INTERNATIONAL JOURNAL OF SOFT COMPUTING
MEDWELL PUBLICATIONS SINCE 2000

Dictionary Based Approach To Detect Cross Language Clones of C and Java Language

[1]Sanjay B. Ankali and [2]Latha Parthiban
[1]*K.L.E. College of Engineering and Technology, Chikodi, Visvesvaraya Technological University,* Belgaum, *Belgavi, India*
[2]*Department of Computer Science, Pondicherry University CC, Pondicherry, India*

Key words: Type-III, type-IV clones, clones, C, Java, porting tools

Corresponding Author:
Sanjay B. Ankali
K.L.E. College of Engineering and Technology, Chikodi, Visvesvaraya Technological University, Belgaum, Belgavi, India

Page No.: 91-96
Volume: 15, Issue 4, 2020
ISSN: 1816-9503
International Journal of Soft Computing

Abstract: Software clones are the result of the copy/paste activity widely used by programmers to reuse existing code to save time. About 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). In this work, 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.

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[3].

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 behaviour 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. Chua[4] in his research 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 and python clone detection is important technique.

The study is organized as follows: related work, architecture design and algorithm, results and discussions, limitations and conclusions.

Literature review: Based on the survey of Su *et al*.[5].

Static approaches:
- Textual approaches
- Token-based techniques
- Tree-based techniques
- PDG-based techniques
- Metrics-based techniques

MATERIALS AND METHODS

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

Deissenboeck *et al*.[6] proposed Simion detection pipeline that works on code chopper, code transformation and filtering. But has limitations:

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

Al-Omari *et al*.[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:

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

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

Limitations:
- No accurate calculation of false positive rate
- No results found for large codebase

Priyambadha and Rochimah[8] proposed method clone detection based on PDG that identifies similar methods in given large codebase.

Limitations:
- Wont detect type-IV clone
- Static variables may not be detected properly
- Applying the method for medium and large size may be challenging

Lazar and Banias[9] proposed clone detection based on AST based method. That works on sequence detection and generalization algorithm.

Limitations:
- Works only on C code clone detection
- Cannot be scaled on large data sets
- Cannot be integrated to IDE

Su *et al*.[5] proposed the 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 >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:
- Experiment was applied on small size code base of 118 projects from Google code jam repository
- More number of false positives.
- There are many implementation limitations to be used in HitoshiIO as experimentation shows small numbers of clone detected
- Capturing inputs/outputs method requires more refinement

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

Limitations:
- No results found on large code base
- Reliability needs to be tested on frequency of false positive

Saini *et al*.[11] proposed 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.

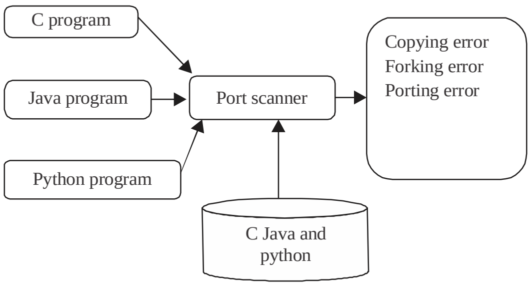

Fig. 1: Overall architecture diagram

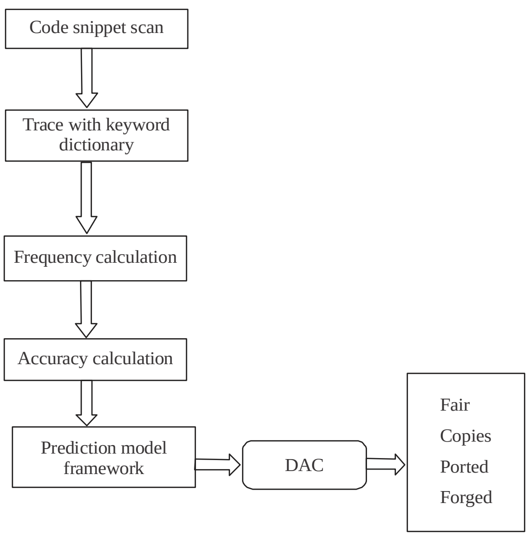

Fig. 2: Flow diagram of Directed Auto Correction (DAC)

Limitations:

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

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:

- Only finds exact clones and near miss clones
- Cannot find type-4 semantic clones
- Cannot find cross language clones

Architecture design and algorithm: Figure 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.

The C code will be converted into Java code using online converter mtsystems 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.

Figure 2 calculate the final frequency for analysis in second phase. Prediction model generate the intelligent code comparators with respect to relevant languages.

