



A Novel Framework to Identify Non-Contiguous Clones

Sonam Gupta¹, Dr. P. C Gupta²

¹Research Scholar, Suresh Gyan Vihar University
JAIPUR (RAJASTHAN), INDIA

²Department of Computer Science and Engineering
University of Kota
KOTA (RAJASTHAN), INDIA

Abstract: Software refactoring is the process of changing the code without changing its behavior and is done to reduce the maintenance effort of the system. Maintenance is a difficult activity which involves adapting the new environment and the difficulty level is increased due to the poor structure of the program. In this paper a new framework is proposed in which the task of removing the clones is done by the developer irrespective of the language in which the project is being made. It also identifies the unique clones which should be removed without changing the behavior of the program thereby the effort at the maintenance side is reduced to a very large extent. This framework has been developed keeping non-contiguous clones in consideration since these types of clones are difficult to extract specially in cases where the flow of program switches from one module to another module.

Keywords: Non-contiguous clones, corrective clone management, dependence-based graph, refactoring code.

I. INTRODUCTION

Developing a software goes through various phases and when each phase is executed properly only then a proper and efficient software is made. It has been proved that the most crucial phase in software development is the maintenance[15]; this is because of the poor structure of the code done [1]. Since the programmers are in a practice of cut-copy-paste they take the code from one place and use it in their structure with some modifications. But the problem with such habit is that those changes are not mentioned anywhere and if that same code needs to be changed at one place then the same changes are not incorporated at every place resulting in wastage of effort, time and cost during maintenance. There are various tools available through which such types of codes can be identified but for that the programmers need to be trained so that they can understand the behavior of the clones [2]. They can identify clones and can use the code without any redundancy and thereby reducing the maintenance effort but this technique requires a lot of time as well as the cost. There are various types of clones, majorly classifies as type-1, type-2 and type-3 clones. This paper proposed the framework specially for type-3 clones(non-contiguous clones) which will cover both type-1 as well as type-2 clones. There are three types of clone management techniques [3]:

- *preventive* clone management or preventive control include activities to avoid new clones. This is done at the developer site itself and for this the developer needs to be trained about how the code can be prevented from various types of clones.
- *compensative* clone management enforces on the activities in which the impact of including the clones and their negativity has to be reduced to a much lower level so that those clones do not harm the system and can be said as a good smell.
- *corrective* clone management covers activities to remove clones from a system. This paper presents the framework on this clone management in which identifying the clone is done by the developer itself after the development of the project but prior to sending it for maintenance so that the maintenance effort can be decreased to a much lower extent.

The rest of the paper is organized as follows:

Section II describes about the related work done so far in reducing the maintenance effort. The proposed framework is given in section III with experiments and results in section IV and finally the conclusion and future work in section V.

II. RELATED WORK

Belady and Lehman [4] showed that the program structure degrades exponentially with regeneration since the code written once for any particular application is reused again and again by some modifications done in it resulting in cloning factors and thereby increasing the maintenance effort. So many models have been developed based on various types like:

- i) Text-based approach [5] in which the exact match of the code is identified but it cannot identify the code in which there is an insertion or deletion of any newline or any new variable etc.

- ii) AST-based approach: In this approach the syntax of the statements are checked and the statements having the same syntax are highlighted [6] but if the statements are not in order like in the code shown in fragment-1 then it will not highlight the same.

```
int a;
for(int i=a;i<5;i++)
{
a=a+1;
cout<<a;
}
(a)
int b;
for(int j=b;j<6;j++)
{
cout<<b;
b=b+2;
}
(b)
```

Fragment-1

AST approach will highlight the declaration of variables as a clone but it will not highlight the statements inside the for loop since those have been reordered.

- iii) Metric-based approach: This approach is very much efficient as compared to the above two approaches but it does not identify non-contiguous clones.
- iv) CFG-based approach: This approach works on clone detection and elimination and it works on both intermediate level as well as on assembly language [7]. It can identify the clones in which there is change in the variable name as well as it identify the clones which include single entry and single exit codes. The researches done by this scheme does not find the non-contiguous clones which is the basis of the research done in this paper.
- v) Dependence based approach: In this approach the code is firstly converted from procedural language to object-oriented code [8]. This uses forward slicing technique to identify the paths in the code that are subject to extraction for the clones and then present the results in the factored form but it does not identify all clones in the codes especially the components which are isomorphic in nature. By this method the slice as a whole is extracted from the code which is known as full slicing approach but it again extracts the meaningless clones.
- vi) Searching for specified patterns: In this method the users themselves search the fragments in the code by the help of selective patterns in the language [9]. The pattern can be identified on the basis of linear searches, successive approximation etc. The fragments are matched with already existing functions in the code and if matched then the fragments are replaced by that corresponding function call.
- vii) Subgraph isomorphism: In this approach the isomorphic graphs are identified but since the approach is heuristic in nature it becomes computationally hard to implement this technique [10].

