Enhancing Quality of Code Clone Detection with Program Dependency Graph

Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{higo, kusumoto}@ist.osaka-u.ac.jp

Abstract

At present, there are various kinds of code clone detection techniques. PDG-based detection is suitable to detect non-contiguous code clones meanwhile other detection techniques are not suited to detect them. However, there is a tendency that it cannot detect contiguous code clones unlike string-based or token-based technique. This paper proposes two techniques to enhance the PDG-based detection for practical usage. The software tool, Scorpio has been developed based on the techniques.

I. Introduction

The existing code clone detection tools can be categorized based on their detection techniques. Major categories should be line-based, token-based, metrics-based, AST-based, and PDG-based. Each detection technique has merits and demerits, and there is no technique that is superior to any other techniques in every way [2].

The advantage of PDG-based detection is that it is suitable to detect non-contiguous code clones meanwhile other detection techniques are not good at detecting them [2]. Non-contiguous code clones are ones whose elements are not consecutively located on the source code. However, PDG-based detection has two disadvantages: the first one is that PDG-based detection tends not to detect contiguous code clones [2]; the second one is that it is not realistic to apply PDG-based detection to middle- or large- scale software systems because it is time-consuming [3], [4].

This paper proposes two techniques to improve the first disadvantage. The techniques are intended to detect significant code clones from software systems, and introduce an implementation of the techniques.

II. Detecting Code Clones using PDGs

The detection process consists of the four steps (, which are derived from the method of Komondoor et al. [3]):

STEP1: All nodes in PDGs are hashed based on properties of the PDG node content. Nodes having the same hash value are classified as an equivalence class.

STEP2: Every pair ($r1$,$r2$) of nodes are selected from every equivalence class, and two isomorphic subgraphs that include $r1$ and $r2$ are identified. Both forward and backward slices are used to identify isomorphic subgraphs. The starting points of the slices are ($r1$, $r2$), and slicing is performed in lock step. If the both predecessors or the both successors are in the same equivalence class, they are added to the pair of slices. The two slices are isomorphic subgraphs, which are clone pairs detected in this paper.

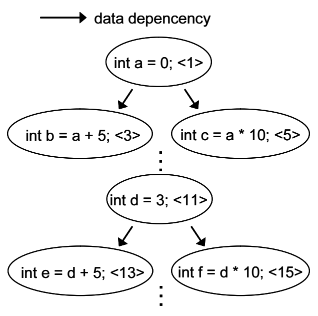
STEP3: If a clone pair ($s1$, $s2$) is subsumed by another clone pair ($s1'$, $s2'$) ($s1 \subseteq s1' \cap s2 \subseteq s2'$), it is removed from the set of detected clone pairs.

STEP4: A clone set is generated from clone pairs sharing the same isomorphic subgraphs. For example, two clone pairs ($s1$, $s2$), ($s2$, $s3$) would be merged into a clone set $\{s1, s2, s3\}$.

A. Execution Dependency

We introduce a special dependency, **execution dependency** (in short, ED), to Program Dependency Graphs (PDGs). ED is the same as control flow of control flow graphs. That is, there is an ED between two nodes if the program element represented by the node may be executed just after the program element represented by the other node. The reason why we introduced ED to PDGs is that PDG-based detection is not suitable to detect contiguous code clones [2]. The fact is due to two consecutive statements in the source code often have neither data dependency nor control one. On the other hand, string-based and token-based techniques are suitable to detect contiguous code clones because their detections are based on textural representation of the source code. To

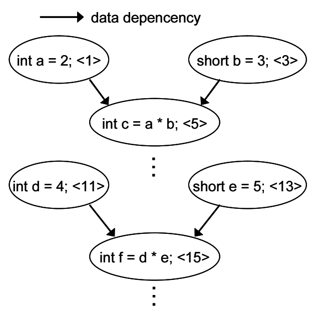
```
++  1: int a = 1;
          :
++  3: int b = a + 5;
          :
++  5: int c = a * 10;
          :
          :
-- 11: int d = 3;
          :
-- 13: int e = d + 5;
          :
-- 15: int f = d * 10;
          :
```
(a) Source code



(b) Program Dependency Graph

Fig. 1. Example of isomorphic graphs that cannot be identified by backward slice

```
++  1: int a = 2;
          :
++  3: short b = 3;
          :
++  5: int c = a * b;
          :
          :
-- 11: int d = 4;
          :
-- 13: short e = 5;
          :
-- 15: int f = d + e;
          :
```
(a) Source code



(b) Program Dependency Graph

Fig. 2. Example of isomorphic graphs that cannot be identified by forward slice

overcome the weakness of PDG-based detection, we use ED. ED makes it possible to detect consecutive statements as code clones even if they have neither data dependency nor control one.

B. Backward and Forward Slices are used

We use both forward slice and backward one meanwhile previous approaches either of them [3], [4]. That is because some kinds of isomorphic subgraphs identified by forward slice cannot be identified by backward slice and vice versa. Figures 1 and 2 are simple examples of such situations. In the both figures, there are three program elements in each code clone. In Figure 1, a variable is referenced twice in each code clone, so that there are two data dependencies, for example, "`<1>` → `<3>`" and "`<1>` → `<5>`" are in code clone`<1,3,5>`. In this instance, backward slice cannot detect the three elements code clones meanwhile forward slices from statements`<1>` and `<11>` can detect them.

III. Implementation

We developed a software tool, Scorpio [1]. The tool implements multi-threads processing to effectively use the resource of multi-cores CPU. The tool has many options to specify what kinds of duplicate code are detected as code clones. For example, minimum size of detected code clones can be configured. The size is specified as the number of node consisting of a code clone. Seven should be an appropriate value from our experience. Also, Parameterization can be variously configured. For example, we can configure how used variables, invoked methods, and literals are parameterized respectively. The parameterization has three level: in level 0, the tokens are used as they are; in level 1, the tokens are replaced with their type names; in level 2, all the tokens are replaced with the same special token.

IV. Conclusion

In this paper, we proposed two techniques for enhancing the quality of code clone detection using program dependency graph. Besides, we are going to use interprocedural PDGs for detecting code clones. Using interprocedural PDGs is more time-consuming; however more interesting and more beneficial code clones will be detected because functionalities straddling two or more methods are identical will be detected as a single clone set.

Acknowledgement

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology. It has been performed under Grant-in-Aid for Scientific Research (C)(20500033) supported by the Japan Society for the Promotion of Science.

References

[1] Scorpio. http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e/.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct 2007.

[3] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 155–169, Jan 2000.

[4] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 301–309, Oct 2001.