Retrieving Similar Code Fragments based on Identifier Similarity for Defect Detection

Norihiro Yoshida, Takashi Ishio, Makoto Matsushita, Katsuro Inoue
Graduate School of Information Science and Technorogy, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
{n-yosida, ishio, matusita, inoue}@ist.osaka-u.ac.jp

ABSTRACT

Similar source code fragments, known as code clones or duplicated code, may involve similar defects caused by the same mistake. However, code clone detection tools cannot detect certain code fragments (e.g. modified after copy-and-pasted). To support developers who would like to detect such defects, we propose a method to retrieve similar code fragments in source code based on the similarity of identifiers between a query and a target code fragment. We present two case studies of similar defects in open source systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids and diagnostics*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Similar code is generally considered as one of factors that make software maintenance more difficult[1, 2, 4]. If developers modify one of similar code fragments, they have to determine whether or not apply the same modifications to the others. Similar code is also called *code clone* or *duplicated code*. Developers often introduce similar code by because of various reasons (e.g. "copy-and-paste")[2, 5]. Especially, large-scale software system, such as Linux, JDK(Java Development Kit), often involves large amount of similar code[4].

Similar code fragments sometimes involve similar defects caused by the same mistake[6, 7]. Therefore, if one of similar code fragments has a defect, it is necessary to inspect the others. Figure 1 is an example of such code fragments in Linux 2.6.6. Those code fragments involve similar defects caused by accessing incorrect pointer (i.e. &prom_phys_total). Because type cast operations (e.g. (char *)) are inserted into

```
(linux-2.6.6/arch/sparc64/prom/memory.c)
for(iter=0; iter<num_regs; iter++) {
 prom_prom_taken[iter].start_adr =
   prom_reg_memlist[iter].phys_addr;
 prom_prom_taken[iter].num_bytes =
   prom_reg_memlist[iter].reg_size;
 prom_prom_taken[iter].theres_more =
   &prom_phys_total[iter+1];
   // should be:&prom_prom_taken[iter+1];
}
```

```
(linux-2.6.6/arch/sparc/prom/memory.c)
for(iter=0; iter<num_regs; iter++) {
  prom_prom_taken[iter].start_adr =
    (char *) prom_reg_memlist[iter].phys_addr;
  prom_prom_taken[iter].num_bytes =
    (unsigned long) prom_reg_memlist[iter].reg_size;
  prom_prom_taken[iter].theres_more =
    &prom_phys_total[iter+1];
  // should be:&prom_prom_taken[iter+1];
}
```

Figure 1: Similar defects

the lower fragment, code clone detection tools[1, 2, 4] do not treat those code fragments as a pair of code clones. Hence, even if developers find out that one of those code fragments has a defect and perform code clone detection, they can not detect the other code fragment that has similar defect. We propose a new approach that compares only identifiers to detect such code fragments.

In this paper, we focus on similar defects that are involved in code fragments sharing identifiers with the same name, and we propose a method to retrieve similar code fragments based on identifier similarity.

2. PROPOSED METHOD

As shown in Figure 2, our method accepts a code fragment as a query and retrieves similar code fragments in target source files. The process comprises the three steps as follows.

(1)Lexical Analysis Both the input code fragment and the target source files are translated into token sequences. Then, only identifiers are extracted from each token sequence. Finally, those identifiers are normalized based on several rules (e.g. dividing at underscore, number suffix elimination) and are listed as Identifier Lists.

(2)Comparison We compare the input identifier list with sublists in the target identifier lists. We compute the

