2017 24th Asia-Pacific Software Engineering Conference

Interface Driven Code Clone Detection

Md Rakib Hossain Misu*, Kazi Sakib†
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: bsse0516@iit.du.ac.bd*, sakib@iit.du.ac.bd†

***Abstract*—Code cloning is a common code reusing technique that occurs when developers replicate similar pieces of code fragments within or between software repositories. Another replication happens when developers repeat method interfaces (i.e., method name, return and parameter types). Two methods are prone to be cloned when those have similar interfaces and perform similar functionalities. Considering this, a new lightweight Interface Driven Code Clone Detection (IDCCD) technique is proposed, that can detect clones by using method interface similarities. First, the method blocks are tokenized from the source files. For those method block tokens, interface information is extracted and indexed with mapped tokens. Then, similar interfaces are queried from that index and compared those with a similarity function for detecting clones. IDCCD is evaluated with other state of the art techniques by using BigCloneEval framework. The experimental results show that IDCCD performs similar comparing to other existing tools with a lower complexity.**

Keywords-Code Clone, Clone Detection, Interface

I. Introduction

Code clones occur when developers replicate codes within or between the software repositories through copying and pasting, automatic code generation or plagiarism. A common replication that appears in large software repositories is method interface. It refers the return type, method names and parameter types of a method that repeats exactly or similarly. If two methods contain the similar interface, it is very likely that those perform analogous functionalities. It indicates that these methods should be semantic or syntactic code clone to each other. So, interface similarity should have significance for detecting clones.

Clone detection tools and techniques differ from many aspects such as what type of detection algorithm is used, how the source code is represented to operate and how various clones can be detected. Textual based techniques use string matching algorithms that are perfect for detecting exact clones but do not work faster for larger dataset, and thus face scalability issues. For example, Cordy et al. applied the Longest Common Subsequence (LCS) algorithm in their tool called NiCard [1] for an efficient text line comparison to find nearly mismatched clones. AST based techniques [2] are useful for refactoring of clones, but they may not scale well as parse trees contain high memory. Token based techniques gain high recall but may yield clones which are not syntactically complete.

Sajnani et al have introduced a token based approach called SourcererCC [3] that works faster for large code repositories. They have used sub block overlapping and token positioning heuristic to reduce code fragments for efficient comparison.

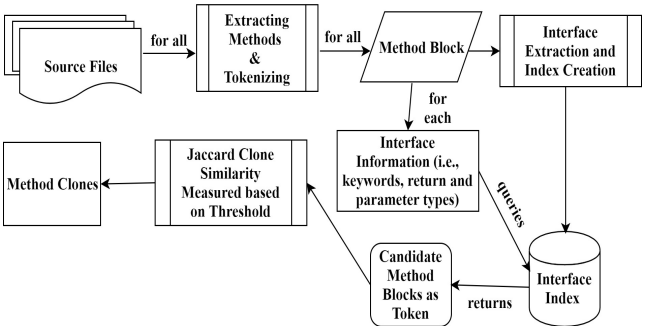

Figure 1. Overview of Interface Driven Clone Detection Process

Similar to SourcererCC [3], Jefferey et al. have proposed a flexible technique CloneWorks for detecting various types of clones [4]. Both SourcererCC [3] and CloneWorks [4] compute sub-block overlapping tokens for indexing that needs extra mathematical calculation and execution time before making actual method fragment comparison.

In this paper, a light weight Interface Driven Code Clone Detection (IDCCD) technique is proposed. This technique can detect clones by using interface information. First, source files are tokenized into method blocks and interface information (i.e., keywords, return and parameter types) is extracted. An inverted index is built using that interface information and mapped into the method block tokens. For each method block, similar interfaces are queried from that index and compare those with a similarity function. Finally, pairs of method blocks are reported as clone, if those satisfy a minimum similarity threshold. The experimental results are promising that yields to use interface similarities for detecting clones.

II. Overview of IDCCD

Figure 1 represents the three steps of IDCCD such as (i) Token Generation, (ii) Interface Index Creation and (iii) Clone Detection. A brief description of each step is mentioned below.

In token generation step, first source files are transformed into AST representation by using *Eclipses ASTParser*. Method blocks are extracted by traversing AST nodes. The extracted method blocks are tokenized with a simple scanner, that is aware of token and block semantics of a given language. Currently, this tokenization only works for Java.

In the index creation step, interface information (i.e., method name, return and parameter types) are extracted from method blocks. After that, keywords are extracted from the method names by removing stop words and performing stemming. Then, synonyms and antonyms of those keywords are identified by using standard WordNet Library. Finally, for each

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.97

method block it provides interface information including (i) return type, (ii) keywords from method name (iii) a set of synonyms and antonyms of those keywords and (iv) parameter types. Next, this information is used to build an inverted interface index that maps tokens of the method block.

