International Journal of Scientific Research in Computer Science and Engineering
Research Paper
Vol.7, Issue.6, pp.19-24, December (2019)
E-ISSN: 2320-7639
DOI: https://doi.org/10.26438/ijsrcse/v7i6.1924

Detection of Cross Language Clones of C and Java Language using Levenshtein Distance Algorithm

Sanjay B. Ankali[1*], Latha Parthiban[2]

[1]Dept. of Computer Science & Engineering, KLECET Chikodi, India & Research Scholar at VTU-RRC, Belagavi
[2] Dept. of Computer Science, Pondicherry University, India

[*]*Corresponding Author: sanjayankali123@gmail.com, Tel.: 8095638746*

Available online at: www.isroset.org

Received: 01/Dec/2019, Accepted: 19/Dec/2019, Online: 31/Dec/2019

Abstract— : In code engineering the study on software forking presents that 10-15% of the code in large codebase are clones. gcc-8.7%, JDK-29%, Linux-22% There are state of art tools for detecting clones like CCFinderX, EqMiner, Dup, Simjava, Nicad but cannot work with IDE's hence To solve the software maintenance efforts in development process it is important to propose efficient techniques to identify clones(especially type-III and type-IV clones) .we propose dictionary based Approach to detect cross clones of C and Java to provide proper inputs to the developers who engage in software forking or porting activities by detecting and correcting porting and copying errors that arise during porting process for IDE's like NetBeans, Eclipse.

Keywords— type-III, type-IV clones

I. INTRODUCTION

Generally code clones are the result of the copy/paste activity widely used by programmers to reuse existing code to save time. Large software codebase consist of 10-15% of duplicate code [1]. Code Cloning is considered harmful to the software quality [2]. i.e if the code containing error is copied then the same error will be distributed across all the target code fragments. [1]. Thus, it is important to develop approaches for clone detection in software systems. Code clones are divided into four classes [4]:

Type I: This type is commonly referenced as exact clones. Clones fragments of type I are exactly identical code fragments. Variations in comments and white space are tolerated.

Type II: identical fragments from the structural and syntactical point of view and with variations in identifiers, literals, types, layout and comments.

Type III: Copied fragments with some modifications. The modifications consist on adding, changing and removing statements.

Type IV: Two or more code fragments that have the same behavior but implemented differently.

To solve the software maintenance efforts in development process it is important to propose efficient techniques to identify type-III and type-IV clones.

Bee Bee Chua [4] in his work analyzed that Java, Python and C are the most preferred languages for implementing Open Source code like Apache, Mozilla and Ubuntu. To help developers that port application among C, Java & python clone detection is important technique.

The paper is organized as follows: Section 2 provides the related work, section 3 provides the Architecture Design & Algorithm, section 4 provides the results and discussions and section 5 limitations and 6 conclusions.

II. RELATED WORK

Based on the survey of Fang –Hsiang Su et al [5]
Static Approaches
- Textual approaches
- Token-based Techniques
- Tree-based Techniques
- PDG-based Techniques
- Metrics-based Techniques

Dynamic Approaches: Work done based on the dynamic profiling. Some of the techniques are listed below.

I. F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner[6] proposed Simion detection pipeline that works on code chopper, code transformation and filtering. But has limitations

1. Identifies only functionally similar Java-codes
2. Efficiency of input-output generation process is not reliable.
3. Cannot be plugged in to IDE's.

II. Farouq Al-omari, Iman Keivanloo, Chanchal K. Roy, Juergen Rilling [1] Work is mainly based on 3 algorithms namely SimHash, Longest Common Subsequence (LCS), and Levenshtein Distance to detect clone-pairs. Study reveals the quantitative and qualitative performance aspects of clone detection approach. Results show number of reported candidate clone-pairs, as well as the precision and recall (using manual validation) for several open source cross-language systems.

Limitations: Matching algorithm is limited to the information present in boxes.