Algorithm Type 1 (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
     sourceLines Comments := 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
```

Algorithm Type 2 (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
```

Algorithm Type 3 (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()
   let  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
```

Algorithm Type 4 (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
```

RESULTS AND DISCUSSION

Experimental results: Table 1 shows amount of copied and forged lines. Inconsistency code was identified by matching the professional origin code with the forged code to present amount of copied and normal code.

FIgure 3 shows the Type 2 clone between C and Java. C code was converted by online converter mtsystems to get java code (clone) then the Java code was manipulated to change the identifiers and variables name. Our framework detects exact amount of code that is similar between 2 codes in green color bar chart. Similarly, Type 3 and Type 4 clones are presented.

Limitations: 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 and Type-4 clones.

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

Table 1: Amount of copied and forged lines

Code under test	Amount of forged code (No. of lines)	Amount of normal code (No. of line)
Clock	20	35
Counter	09	10
String print	20	14

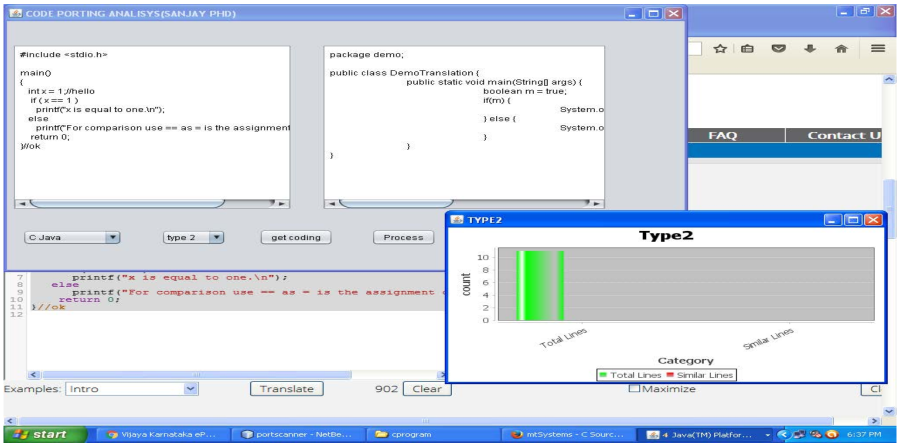

Fig. 3: Type-2 clone detection for C to Java code

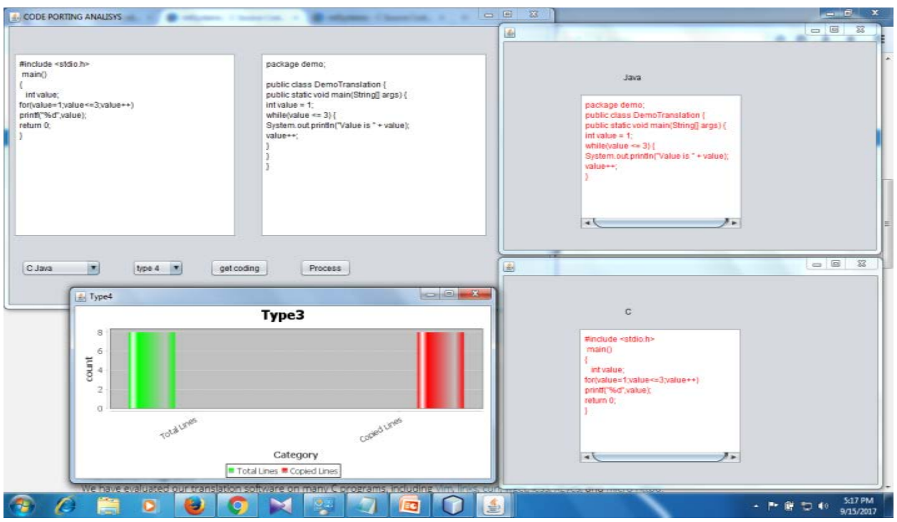

Fig. 4: Type-3 clone detection for C to Java code

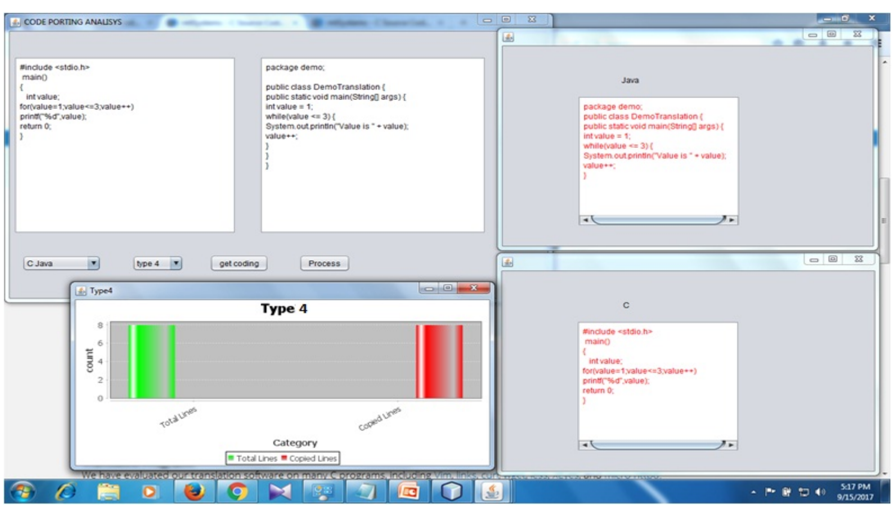

Fig. 5: Type-4 clone detection for C to Java code

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 REPERTOIRE that answers many questions such as: What percentage of mainline commits is back ported?. What are the characteristics of back ported patches-bug fixes, feature additions, new functionalities, etc.?. How different is a back ported patch with respect to its original main-line patch?

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.

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

REFERENCES

01. Al-Omari, F., I. Keivanloo, C.K. Roy and J. Rilling, 2012. Detecting clones across microsoft. net programming languages. Proceedings of the 2012 19th Working Conference on Reverse Engineering (WCRE), October 15-18, 2012, IEEE, Kingston, Ontario, Canada, ISBN: 978-1-4673-4536-1, pp: 405-414.
02. Abdelkader, M. and M. Mimoun, 2015. Clone detection using time series and dynamic time warping techniques. Proceedings of the 2015 3rd World International Conference on Complex Systems (WCCS), November 23-25, 2015, IEEE, Marrakech, Morocco, ISBN:978-1-4673-9669-1, pp: 1-6.