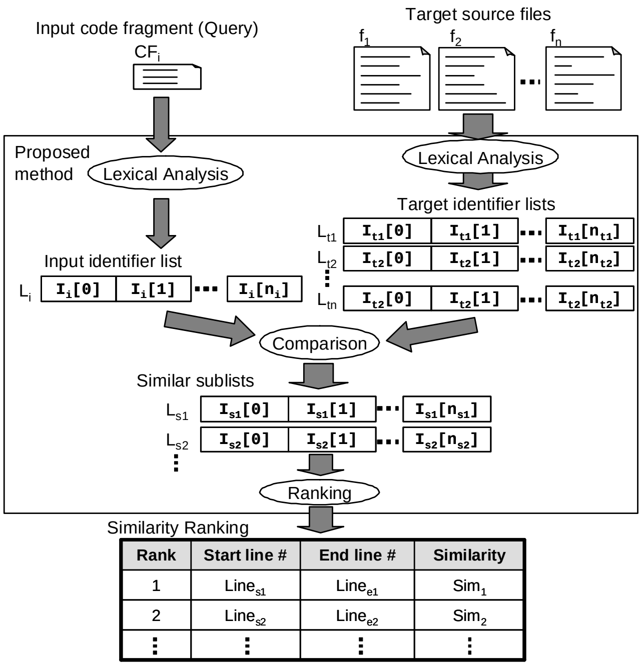

Figure 2: The overview of proposed method

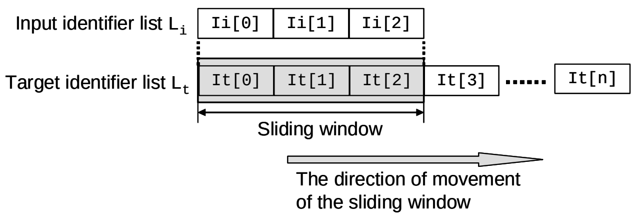
Input identifier list L_i
Ii[0] Ii[1] Ii[2]
Target identifier list L_t
It[0] It[1] It[2] It[3] It[n]
Sliding window
The direction of movement of the sliding window

Figure 3: Sliding window

similarity for each sublist and extract Similar Sublists. The detail of this setep is described later.

(3)Ranking Similar sublists are ranked according to similarity between them to the input identifier list. We call the ranking Similarity Ranking.

Figure 3 shows the comparison between an input identifier list and a target identifier list. We call the scope of comparison Sliding Window, and it moves through the target identifier list. To reduce computational cost, we fix the length of the sliding window to the length of the input identifier list. Hence, the length of a similar sublist is fixed to the length of the input identifier list.

The definition of the similarity in our method is shown in Equation 1. Let S_i be a set of elements in input identifier list L_i, S_w be a set of elements in the sliding window. Then we define the similarity in our method as $Similarity(S_i, S_w)$:

$$Similarity(S_i, S_w) = \frac{2 * |S_i \cap S_w|}{|S_i| + |S_w|} \qquad (1)$$

In the ranking step, since similar sublists are ranked using the similarity score described above, the code fragments which share more identifiers with the input code fragment are ranked higher in the result. Since the ranking involves a huge number of code fragments, developers may investigate similar code fragments according to their resource.

3. CASE STUDY

We performed two case studies of software systems, Linux 2.6.6 and Canna 3.6[3].

The arch directory of Linux 2.6.6 has two similar defects (see Figure 1). We used the two code fragments as queries and then retrieved similar code fragments to each of them in the arch directory. As a result, in both of those queries, the two code fragments in Figure 1 are detected as the top two code fragments in the similarity ranking. If one of those code fragments is given, we can detect the other one having similar defect.

The server directory of Canna 3.6 has 19 buffer overflow errors. Like the case of Linux, we used the 19 code fragments as queries and then retrieved similar code fragments in the server directory. As a result, in all of those queries, 18 or 19 queried code fragments are ranked in the top 30. If one of those code fragments is given, we can detect the almost the other code fragments having similar defect.

4. CONCLUSION

In this paper, we proposed a method to retrieve similar code fragments based on identifier similarity. In the case studies, by providing a code fragment having a defect, we could detect most of similar defects. We need further case studies on other software systems having similar defects.

Acknowledgments

This reseach was supported by JSPS, Grant-in-Aid for Scientific Research (A) (No.17200001) and Grant-in-Aid for JSPS Fellows (No.20-1964).

5. REFERENCES

[1] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Trans. Softw. Eng.*, 33(9):608–621, 2007.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM '98*, pages 368–377, 1998.

[3] Canna. `http://canna.sourceforge.jp`.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[5] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proc. of ISESE 2004*, pages 83–92, 2004.

[6] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. of ICSM '97*, pages 314–321, 1997.

[7] A. Zeller. *Why Programs Fail.* Morgan Kaufmann Pub., 2005.