1. Platform dependent.
2. Cannot be applied on large codebase as length of CIL is more for corresponding C#, VB code.
3. CIL instructions contain noise that needs filtering which imposes lot of processing burden.

III. Yang Yuan and Yao Guo, Boreas [7] proposed token based clone detection techniques that matches based on number of different identifiers present in the code.

Limitations

1. No accurate calculation of false positive rate
2. No results found for large codebase

IV. Bayu Priyambadha, Siti Rochimah [8] proposed method clone detection based on PDG which that identifies similar methods in given large codebase

Limitations

1. Wont detect type-IV clone
2. Static variables may not be detected properly
3. Applying the method for medium and large size may be challenging.

V. Flavius-Mihai Lazar, Ovidiu Banias [9] proposed clone detection based on AST based method. That works on sequence detection and generalization algorithm.

Limitations:

1. Works only on C code clone detection
2. Cannot be scaled on large data sets
3. Cannot be integrated to IDE

VI. Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, Simha Sethumadhavan [5] Technique that detects functional clones in arbitrary programs by identifying and mining their inputs and outputs. The key insight is to use existing workloads to execute programs and then measure functional similarities between programs based on their inputs and outputs, which mitigates the problems in object oriented languages reported by prior work. The technique is implemented in system, HitoshiIO, which is open source and freely available. Experimental results show that HitoshiIO detects more than 800 functional clones across a corpus of 118 projects. In a random sample of the detected clones, HitoshiIO achieves 68+% true positive rates with only 15% false positive rate

Limitations

1. Experiment was applied on small size code base of 118 projects from Google code jam repository.
2. More number of false positives.
3. There are many implementation limitations to be used in HitoshiIO as experimentation shows small numbers of clone detected.
4. Capturing inputs/outputs method requires more refinement.

VII. Chaiyong Ragkhitwetsagul,Measuring [10] Technique uses "Internet-scaled Similar Code Search (ISiCS)" framework is a code search framework that is scalable and resistant to code incompleteness

Limitations

1. No results found on large code base
2. Reliability needs to be tested on frequency of false positive.

VIII. Vaibhav Saini, Hitesh Sajnani, Jaewoo Kim, and Cristina Lopes [11] It is a token-based clone detector that targets the first three clone types, and exploits an index to achieve scalability to large inter-project repositories using a standard workstation. It uses an optimized inverted-index to quickly query the potential clones of a given code block.

Limitations

1. Wont detect near miss clones and type 4 clone
2. Reduced efficiency because of heuristic filtering.
3. No enough results presented to prove efficiency

IX. Chanchal K. Roy [12] Proposed NICAD to detect Near Miss clones by applying Pretty Printing, Code Normalization and code filtering. Runs LCS algorithm to detect similarity among the lines of codes

Limitations

1. Only finds exact clones & near miss clones.
2. Cannot find Type-4 semantic clones.
3. Cannot find cross language clones

III. METHODOLOGY

Fig. 1 shows overall architecture, where C, Java, Python (clone of C) vice-versa are given as input to Porting scanner. Intelligent code comparator algorithm finds copied, forked and porting code to calculate frequency of the porting by using comparisons and frame the consistency and inconsistency blocks.

