# Software Maintenance using Duplicate Code Detection and Extraction

Krishna.S.Kadam
*DKTES Textile and Engineering Institute Ichalkaranji, Maharashtra, India*

Amol B.Majgave
*DKTE Society's Textile and Engineering Institute, Ichalkaranji, Maharashtra, India*

**Abstract- Making changes to software are a difficult task. Up to 70% of the effort in the software process goes towards maintenance. This is mainly because programs have poor structure (due to poor initial design or due to repeated ad hoc modifications) which makes them difficult to understand and modify. The focus of this project is duplication in source code, which is a major cause of poor structure in real programs. Using a novel program slicing based [2] approach for detecting duplicated fragments [3] in source code such that the duplicated fragment(s) can be easily extracted into a separate procedure. The key, novel aspect of duplication-detection approach is its ability to detect "difficult" groups of matching fragments, i.e., groups in which matching statements are not in the same order in all fragments, and groups in which non-matching statements intervene between matching statements.**

**Keywords – Program slicing, duplicate fragment.**

## I. INTRODUCTION

Software maintenance involves changing programs to remove bugs, add new features, adapt to changes in the environment, or improve the structure of the program. Maintenance related changes are continually made to programs after they are deployed. However, maintenance is a difficult activity, with the effort needed to make a change often being out of proportion to the magnitude of the change. In fact, a study of 487 companies found that over 70% of the total effort in the software life cycle went to maintenance activities (as opposed to initial development). Maintenance is difficult because programs often have poor structure. One aspect of poor structure is lack of modularity – the part of the program that is pertinent to the upcoming change is not localized and is not easily locatable. Poor structure is sometimes due to poor initial design or lack of necessary features in the programming language, but is often due to repeated, poorly planned changes made to programs. The maintainability of a much modified program can be improved by periodically restructuring the program (improving its structure without affecting its externally observed behavior). Restructuring rolls back the degradation caused by adhoc modifications, and therefore improves future maintainability.

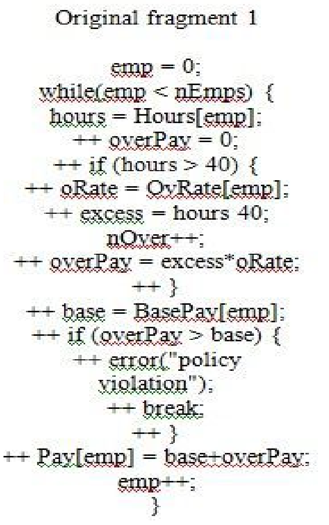
## II. RELEVENT THEORY

Software maintenance involves changing programs to remove bugs, add new features, adapt to changes in the environment, or improve the structure of the program. Maintenance related changes are continually made to programs after they are deployed. However, maintenance is a difficult activity, with the effort needed to make a change often being out of proportion to the magnitude of the change. In fact, a study of 487 companies [5] found that over 70% of the total effort in the software life cycle went to maintenance activities (as opposed to initial development). Maintenance is difficult because programs often have poor structure. One aspect of poor structure is lack of modularity – the part of the program that is pertinent to the upcoming change is not localized and is not easily locatable. Poor structure is sometimes due to poor initial design or lack of necessary features in the programming language, but is often due to repeated, poorly planned changes made to programs. The maintainability of a much modified program can be improved by periodically restructuring the program (improving its structure without affecting its externally observed behavior). Restructuring rolls back the degradation caused by adhoc modifications, and therefore improves future maintainability. However, restructuring itself is a maintenance activity; as such it is time consuming, and it carries the risk of inadvertently changing the program's semantics (behavior). For these reasons, and also because it does not have any immediate benefits, restructuring is rarely done in practice. The mentioned earlier, the study [5] found that only about 5% of the time devoted to maintenance is spent in restructuring activities. This motivates the need for developing restructing tools[1,3], which reduce the effort required for restructuring programs, and give a guarantee that the program's behavior is not changed in any way. The focus of our work is on providing tool support for dealing with one of the common sources of poor program structure, duplication in source code.

