ISSN 2395-1621

To Develop New Techniques and Algorithm for Code Clone Detection

[#1]Prajakta R. Ashtaputre, [#2]Prof. C. S. Kulkarni

[1]prajackta.ashtaputre@gmail.com
[2]ckulkarni47@gmail.com

[#1]Department of Computer Engineering, JSPM Narhe Technical Campus,Narhe
RajarshiShahu School of Engineering & Research
Pune ,Maharashtra, India

[#2]Prof. Department of Computer Engineering, JSPM Narhe Technical Campus,Narhe
RajarshiShahu School of Engineering & Research
Pune ,Maharashtra, India

ABSTRACT

In the case of maintenance and evolution of source code, many studies have shown that the duplicated code or code clone in software are potentially harmful. Though this is the serious problem in software, through refactoring, there is little bit support for eliminating software clones. A very challenging problem is unification and merging the duplicated code, especially after initial introduction to the software clone they are going through the several modifications in them. This paper presents an approach in which a pair of clone is automatically assessed and without changing the program behavior that clone pair is re-factored safely. The differences present in the clones are examined by this approach and those are safely parameterized without occurrence of any side effect. One of the benefit of this approach is that the negligible computational cost. Finally, a large-scale empirical study has been performed on over a million clone pairs detected and this detection is done by four different clone detection tools. This has been conducted in nine open source projects for investing how re-factorability is affected by different clone properties and tool configuration options.

ARTICLE INFO

Article History

Received:21th December 2015

Received in revised form :

22nd December 2015

Accepted: 23rd December, 2015

Published online :

24th December 2015

*Keywords:***Code duplication, Software clone management, Clone refactoring, Re-factorability assessment, Empirical study**.

I. INTRODUCTION

In software systems, it has been recognized that the code duplication is serious problem. Code duplication is having bad effect on software system maintenance and evolution[1]-[2]. In last few years different research communities have developed several techniques which were able to detect and analyze the duplicated code[3]. And now more recent research is focused on clone management activities[4], which includes tracing clones in the history of a project, analyzing the consistency of modifications to clones, updating incrementally clone groups as the project evolves, and prioritizing the refactoring of clones. In addition to above development work, the effect of duplicated code on maintenance effort and cost, error-proneness due to inconsistent updates, software defects, change-proneness, and change propagation have been investigated empirically by several researchers. There is a lack of tools which can automatically analyze software clones to determine whether they can be safely re-factored without changing the behavior of program. One of the important but missing features from clone management is re-factorability analysis. When the developers are interested in finding refactoring opportunities for duplicated code it could be used to filter clones that can be directly re-factored. This is the way by which maintainers can focus on parts of the code that can immediately benefit from refactoring, and thus causes improvement in maintainability. This paper presents an approach that takes two clone fragments as input which are detected from any tool and it applies following three steps to

determine whether they can be re-factored without any side effects or not.

Step 1: in this step, this approach finds code fragments with identical nesting structures within those input clones which serve as potential refactoring opportunities. If they are sharing a common nesting structure then consider that two code fragments can be unified, and therefore re-factored.

Step 2: in this step, this approach finds a mapping between the statements of the code fragments that maximizes the number of mapped statements and minimizes the number of differences between the mapped statements by exploring the search space of alternative mapping solutions.

Step 3: in the last step, the differences between the mapped statements which were detected in the previous step are examined against a set of preconditions. This is done to determine whether they can be parameterized without changing the program behaviour or not.

I. RELATED WORK

Following are the two core program structures that are used in this approach :

a) **Program Structure Tree.**
b) **Program Dependence Graph.**

a) Program Structure Tree

The Program Structure Tree (PST)[5] was introduced by Johnson et al. as a hierarchical representation of program structure and this structure is based on single-entry single-exit (SESE) regions of the control flow graph. The nesting relationship of SESE regions and chains of sequentially composed SESE regions are captured by essentially PST.

b) Program Dependence Graph

The Program Dependence Graph (PDG)[6] is a directed graph which consists of multiple edge types. In PDG the nodes denotes the statements of a function or method, and the edges denotes control and data flow dependencies between statements. In this approach PDG representation is used in two ways. In first way, the composite variables are introduces which represents the state of objects which are referred in body of method and it also creates data dependencies for these variables. In second way, two more types of edges are added in the PDG, which are helpful in the examination of preconditions. These two types of edges are: anti-dependencies and output dependencies.