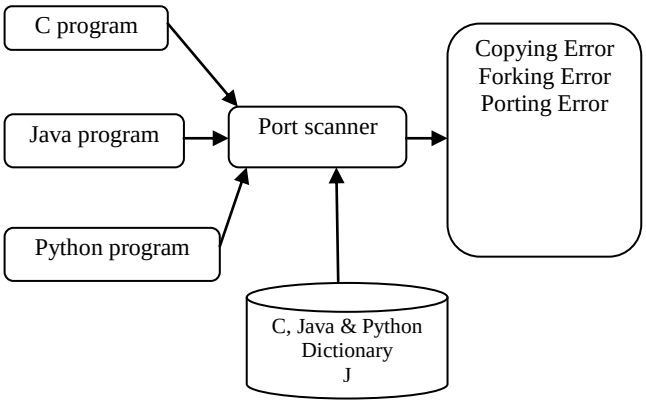

Fig. 1 Overall Architecture Diagram

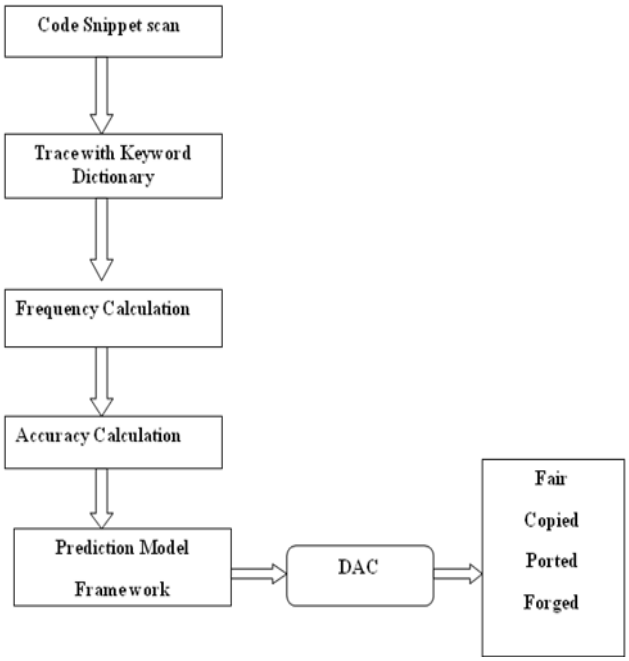

Fig. 2 Flow Diagram of Directed Auto Correction (DAC)

The C code will be converted into Java code using online converter www.mtsystems.com then DAC takes input from prediction model and finds amount of fair, copied, ported and forged code snippet. Then necessary modifications will be done in either of the code to make it clone of cross language. The prediction model creates bar chart to indicate amount of lines that are part of clone. The same result will be displayed graphically to help developers monitor and analyze amount of porting taken place.