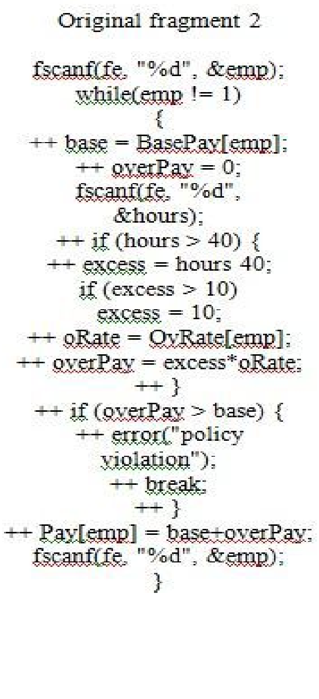
*Present Theories and Practices*

Programs undergoing ongoing development and maintenance often have a lot of duplicated code, a fact that is verified by several reported studies. Their finding was that on the average 7% of the functions in a release were exact clones of other functions. A study [4] found that in the source code of the X Window System (714,479 lines of code), there were 2,487 matching pairs of (contiguous) fragments of length at least 30 lines (the matches were exact, except possibly for one to one variable name substitutions). These matches involved 19% of the entire source code. The same study found that in a production system of over 1.1 million lines 20% of the source code was duplication. Duplication is usually caused by copy and paste activities: a new feature that resembles an existing feature is implemented by copying and pasting code fragments, perhaps followed by some modifications (such modifications result in the creation of "inexact" copies). Programmers also introduce duplication by re implementing functionality that already exists in the system, but of which they are not aware. We have located several clones in real programs that have nearly identical functionality, that have many matching lines, but that nevertheless differ in the way the matching lines are ordered. These clones are evidence that duplication can be introduced without copy and paste. Duplication in source code, irrespective of the reason why it exists, is the cause for several difficulties. It increases the size of the code, which can be a problem in domains (such as embedded systems) where space is a limited commodity. It also makes software maintenance more difficult (which is the aspect we focus on). For example, if an enhancement or bug fix is done on a fragment, it may be necessary to search for clones of the fragment to perform the corresponding modification. Therefore, not only is the effort involved in doing a modification increased, but so also is the risk of doing it incorrectly. By modifying some, but not all copies of a fragment, bugs may be removed incompletely, or worse still, new bugs may be introduced (due to the inconsistency introduced between the clones).

Figure contains an example (with two code fragments) that we use to illustrate clone detection.



```
Original fragment 1

emp = 0;
while(emp < nEmps) {
hours = Hours[emp];
++ overPay = 0;
++ if (hours > 40) {
++ oRate = OvRate[emp];
++ excess = hours 40;
nOver++;
++ overPay = excess*oRate;
++ }
++ base = BasePay[emp];
++ if (overPay > base) {
++ error("policy
violation");
++ break;
++ }
++ Pay[emp] = base+overPay;
emp++;
}
```