II. PROBLEM STATEMENT

Though duplication code having large importance in software systems. Many research studies have proven that in the maintenance and evolution of source code, clones can be potentially harmful. Though this is the serious problem in software, through refactoring, there is little bit support for eliminating software clones.

III. MOTIVATION

Though this is the serious problem in software, through refactoring, there is little bit support for eliminating software clones. A very challenging problem is unification and merging the duplicated code, especially after initial introduction to the software clone they are going through the several modifications in them. Here is an approach which automatically assesses whether a clone pair can be safely refactored or not and that is also without changing the behaviour of the program.

IV. OBJECTIVE

Here is the first objective which is nothing but to overcome precondition violations by making changes facilitating the successful unification of the clones and developers could be assisted with a more thorough and advanced automated guidance driven by the detected precondition violations. Second is, in order to find tool configurations that maximize the refactor ability of the detected clones apply search-based techniques.

V. PROPOSED SYSTEM

Two different forms of input are processed by this approach, and those are :
1) Two code fragments are declared as clones by clone detection tool within the body of the same method, or different methods.
2) Two method declarations considered to be duplicated, or it may contain duplicate code fragments somewhere inside their bodies.
Here are the three major steps for assessing the refactorability in this approach :

1) **Nesting Structure Matching [1]**:
In this step, nesting structure of the input clone fragments is analyzed which is useful in finding maximal isomorphic subtrees. It is assumed that two code fragments can be unified only if they are having an identical nesting structure. Each matched subtree pair will be further investigated as a separate clone refactoring opportunity in the next steps.

2) **Statement Mapping [1]**:
The statements extracted from the previous step within the subtree pairs are mapped in a divide-and-conquer fashion. It takes advantage of the identical nesting structure between the isomorphic subtrees, the global mapping problem is divided into smaller subproblems. The corresponding Program Dependence subgraphs are mapped by applying a Maximum Common Subgraph (MCS) algorithm for each sub-problem.

These subsolutions are combined to give the global mapping solution at the end.

3) **Precondition Examination [1]**:
A set of preconditions regarding the preservation of program behavior is examined based on the differences between the mapped statements in the global solution, as well as the statements that may have not been mapped. If no preconditions are violated, then the clone fragments corresponding to the mapped statements can be safely refactored, and thus those are considered to be refactored.

Following Fig shows the steps mentioned above :

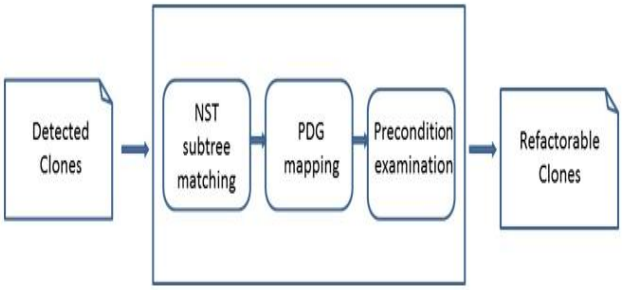

Fig 1. An overview of the proposed refactorability analysis approach

VI. CONCLUSION

This approach introduces an important and missing feature in clone management i.e. refactorability analysis which was unsupported previously.To achieve this goal, here is a technique which first matches the statements of the clones in such a way that minimizes the number of differences between them. After this, these differences are examined against a set of preconditions to determine whether they can be parameterized without changing the program behavior. If no precondition violations are found, provided tool support for the automatic refactoring of the clones.

REFERENCES

1)NikolaosTsantalis, Member, IEEE, DavoodMazinanian, and GiriPanamoottil Krishnan “Assessing the Refactorability of Software Clones” , IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. XX, NO. XX, MONTH 2015

2)G. P. Krishnan and N. Tsantalis, “Unification and refactoring of clones,” in Proceedings of the IEEE Conference on Software Maintenance,Reengineering and Reverse Engineering (CSMR-WCRE),2014 Software Evolution Week, 2014, pp. 104–113.

3)C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitativeapproach,” Science of Computer Programming, vol. 74, no. 7, pp. 470 – 495, 2009.

4)C. Roy, M. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (keynote paper),”in Proceedings of the IEEE Conference on Software Maintenance,Reengineering and Reverse Engineering, 2014 Software Evolution Week, 2014, pp. 18–33.

5) R. Johnson, D. Pearson, and K. Pingali, “The program structure tree: Computing control regions in linear time,” in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 171–185.

6) J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp

