. Table 1 shows the conditions, as well as the number of pairs removed due to not satisfying our conditions.

The vast majority of the functions in SemanticCloneBench were `main` functions, which our approach does not handle. This is because code clone search is normally done within a project, which will likely not have multiple cloned main functions.

Table 1. Conditions of target pairs

Conditions of target pairs	Number of deleted pairs
Not main function	720 pairs
Input/output is parameter/return value	47 pairs
Struct does not have a circular structure	34 pairs
Function is executable	76 pairs
Parameter type is supported by our tool	32 pairs

4.3. Result

Of the 91 pairs, 72 pairs were type matched by function grouping. Of the 72 pairs, one pair could not be checked as execution of test cases ended with errors or the execution time exceeded the limit of two seconds for each test case. Table 2 shows the evaluation results for the remaining 71 pairs, where MR denotes the match rate. The total execution time of the tool was 1353 seconds.

Table 2. Results

Match rate	RQ1	RQ2	RQ3
$MR = 100\%$	33	25	15
$80\% \leq MR < 100\%$	5	3	1
$50\% \leq MR < 80\%$	12	8	1
$30\% \leq MR < 50\%$	5	5	1
$MR < 30\%$	16	12	6

4.4. RQ1: How well can semantic code clones be detected?

Column "RQ1" in Table 2 shows the distribution of the 71 pairs in terms of match rate. 33 pairs out of 71 pairs had a match rate of 100%, and can be considered as semantic code clones. As described in section 4.2, SemanticCloneBench [2] is a benchmark containing pairs of functions that were considered to be clones based on human judgment. We expected that the number of pairs with a 100% match rate would be the majority. However, this was not the case. We further manually analyzed the results to understand what caused the differences.

We first manually checked the results when the match rate was 100%. We found that 6 of the 33 pairs were actually not completely the same, i.e., the match rate should not have been 100%. One example is a pair of functions that perform Caesar cipher encryption. One function performs Caesar cipher encryption for the letters A to Z, while the other function performs Caesar cipher encryption for the numbers 0 to 9 in addition to the letters A to Z. Therefore, this pair can be considered to be similar but is partially different, and the match rate should not have been 100%. The reason this occurred was because one of the parameters did not affect the execution path, and thus only one value was generated for that parameter. Multiple values needed to have been generated for that parameter for the difference between the two functions to appear. This can be considered as an issue with using symbolic execution. We are currently considering this as part of future work.

For pairs having a match rate between 80% and 100%, a common issue were cases that had "special" input values. One example was a pair of functions that returned a Fibonacci number. Both functions have the same output value for inputs greater than or equal to 2. But the output was different when the input was less than or equal to 1.

For pairs having a match rate between 50% and 80%, we found cases where the function itself is the same, but the range of input values it can handle differs. One example was where one function supports up to 32-bit input values while the other supports only 16-bit values.

For pairs having a match rate between 30% and 50%, we found cases where one function would execute correctly under specific conditions. In one pair, one of the function would execute correctly even if a string had two or more spaces, but the other would not.

Note that all of these pairs were considered to be semantic code clones as they came from SemanticCloneBench. Furthermore, there are several definitions of semantic code clones, where the core is that (1) the clones have the same functionality, and (2) are implemented with different syntax [2]. All of these pairs had the same basic function, but many had differences, which could be due to exceptional values, input range, or conditions. 100% match rate is a very strict application of "same functionality", and our results strongly suggests that the definition of semantic code clone may need to be refined depending on the scenario. At the same time, as our approach outputs match rate, the broadening of the semantic scope may be easily attained.

4.5. RQ2: Can outputs other than return values be considered?

Our approach considers outputs that do not take the form of return values. Specifically, our approach can also handle arrays and pointers as outputs if they appear as a parameter. Column "RQ2" in Table 2 shows the number of cases where the output included a parameter.

In some cases, although the array or pointer was handled correctly, the function pairs had differing return values causing the match rate to be low. One such example was a pair of functions that convert numeric values to strings. The two functions were implemented recursively, returning the result of the execution as a return value. The functions assumed that the parameter values (strings) were to be used as the "actual" output, and not necessarily the return value itself. Even if the resulting parameter values were the same, the return values were not always the same. This resulted in the match rate to be low. In order to deal with such cases, we need to take into consideration such factors as the partial match of outputs.

4.6. RQ3: Can our approach support semantic code clones of different input/output types?

Our approach classified types into type-class to enable finding semantic code clones even if the types do not exactly match. Column "RQ3" in Table 2 shows the number of cases where the input and/or the output types differed between the two functions, showing the importance of being able to handle such cases.

4.7. Threats to validity

First, the number of target pairs used in the evaluation is a threat. SemanticCloneBench has 1000 pairs of functions. However, most of them were main functions or used standard input/output, which our approach does not handle. In order to strengthen the generality of our evaluation, it is necessary to conduct evaluation on a larger data set.

Second, the execution time of symbolic execution, the execution time of tests, the length of the area pointed to by arrays and pointers, and the number of test cases generated for each function were all fixed. The experimental results may differ if these parameter values are changed.

Third, there is the possibility of human error when manual analysis was done in RQ1. We took great care to limit this possibility.

5. CONCLUSION AND FUTURE WORK

We proposed an approach that used symbolic execution to detect semantic code clones. The novel part of our approach is (1) the comparison of the execution paths of function pairs to cover the full range of execution paths, (2) the ability to handle outputs other than return values, specifically pointers and arrays, and (3) type classification to enable cases where input/output types are not exactly the same. Evaluation using SemanticCloneBench showed the viability of our approach.

Future work includes the following. First, we need to consider the definition of semantic code clone. Our definition is currently strict, i.e., the input/output needs to match 100% between functions. Second, we need to consider cases where the number of parameters differs. Third, we need to consider cases where a parameter does not affect the execution path, but will affect the variety of inputs needed to more correctly obtain the match rate. Finally, we need to consider other types of outputs, e.g., global variables and print functions such as `printf()`.

References

[1] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A Systematic Review on Code Clone Detection. *IEEE Access*, 7:86121–86144, 2019.

[2] F. Al-Omari, C. K. Roy, and T. Chen. SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge. In *IWSC 2020*, pages 57–63, 2020.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI 2008*, pages 209–224, 2008.

[5] V. Käfer, S. Wagner, and R. Koschke. Are there functionally similar code clones in practice? In *IWSC 2018*, pages 2–8, 2018.

[6] G. Li, H. Liu, Y. Jiang, and J. Jin. Test-Based Clone Detection: an Initial Try on Semantically Equivalent Methods. *IEEE Access*, 6:77643–77655, 2018.

[7] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle. Floating-Point Symbolic Execution: A Case Study in n-Version Programming. In *ASE 2017*, pages 601–612, 2017.

[8] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[9] A. Walker, T. Cerny, and E. Song. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *SIGAPP Appl. Comput. Rev.*, 19(4):28–39, 2020.

[10] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis. In *ASE 2020*, pages 821–833, 2020.

[11] Y. Zou, B. Ban, Y. Xue, and Y. Xu. CCGraph: a PDG-based code clone detector with approximate graph matching. In *ASE 2020*, pages 931–942, 2020.