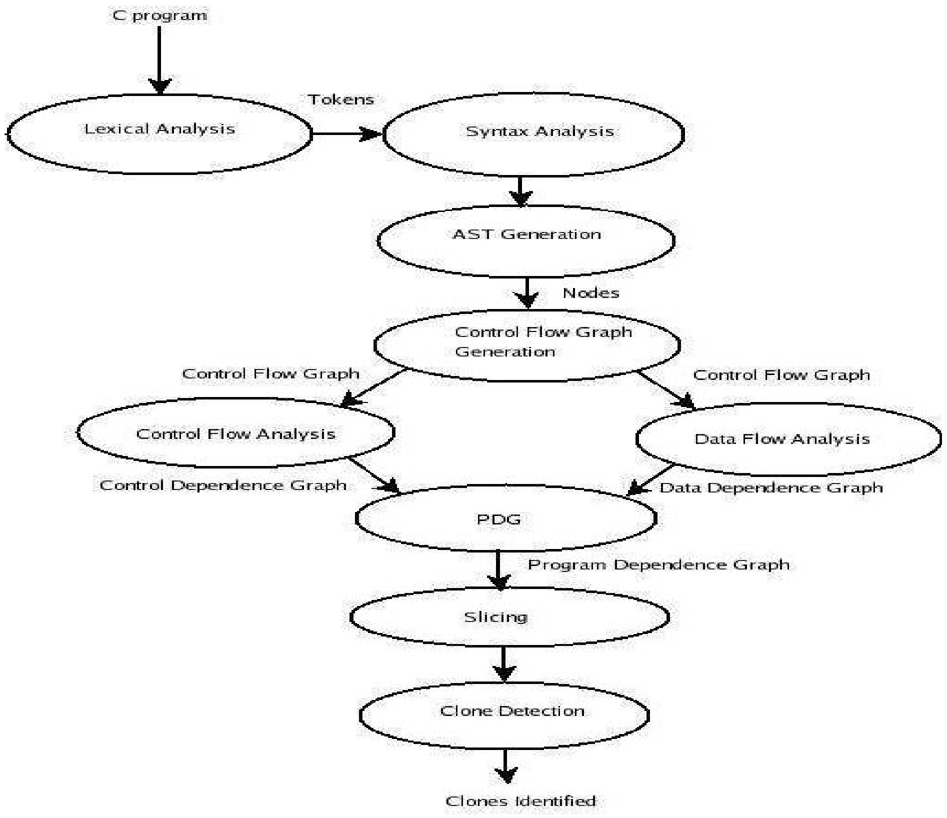
```
Original fragment 2

fscanf(fe, "%d", &emp);
while(emp != 1)
{
++ base = BasePay[emp];
++ overPay = 0;
fscanf(fe, "%d",
&hours);
++ if (hours > 40) {
++ excess = hours 40;
if (excess > 10)
excess = 10;
++ oRate = OvRate[emp];
++ overPay = excess*oRate;
++ }
++ if (overPay > base) {
++ error("policy
violation");
++ break;
++ }
++ Pay[emp] = base+overPay;
fscanf(fe, "%d", &emp);
}
```

The two clones are indicated using "++" signs. The first fragment has a loop that iterates over all employees, while the second fragment has a loop that iterates over employees whose records are present in the file fe. Besides the set of employees that each fragments deals with, the two code fragments have other differences. The first fragment obtains the hours worked from the Hours array, whereas the second fragment obtains this information from the fe file. Also, the first fragment (alone) counts the number of employees earning overtime pay (via the statement "n Over++"), while the second fragment (alone) caps the overtime hours worked at 10. The two clones in this example have characteristics that make them difficult to detect. Each clone is noncontiguous (i.e., has intervening non matching statements). Also, the matching statements in the two clones are out of order: the relative positions of the two statements "if (hours > 40) .." and "base = BasePay[emp]" are different in the two clones, as are the relative positions of the two statements "oRate = OvRate [emp]" and "excess = hours - 40". The clone pair in this example is also difficult to extract. Let us first consider the extract ability of each individual clone in the example (into a procedure of its own). Each clone is difficult to extract for two reasons: the clone is noncontiguous, and it involves an exiting jump (the break statement). A noncontiguous set of statements is difficult to extract because it is not clear which of the several "holes" that remain after the statements are removed should contain the call to the new procedure.

## III. PROPOSED WORK

This paper will focus on detecting the duplicated code for C source program as input. The duplicated code is determined by using program slicing[3]. The graph node and source code statements mapping is maintained and corresponding statements in source code are highlighted. The interesting duplicated clones can be easily extracted into new procedure and replace those clones with a call to extracted procedure.

*Limitations:* Detecting and extracting code only for simple data types.



## IV. ALOGORITHM OVERVIEW

A high-level outline of algorithm is:

1. Partition all nodes in all Program Dependence Graphs[1,3,4] into equivalence classes, such that two codes are in the same class if and only if they match.

2. For each pair of matching nodes (root1, root2), find two matching sub graphs of the Program Dependence Graphs that contain root1 and root2, such that the sub graphs are "rooted" at root1 and root2, using a variation of the slicing operations. The pair of sub graphs found is pair of clones.