Fig. 2 shows steps to find frequency for analysis in second phase. Prediction model Generate the intelligent code comparators with respect to relevant languages.

```
AlgorithmType1(String strS, String strD)
begin
let preProcessCommentS := 0
let preProcessCommentD :=0
let sourceLineDupCommentsOffset[]
let destLineDupCommentsOffset[]

Read source code1 to strS
Read source code2 to strD
let sourceLinesComments:=0
let destLinesComments:=0
sourceLinesComments :=
getCommentedPortion of strS
destLinesComments := getCommentedPortion
of strD
for i:=0 to sourceLinesComment
begin
String tempS := Read
sourceLinesComments(i)
If destLinesComments contains(tempS)
begin
prePocessIndex :=
sourceLinesComments(tempS)
add sourceLineDupCommentsOffset(i)
if prePocessIndex==0
begin
++preProcessCommentD
End if
End if
End for
For i=0 to destLinesComments
begin
String tempS := read destLinesComments(i)
If sourceLinesComments contains(tempS)
begin prePocessIndex :=
destLinesComments(tempS)
add destLineDupCommentsOffset(i)
if prePocessIndex==0
begin
++preProcessCommentS
End if
End if
End for

Draw Bar chart to indicate clone type
number of comments among both codes

End Algorithm

AlgorithmType2(String strS, String strD)
begin
let totalSyntacticSimCOunts := 0
let totalSyntacticSimLines:=0
read code1 in strS
read code2 in strD
sourceLines := getTokensFromString(strS)
destLines := getTokensFromString(strD)
int sSize := sourceLines
int dSize := destLines
```



```
int actSize := 0
if sSize < dSize
actSize := sSize
else if sSize > dSize
actSize = dSize
else if sSize == dSize
actSize = dSize
for i=0 to actsize
begin
begin
String toTestS := sourceLines (i)
String toTestD := destLines(i)
Let actSimCounts:=0
actSimCounts :=
getLineSimilarityType2(toTestS,toTestD)
let cc := actSimCounts.get(1)
let dd := actSimCounts.get(2)
let diff := dd - cc
if cc>0 && dd>0 && diff>=0
begin
++totalSyntacticSimCOunts
totalSyntacticSimLines.add(i)
end if
end for
Draw Bar chart to indicate clone type &
number of similar lines among both
codes
End Algorithm
```

```
AlgorithmType3(String strS, String strD)
begin
Read code1 in strS
Read code2 in strD

Let sourceLines:=0, destLines:=0
sourceLines := getTokensFromString(strS)
destLines := getTokensFromString(strD)
let sourceSize := sourceLines
let destSize := destLines
let actSize := 0
if sourceSize < destSize
actSize := sourceSize
else if sourceSize>destSize
actSize := destSize;
else if sourceSize==destSize
actSize := destSize
String sss := sourceLines(i)
String ddd := destLines(i)
For j=0 to sDictionarysize
begin
String sKeyWord := sDictionary(j)
String dKeyWord := dDictionary(j)
If sss contains(sKeyWord) && ddd
contains(dKeyWord)
begin
Levenshtein l := new Levenshtein()
double value := l.distance(sss, ddd)
if value<0.85
begin
print value;
add copiedIndexes(i);
endif
endfor
endfor
Draw Bar chart to indicate clone type
number of similar lines among both
codes
End Algorithm
```

```
AlgorithmType4(String strS, String strD)
begin
Read code1 in strS
Read code2 in strD

Let sourceLines:=0, destLines:=0
sourceLines := getTokensFromString(strS)
destLines := getTokensFromString(strD)
let sourceSize := sourceLines
let destSize := destLines
let actSize := 0
if sourceSize < destSize
actSize := sourceSize
else if sourceSize>destSize
actSize := destSize;
else if sourceSize==destSize
actSize := destSize
String sss := sourceLines(i)
String ddd := destLines(i)
let index := 0
for i:=0 to actSize
begin
String sSource := sourceLines
String dSource = destLines
If sSourcecontains("for") ||
sSourcecontains("do") ||
sSource.contains("while")
begin
if dSourcecontains("for") ||
dSourcecontains("do") ||
dSource.contains("while")
begin
index = i;
endif
endif
endfor
Draw Bar chart to indicate clone type
number of similar lines among both
codes
End Algorithm
```


IV. RESULT AND DISCUSSION

Experimental Results

Table 1 Amount of copied and forged lines

Sl.No.	Code under test	Amount of forged Code (No.Of lines)	Amount of Normal Code (No. of Line)
01	Clock	20	35
02	Counter	09	10
03	String print	20	14

Table 1 shows the similarity measures of three C and Java codes namely Clock, Counter & String print where tested for similarity