In clone detection, for each method block, IDCCD retrieves the candidate method block tokens from the interface index by querying. Since the interface index is built using interface (i.e., keywords, return and parameter types), only the interface of those method blocks is used to query into the index. The interface index only results those method block tokens that are similar to query interface. As a result, detection comparison is performed among the method block tokens for which interfaces are similar. After getting all candidate method blocks, clone detection is performed by modified Jaccard similarity metric that is also used by CloneWorks [4]. It first takes a pair of method blocks as a set of tokens, and computes minimum token intersection ratio. Finally, a pair of method blocks is reported as method clone, if these blocks satisfy user given similarity threshold.

III. Evaluation

To evaluate IDCCD performance, recall and precision are calculated using a clone evaluation framework namely BigCloneEval [5]. It contains a standard clone benchmark called BigCloneBench [5].

A. Experimental Setup

BigCloneEval evaluates the performance of the clone detection tool with recall and precision using BigCloneBench. BigCloneBench contains manually validated clone pairs. It includes a repository called IJDataset2.0[1] containing 25K open source Java projects with 3 million source files and 250 MLOC. All types of clones such as Type-1 (T1), Type-2 (T2) , Type-3 (T3) and Type-4 (T4) present in BigCloneBench. However, there is no agreement on, when a clone is no longer syntactically similar. As a result, it is difficult to separate T3 and T4 clones [5]. So, researchers categorized IJDataset's T3 clones into Very Strongly Type 3 (VST3) and Strongly Type 3 (ST3) based on the similarity threshold. VST3 and ST3 clones contain 90% and 70-90% syntactical similarity. TABLE I represents the number of clones present in BigCloneBench.

For experiment, standard configuration parameters are set to minimum 6 lines and greater than 50 tokens with 70% similarity threshold. IDCCD is run in BigCloneEval's VM with an average workstation (e.g., 3.26GHz quad-core i7, 8GB ram and 500GB drive). Comparative results are also generate and presented in TABLE II.

B. Result

Recall measured by BigCloneEval is summarized in TABLE II. It is summarized using various types such as intra-project, inter-project, T1, T2 and T3 category (e.g., VST3, ST3). IDCCD had perfect detection of T1 clones with 100% recall. For detecting T2 clones, it achieved near perfect recall of 98%.

[1]IJaDataset 2.0, http://secold.org/ projects/seclone

TABLE I. BigCloneEval Clone Summary

Clone Type	T1	T2	VST3	ST3	Intra-Project	Inter-Project
Clone Pairs	48,116	4,234	4,577	9,569	47,852	17,265

TABLE II. Recall Comparison of IDCCD with Stat-of-the-art Techniques

Tool	Clone Type						
	T1	T2	VST3	ST3	Intra Project	Inter Project	Precision
IDCCD	100	98	96	81	98	88	84
CloneWorks	100	100	100	92.5	96	84	87
SourcererCC	100	98	93	61	96	82.75	83
NiCad	100	100	100	95	99.75	98.25	56
Deckard	60	54	62	31	56.5	49.5	28

In VST3 and ST3, it gained 96% and 81% recall similar to other tools. For intra-project and inter-project clones, IDCCD performed well and was able to gain 98% and 88% recall. Comparing to the other tools, IDCCD had the third best recall overall, with NiCad taking the lead. In case of precision, IDCCD also performed better comparing to others and got the second best precision 84%. However, NiCad [1] used LCS algorithm that can only compare two potential clones at a time. Since each potential clone needs to be compared with all of the others, making the comparisons using LCS was very expensive. On the other hand, SourcererCC [3] and CloneWorks [4] need extra mathematical calculation and execution time to index sub-block overlapping. In IDCCD, no extra calculation is needed, since the number of similar interfaces in a code repository is smaller than the total number of interfaces, the use of query in the interface index reduces the number of candidate method block comparison with lower time complexity comparing to NiCad [1], SourcererCC [3] and CloneWorks [4]

IV. Conclusion

Code clone detection using interface information has never been performed before. In this paper, an Interface Driven Code Clone Detection (IDCCD) approach is developed that can detect clones by using interface information (e.g., method name return and parameter types). After tokenizing method blocks and indexing its interface information, IDCCD compares those method blocks that have similar interfaces. Finally a pair of method blocks are reported as clone satisfying a similarity threshold. IDCCD gained an acceptable recall and precision. Comparing the execution time of IDCCD with other tools and scalability testing are potential future directions of this work.

References

[1] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 219–220.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[3] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 1157–1168.

[4] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 27–30.

[5] C. K. Roy and J. Svajlenko, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *Proceedings of the Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 596–600.