III. PROPOSED FRAMEWORK

The proposed framework is shown in the figure1, in which the developer first develops the code or the required project then the developer itself refractors the code developed, by refractoring tool the developer will be able to see how much code has been cloned in the project and will be highlighted for the future reference. By this mechanism lots of maintenance work will be reduced and thereby no time and effort will be wasted in identifying the places where the code has been cloned.

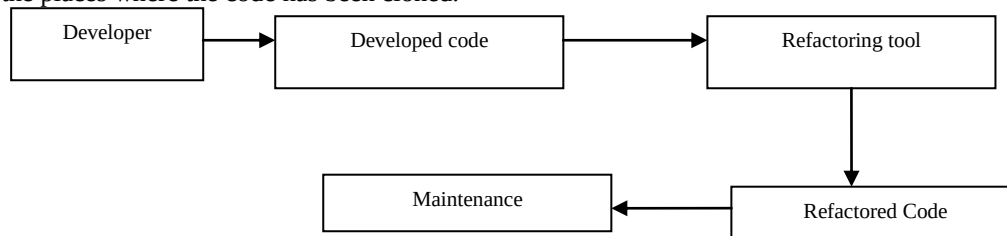


Figure 1

By the help of this framework both semantic as well as syntactic clones will be identified. The approach used will be implemented in a manner to reduce the workload of maintenance[14] as well as of programmer and also reducing the complexities of the approaches discussed earlier. The work of identifying and removing the perfect clones has been left for programmers since they can easily identify that which clone is a true positive and needs to be removed so as to reduce maintenance effort[13]. The framework developed will be able to handle various

types of clones as well the clones having sequence breaking or jumping statements like break, continue, exit etc. It is a combination of CFG and PDG techniques having the advantages of both of them. Since CGF works best at compiler level so the grammar is tested by using CFG and by the help of slicing technique of PDG the clones can be converted into corresponding functions. Moreover researches [16] proved that by using PDG graph matching is comparatively very less expensive as compared to other techniques described earlier.

It comprises two major steps which will be presented in detail in this section.

- (A) Input: the framework is able to handle codes within the same or different method, method declarations, loops, identifiers etc. Through the code given as input to the structure Program Dependence Graph(PDG) will be constructed which will help to identify that whether the code or any other fragment of the code is a clone or not.
- (B) PDG mapping and evaluation: The PDG is constructed from the input and its roots are given as input to the Algorithm1. The algorithm will extract the nodes of the constructed PDG by nextnode (Algorithm 1 Line 3, 4) and then will map the matched nodes by matched_pair (Algorithm1 Line 9). After that by following the principle of bijection of set theory if both the extracted nodes are same then it will be considered as exact clones and then it will be referred to the programmer to indicate about the presence of clones. Now the decision of whether to remove it or not is left to the programmer since earlier studies proved that all clones are not bad smells[11].

Function clone_match(root1,root2)

Data: root1, root2 consists of the roots of the PDG constructed which will be traversed by node_i, node_j respectively

nextnode_i ← root1.node_i.nextnode

nextnode_j ← root2.node_j.nextnode

matchnode ← ∅

for each node_i ∈ nextnode_i do

for each node_j ∈ nextnode_j do

if AST_matched(node_i, node_j) then

matched_pair ← (node_i,node_j)

end

end

end

Algorithm1

The algorithm will work irrespective of the fact that whether the clones belong to a single file or multiple files.

III. EXPERIMENTS AND RESULT

An initial tool is developed in Java on the proposed framework and it is tested on some projects like Apache Ant 1.7.0, Columba 1.4, JEdit 4.2, JFreeChart 1.0.1 and JMetro 2.3.2 by both manually and by the help of the detection tool developed. This has been evaluated on only Java projects since the algorithm is currently developed for just Java codes. The results are shown by the help of graphs shown in figure2 and figure3. Figure2 clearly shows that the proposed approach is able to detect much more number of clones although the number is still less than that of clones detected manually but along with this it does not detect any false positives. When the results are compared with Eclipse and CeDAR [12] which are well known code refactoring tools then it proved that the proposed approach is able to detect more number of clones than these tools which is shown in figure 3.