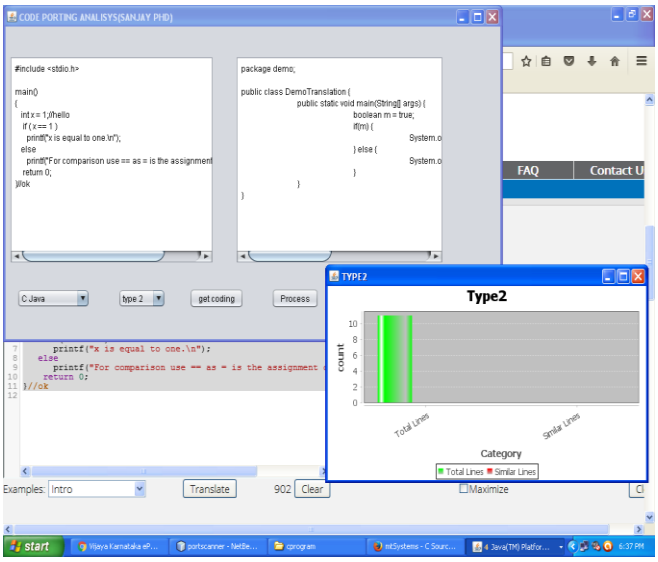

Fig. 3 shows Type-2 clone detection for C to Java code

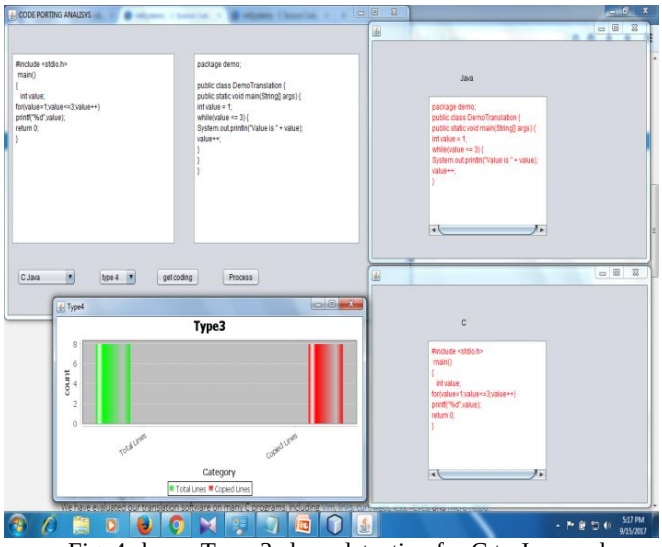

Fig. 4 shows Type-3 clone detection for C to Java code

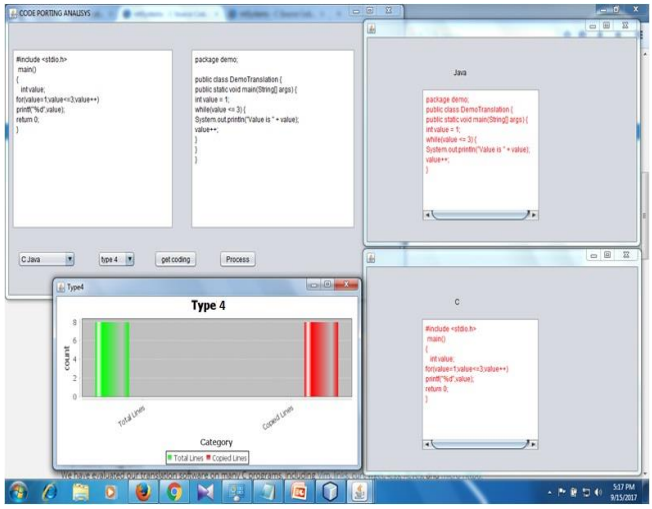

Fig. 5 shows Type-4 clone detection for C to Java code

IV Limitations

i. Experiments are conducted only on small applications such as clock, counter, string print. Proposed method works very well to detect amount of forged and copied code by detecting Type-2, Type-3 & Type-4 clones.

ii. Efficiency has to be identified for large code base of several KLOC.

iii. Same work needs to be extended to detect clones among C, Java and python for online open source hubs like GITHub by taking input as version histories of two different projects.

IV. CONCLUSION

The proposed method detects all 4 types of clones in cross language under common umbrella and presents the results graphically that helps maintenance engineers to develop the porting analysis tools such as REPRTOIRE[3] that answers many questions such as i. What percentage of mainline commits is back ported? ii. What are the characteristics of back ported patches— bug fixes, feature additions, new functionalities, etc.? iii. How different is a back ported patch with respect to its original main-line patch?

iv. How much time does it take to test a back ported patch? These questions could help us to understand the effort of maintaining parallel versions of a project. Studying bug report similarities in a product family.

v. The proposed work helps developers involved in software porting to detect and correct porting and copying errors.

REFERENCES

[1] F. Alomari, I. Keivanloo, C. K. Roy, J. Rilling, "*Detecting Clones across Microsoft .NET Programming Language"s*, 1095-1350/91 4891 IEEE.

[2] M. Abdelkader M. Malki, "*Clone Detection Using Time Series and Dynamic Time Warping Techniques"*,978-1-4673-9669-1/15, 2015 IEEE.