The notion of matching nodes is defined (recursively) as follows;

- Two expression match if they have the same syntactic structure, ignoring variable names and literal values; e.g., "b+1" matches "d+2", but does not match "d-2" or "2+d". Array references match other array references if and only if the subscript expressions match.
- Two function calls within expressions (or two procedure-call nodes) match if and only if both are calls to same function, and corresponding actual parameters match (as expressions); e.g., "f(a+1,b())" matches "f(c+2,b())", but does not match "f(c+2,d())" or "f(c-2,b())".
- Two predicates match if and only if their expressions match, and both are of the same kind (while, do-while, if).
- Two assignments match if and only if the left hand sides, as well as the right hand sides match (as expressions).
- Two jumps match if and only if they are of the same kind (return, goto, break, continue). For returns, their expressions must match, too.

*ALGORITHM*

*1]. Procedure FindAllClonePairs.*

```
1.For each PDG p in the program create p.list, the list of all nodes in p sorted in the reverse of the order in which nodes are visited in a depth first traversal of p starting from its entry node.
2: Partition the set of all nodes in the program (i.e., in all PDGs) into equivalence classes, such that matching nodes are in the same class. Represent each class as a list, and sort the list such that the relative ordering of nodes from a single PDG p within the list is the same as their ordering in p.list.
3: Initialize globalHistory to empty.
4: for all PDGs p in the program do
5: for all nodes root1 in p.list do
6: for all nodes root2 in root1's equivalence class (a list), not including root1 and not including nodes following root1 in the list do
7: if (root1, root2) is not present in globalHistory then
8: Call FindAClonePair (root1, root2).
9: end if
10: end for
11: end for
12: end for
```

*1.1] Procedure FindAClonePair (root1, root2).*

```
1: Initialize curr and worklist to empty.
2: Place the pair of edges (root1, root2) in curr. { That is, map these two edges to each other and add them to the current clone pair.} Place (root1, root2) in worklist, and in globalHistory.
3: repeat
4: Call GrowCurrentClonePair.
5: until worklist is empty.
6: Write curr (the current clone pair) to output
```

2] *Procedure GrowCurrentClonePair.*

```
1: Remove a node pair (node1, node2) from worklist. {Map flow dependence parents of node1 to flow dependence parents of node2, as follows.}
2: for all flow dependence parents p1 of node1 in the PDG that are not an endpoint of any edge in curr do
3: if there exists a flow dependence parent p2 of node2 such that:
        • p2 is in the same equivalence class as p1, and p2 is not an endpoint of any edge in curr, and
        • the two flow dependence edges p1 -> node1 and p2-> node2 are either both loop carried, or are both loop independent, and
        • the predicates of the loops crossed by the flow dependence edge p1 ->node1 are in the same equivalence class as the corresponding predicates of the loops crossed by the flow dependence edge p2-> node2  then
4: Place ( p1-> node1, p2-> node2) in curr. { That is, map the two edges to each other and add them to the current clone pair.} Place (p1, p2) in worklist, and in globalHistory.
5: end if
6: end for
{Map control dependence parents of node1 to control dependence parents of node2, as follows.}
7: for all control dependence parents p1 of node1 in the PDG that are not an endpoint of any edge in curr do
8: if there exists a control dependence parent p2 of node2 such that:
        • p2 is in the same equivalence class as p1, and p2 is not an endpoint of any edge in curr, and
        • the two control dependence edges p1 -> node1 and p2-> node2 have the same label (true or false) then
9: Place (p1-> node1, p2-> node2) in curr. Place (p1, p2) in worklist, and in global History.
10: end if
11: end for
12: if node1 and node2 are predicates then
13: {Map control dependence children of node1 to control dependence children of node2, as follows.}
14: for all control dependence children c1 of node1 in the PDG that are not an endpoint
of any edge in curr do
15: if there exists a control dependence child c2 of node2 such that:
        • c2 is in the same equivalence class as c1, and c2 is not an endpoint of any edge in curr, and
        • the two control dependence edges node1 -> c1 andnode2->c2 have the same label (true or false).
then
16: Place (node1-> c1, node2-> c2) in curr. Place (c1, c2) in worklist, and in globalHistory.
17: end if
18: end for
19: end if
```