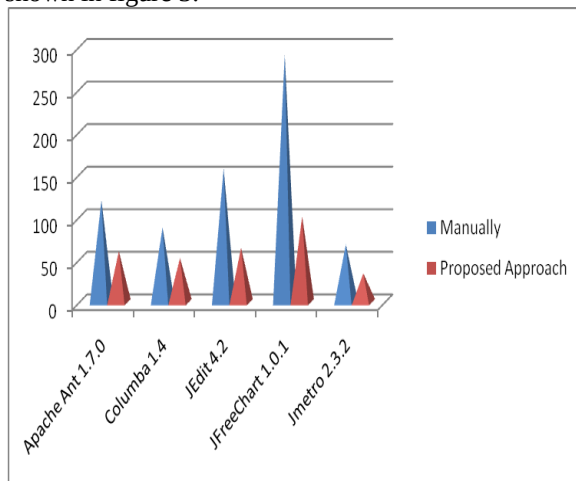


Figure 2

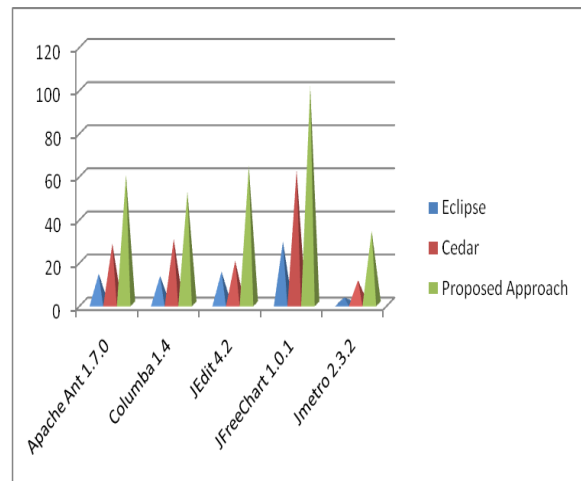


Figure 3

V. CONCLUSION AND FUTURE WORK

The proposed framework is developed by using the features of both CFG and PDG since combination of these two techniques will be able to detect much more number of true positive clones. This will reduce the efforts at maintenance level to a much lower extent since now the developer itself will recognize the occurrence of clones in his code and then can take the appropriate action. The algorithm developed had been tested on five Java projects. In the first phase the results have been compared by calculating the number of clones manually as well as by proposed approach and in the second phase by two refactoring tools Eclipse and Cedar. The results of both the phases show that the structure given in this paper is able to detect much more number of meaningful clones. But the algorithm developed is not able to identify clones in other language except than Java. So the future work will be to extend the algorithm to detect clones of other languages as well as to handle the clones in case of the codes containing switch, break, continue, exit etc. i.e. the commands through which the flow of the code changes

VI. REFERENCES

- [1] A. Lozano, and M. Wermelinger, "Assessing the Effect of Clones on Changeability", Proc. ICSM, pp. 227-236, 2008.
- [2] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones", Proc. WCRE, pp. 170-178, 2007.
- [3] Lozano Rodriguez, Angela, "Assessing the effect of source code characteristics on changeability", PhD thesis, The Open University, 2009.
- [4] L. A. Belady and M. M. Lehman, "A model of large program development", IBM Systems Journal, 15(3):225-252, 1976.
- [5] Kim, M., Bergman, L., Lau, T., Notkin, D.: An ethnographic study of copy and paste programming practices in OOPL. In: International Symposium on Empirical Software Engineering, IEEE Computer Society Press 83-92, (2004).
- [6] I. Baxter, A. Yahin, L. Moura, M. Anna, "Clone detection using abstract syntax trees", Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368 - 377, 1998.
- [7] Di Lucca, G., Di Penta, M., Fasolino, A.: An approach to identify duplicated web pages. In: International Computer Software and Applications Conference. (2002) .
- [8] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code" Proc. Int. Symp. on Static Analysis, pages 40-56, July 2001.
- [9] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection" Automated Software Eng., 3(1-2):77-108, 1996.
- [10] J. McGregor "Backtrack search algorithms and maximal common subgraph problem" Software - Practice and Experience, 12:23-34, 1982.
- [11] Rahman, Foyzur, Christian Bird, and Premkumar Devanbu. "Clones: What is that smell?." Empirical Software Engineering 17.4-5 (2012).
- [12] Giri Panamoottil Krishnan, Nikolaos Tsantalis, "Unification and Refactoring of Clones", CSMR-WCRE 2014, Antwerp, Belgium, IEEE 978-1-4799-3752-3/14 2014.
- [13] Harder, J. , "How Multiple Developers Affect the Evolution of Code Clones," Software Maintenance (ICSM), 2013 29th IEEE International Conference on 22-28 Sept. 2013.
- [14] Giri Panamoottil Krishnan, Nikolaos Tsantalis, "Refactoring Clones: An Optimization Problem", IEEE International Conference on Software Maintenance 2013.
- [15] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An Empirical Study on the Maintenance of Source Code Clones", ESE, 15(1, pp. 1-34,), 2009.
- [16] D. Speicher and A. Bremm, "Clone removal in java programs as a process of stepwise unification," in Proceedings of the 26th Workshop on Logic Programming, 2012.