A Criterion for Filtering Code Clone Related Bugs

Yasuhiro Hayase† Yii Yong Lee Katsuro Inoue†
Graduate School of Information Science and Technology, Osaka University
†{y-hayase,inoue}@ist.osaka-u.ac.jp

ABSTRACT

Software reviews are time-consuming task especially for large software systems. To reduce the efforts required, Li *et al.* developed CP-Miner, a code clone detection tool that detects identifier naming inconsistencies between code clones as bug candidates. However, reviewers using CP-Miner still have to assess many inconsistencies, since the tool also reports many false-positive candidates. To reduce the false-positive candidates, we propose a criterion for filtering the candidates. In our experiments, filtering with the proposed criterion removed 30% of the false-positive candidates and no true-positive candidates. This result shows that the proposed criterion helps the review task by effectively reducing the number of bug candidates.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; D.2.8 [**Software Engineering**]: Metrics

General Terms

Experimentation

Keywords

Code Clone, Defect Mining

1. INTRODUCTION

Large software systems tend to have a significant amount of similar code fragments. These code fragments are called code clones. Often the code clones are introduced through copy-and-paste actions for the purpose of code reuse, and the pasted code usually will go through some modifications.

Bellon *et al.* classified clones into three types according to the modifications of the clones [1]: type 1 clones are exact copies, type 2 clone are syntactically identical copies except for identifier names, and type 3 clones are further modified copies.

However, when the modifications are done manually, there is the possibility that bugs are introduced by inconsistent modifications.

DEFECTS'08, July 20, 2008, Seattle, Washington, USA.

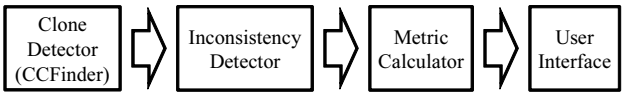

Figure 1: Overview of the inconsistency detection tool

Li *et al.* developed CP-Miner, a code clone detection tool that can also detect such bugs [3]. CP-Miner operates by analyzing the type 2 clones in a software system, and reporting code clone with inconsistent identifier re-naming as possible bug.

However, with CP-Miner, the ratio of false-positive in the reported bug candidate is large; consequently, for large-scale systems, separating true bugs from false ones is a demanding task. Consequently, filtering the inconsistencies may reduce the effort required to perform such separation.

In this paper, we propose a criterion for filtering the name inconsistencies in code clones. Moreover, we build a tool for finding naming inconsistencies that uses CCFinder [2] and implement the proposed criterion. The effectiveness of the criterion is evaluated through an experiment.

The rest of this paper is structured as follows. Section 2 presents the implemented tool, and describes the details of the filtering criterion. Section 3 illustrates the experiment performed for evaluating the filtering criterion. Finally, section 4 concludes this paper with some final remarks and future works.

2. INCONSISTENCY DETECTION TOOL

This section describes the inconsistency detection tool that we developed, and the criterion used for filtering inconsistencies.

2.1 Overview of the Tool

Figure 1 shows the overview of the tool.

In the first step, CCFinder [2] detects the *clone pairs* (pairs of code clones) in a software system.

Then, as described in [3], Inconsistency Detector finds one-to-many mappings of identifiers for each clone pair. For each clone pair, mappings are computed based on the assumption that each code clone is copied from the other. Table 1 shows an example of such identifier mapping; in this table, it is assumed that *clone 2* has been copied from *clone 1*. The first column lists the identifiers appearing in *clone 1*. The second column lists the identifiers appearing in *clone 2* and at the same position of the corresponding identifier in the left column. The third column shows how many times the identifier in the second column appears in the code clone. The remaining columns will be described later. In this example, `p` and `x` in *clone 1* are considered inconsistently renamed because

there is no univocal mapping to identifiers in *clone 2*.

Next, Metric Calculator calculates the metric *UnchangedRatio* (as described in [3]) and the metric *Conflict*, whose details will be described in the next section. The last two columns of table 1 show the values of *UnchangedRatio* and *Conflict* respectively for the considered example.

Finally, the inconsistencies are displayed with a graphical tool. The tool allows filtering the inconsistencies easily based on values of *UnchangedRatio* or *Conflict* or both.

2.2 Filtering Criterion

CP-Miner filters the inconsistencies using the *UnchangedRatio*. With regards to the example of table 1, CP-Miner guesses that the multiple renaming of `x` is more likely to be a bug than the multiple renaming of `p` because the lower *UnchangedRatio* value implies unintentionally change. However, from another perspective, `x` is considered to be intentionally renamed because `x` is renamed to two identifiers: `y` and `z`.

To distinguish these inconsistencies, we propose a criterion *Conflict*, a boolean measure for an inconsistency. For an identifier, *Conflict* is true if the identifier is renamed to two or more identifiers that are not the original identifier, otherwise it is false. In table 1, the values of *Conflict* for `p` and `x` are *false* and *true* respectively.

3. EVALUATION

To evaluate the effectiveness of *Conflict*, we applied the tool to the `arch` module of the Linux kernel version 2.6.6, which is also used in [3]. According to Li *et al.*, the inconsistencies whose *UnchangedRatio* value is equal to or less than 0.4 are used for evaluation. The inconsistencies are reviewed in order of increasing *UnchangedRatio* value.

Figure 2 shows the evaluation result. The horizontal axis shows the number of reviewed inconsistencies; the vertical axis show the accumulated number of bugs found. Using *Conflict* to filter the results removed 36 of the 127 reported inconsistencies, and the removed inconsistencies contained no bugs. Without the filtering, the last bug was the 62nd inconsistency. With the filtering, it was the 42nd inconsistency.

This result shows how filtering using *Conflict* effectively removes false-positive inconsistencies.

4. CONCLUSION REMARKS AND FUTURE WORKS

This paper presents a criterion for filtering renaming inconsistencies in code clones. The criterion detects intentionally renamed inconsistencies using mapping of the identifier. We also implemented the filter using the criterion on a bug detection tool. The evaluation experiment was shown that the filter effectively removes false candidates.

Table 1: Example of identifier mapping

Clone 1	Clone 2	cnt	*UnchangedRatio*	*Conflict*
a	a	2		
b	c	2		
p	p	1	0.25	**false**
	q	3		
x	x	1	0.2	**true**
	y	2		
	z	2		

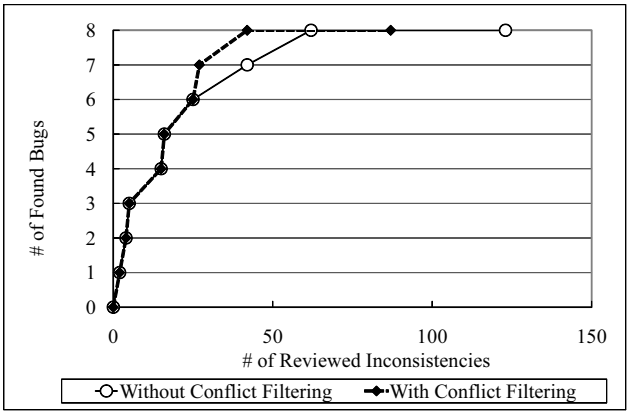

Figure 2: Result of *Conflict* Filtering

In future, we will conduct more experiments on production quality software and perform a comprehensive analysis on the results. Also, we are considering to improve our tool in order to detect more bugs caused by *type 3* clones.

Acknowledgment

This work was supported by Japan Society for the Promotion of Science under Grant-in-Aid for Scientific Research (A) (17200001). And the 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.

5. REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[2] 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.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.