3] *Clone-group extraction algorithm:*

The input to the algorithm is a group of clones, and a mapping that specifies how the nodes in one clone match the nodes in the other clones. The output from the algorithm is a transformed program such that:

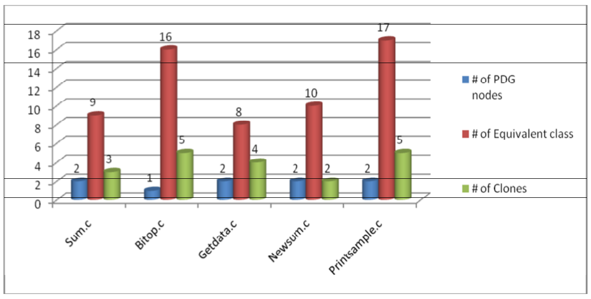
- Each clone is contained in a hammock (which is suitable for extraction into a separate procedure), and
- Matching statements are in the same order in all clones.

## V. EXPERIMENTAL RESULTS

*Running clone detection and extraction:*

| Program | # of lines of source | # of PDG nodes | # of Equi. class | # of Clones | Start Time and End Time Difference(In Milliseconds) |
|---|---|---|---|---|---|
| Sum.c | 38 | 02 | 09 | 03 | 89 ms |
| Bitop.c | 47 | 01 | 16 | 05 | 85 ms |
| Getdata.c | 27 | 02 | 08 | 04 | 88 ms |

| | | | | | |
|---|---|---|---|---|---|
| Newsum.c | 35 | 02 | 10 | 02 | 83 ms |
| Printsample.c | 34 | 02 | 17 | 05 | 84 ms |



## VI .CONCLUSION

Code duplication is a widespread problem in real programs. Duplication is usually cause by copy and paste. a new feature that resembles an existing feature is implemented by copying and pasting code fragments, perhaps followed by some modifications. Duplication degrades program structure. Detecting clones (instances of duplicated code) and eliminating them via procedure extraction gives several benefits: program size is reduced, maintenance becomes easier (bug fixes and updates done on a fragment do not have to be propagated to its copies), and understandability is improved (only one copy has to be read and understood).

## REFERENCES

[1] B. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE TSE*, 33(9):608-621, 2007.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9):577-591, 2007.

[3] W. Evans and C. Fraser. Clone Detection via Structural Abstraction. In *WCRE*, pp. 150-159, 2007.

[4] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminar 06301*, 24pp., 2006.

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy- Paste and Related Bugs in Large-Scale Software Code. *IEEE TSE*, 32(3):176-192, 2006.

[6] R. Tairas and J. Gray. Phoenix-Based Clone Detection Using Suffix Trees. In *ACM-SE*, pp. 679-684, 2006. S. Lee and I. Jeong. SDD: High performance Code Clone Detection System for Large Scale Source Code. In *OOPSLA*, pp. 140-141, 2005.

[7] F. Calefato, F. Lanubile and T. Mallardo. Function Clone Detection in Web Applications: A Semiautomated Approach. *J. of Web Eng.*, 3(1):3-21, 2004.

[8] F. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques. In *ELISA*, 12pp., 2003.

[9] J.R. Cordy, "Comprehending Reality: Practical Challenges to Software Maintenance Automation", Proc. IWPC 2003, IEEE 11th International Workshop on Program Comprehension, Portland, Oregon, May 2003, pp. 196-206 (Keynote paper).

[10] R. V. Komondoor. Automated Duplicated Code Detection and Procedure Extraction. Ph.D. Thesis at Wisconsin Madison University 2003.

[11] S. Horwitz and R. Komondoor. Using slicing to identify duplication in source code.

[12] S. Horwitz, T. Reps and D. Binkley. Interprocedural Slicing Using Dependence Graphs Compilers Principles, Techniques and Tools – Aho, Ullman, and Sethi.

[13] B. Baker. On finding duplication and near duplication in large software systems. In Proc. IEEE Working Conf. on Reverse Eng., pages 86–95, July 1995.