03. Abid, S., S. Javed, M. Naseem, S. Shahid and H.A. Basit *et al*., 2017. Codeease: Harnessing method clone structures for reuse. Proceedings of the 2017 IEEE 11th International Workshop on Software Clones (IWSC), February 21-21, 2017, IEEE, Klagenfurt, Austria, ISBN: 978-1-5090-6595-0, pp: 1-7.
04. Chua, B., 2015. Detecting sustainable programming languages through forking on open source projects for survivability. Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), November 2-5, 2015, IEEE, Gaithersburg, Maryland, USA., ISBN:978-1-5090-1943-4, pp: 120-124.
05. Su, F.H., J. Bell, G. Kaiser and S. Sethumadhavan, 2016. Identifying functionally similar code in complex codebases. Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC), May 16-17, 2016, IEEE, Austin, Texas, USA., ISBN:978-1-5090-1428-6, pp: 1-10.
06. Deissenboeck, F., L. Heinemann, B. Hummel and S. Wagner, 2012. Challenges of the dynamic detection of functionally similar code fragments. Proceedings of the 2012 16th European International Conference on Software Maintenance and Reengineering (CSMR), March 27-30, 2012, IEEE, Szeged, Hungary, ISBN:978-1-4673-0984-4, pp: 299-308.
07. Yuan, Y. and Y. Guo, 2012. Boreas: An accurate and scalable token-based approach to code clone detection. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), September 3-7, 2012, IEEE, Essen, Germany, ISBN:978-1-4503-1204-2, pp: 286-289.
08. Priyambadha, B. and S. Rochimah, 2014. Case study on semantic clone detection based on code behavior. Proceedings of the 2014 International Conference on Data and Software Engineering (ICODSE), November 26-27, 2014, IEEE, Bandung, Indonesia, ISBN:978-1-4799-8175-5, pp: 1-6.
09. Lazar, F.M. and O. Banias, 2014. Clone detection algorithm based on the abstract syntax tree approach. Proceedings of the 2014 IEEE 9th International Symposium on Applied Computational Intelligence and Informatics (SACI), May 15-17, 2014, IEEE, Timisoara, Romania, ISBN:978-1-4799-4694-5, pp: 73-78.
10. Ragkhitwetsagul, C., 2016. Measuring code similarity in large-scaled code corpora. Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), October 2-7, 2016, IEEE, Raleigh, North Carolina, USA., ISBN:978-1-5090-3806-0, pp: 626-630.
11. Saini, V., H. Sajnani, J. Kim and C. Lopes, 2016. SourcererCC and SourcererCC-I: Tools to detect clones in batch mode and during software development. Proceedings of the 38th International Conference on Software Engineering Companion, May 14-22, 2016, ACM, New York, USA., ISBN:978-1-4503-4205-6, pp: 597-600.
12. Roy, C.K., 2009. Detection and analysis of near-miss software clones. Proceedings of the IEEE International Conference on Software Maintenance ICSM, September 20-26, 2009, IEEE, Edmonton, Alberta, Canada, ISBN: 978-1-4244-4828-9, pp: 447-450.