[3] B. Ray, C. Wiley, M. Kim, "*REPERTOIRE: A Cross-System Porting Analysis Tool for Forked Software Project"s*, SIGSOFT'12/FSE-20,November,11–16,2012, Cary, North Carolina, USA. Copyright 2012 ACM 978-1-4503-1614-9/12.

[4] S. Abid, S. Javed, Momna Naseem, Suleman Shahid, H.Abdul Basit,Yoshiki Higo,"*CodeEase: Harnessing Method Clone Structures for Reuse"*, 978-1-5090-6595- 0/17, 2017 IEEE .

[4] B.B.chua, "*Detecting sustainable programming languages through forking open source project for survivability"y*, 978-1-5090-1944-1/15, 2015 IEEE.

[5] F.-Hsiang Su, J.Bell, G.Kaiser, S. Sethumadhavan, "*Identifying Functionally Similar Code in Complex Codebases"*, ICPC 2016, Austin, Texas, 978-1-5090-1428-6/2016 IEEE

[6] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, *"Chal-lenges of the Dynamic Detection of Functionally Similar Code Fragments*," ser. CSMR '12.

[7] Y.Yuan and Y. Guo, "*Boreas: An Accurate and Scalable Token-Based Approac to Code Clone Detection"* ,2013 International Conference on Software Engineering.

[8] B. Priyambadha, S. Rochimah, "*Case Study on Semantic Clone Detection Based On Code Behavior"*, 978-1-4799-7996-7, 2014 IEEE.

[9] F. Mihai Lazar, O. Banias, "*Clone detection algorithm based on the Abstract Syntax Tree approach"*, 9th IEEE International Symposium on Applied Computational Intelligence and Informatics • May 15-17, 2014 • Timişoara, Romania.

[10] C. Ragkhitwetsagul, "*Measuring Code Similarity in Large-scaled Code Corpora"*, 978-1-5090-3806-0/2016 IEEE.

[11]V. Saini, H. Sajnani, J. Kim, and C. Lopes, *"SourcererCC andSourcererCC-I: Tools to Detect Clones in Batch mode and During Software Development,"* ICSE'16 Companion, May 14-22, 2016, Austin, TX,USA c 2016 ACM. ISBN 978-1-4503-4205-6/16/05.

[12]C. K. Roy, "*Detection and Analysis of Near-Miss Software Clones"*, 978-1- 4244-4828 2009 IEEE.

[13] www.mtsystems.com

[14] M.Steinbeck, "*an Arc-Based Approach for Visualization of Code Smell"s*, 978-1-5090-5501-2, 2017 IEEE

[15]Y. Hu, Y.Zhang, J. Li, D. Gu, *"Cross-Architecture Binary Semantics Understanding via Similar Code Comparison*", 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering.

[16]D. Mazinanian, N. Tsantalis, R. Stein, Z. Valenta, "*JDeodorant: Clone Refactoring"*, 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

[17] T. Ishio, Y.e, Kaoru, "*source file search file for clone and own"*, 2017 IEEE, 14th International Conference on Software Repository

[18] K. Raheja,, R. Tekchandani, "*An Efficient Code Clone Detection Model on Java Byte Code Using Hybrid Approach*"

[19]M. Stephan,James R. Cordy, "*Model-Driven Evaluation of Software Architecture Quality Using Model Clone Detection"*, 978-1-5090-4127-5/ 2016 IEEE

AUTHORS PROFILE

Sanjay Ankali is Research scholar at VTU, RRC-Belagavi-590018 and working as Assistant Professor in department of CSE at KLECET, Chikodi, India-591201. His research interest is in the field of Software Engineering.

Dr. Latha Parthiban is working as Assistant Professor in department of Computer Science at Community College, Pondicherry University, India-605008. She received Bachelors of Engineering in Electronics from Madras University in the year 1994. M. E from Anna University in the year 2008 and Ph D from Pondicherry University in the year 2010. Her research interest is in Software Engineering, Big Data Analytics, and Computer Networking.

