Mining Revision Histories to Detect Cross-Language Clones without Intermediates

Xiao Cheng[1], Zhiming Peng[2], Lingxiao Jiang[2], Hao Zhong[1], Haibo Yu[3], Jianjun Zhao[4]
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[2]School of Information Systems, Singapore Management University, Singapore
[3]School of Software, Shanghai Jiao Tong University, China
[4]Department of Advanced Information Technology, Kyushu University, Japan
{x.cheng, zhonghao, haibo_yu}@sjtu.edu.cn, {zmpeng, lxjiang}@smu.edu.sg, zhao@ait.kyushu-u.ac.jp

ABSTRACT

To attract more users on different platforms, many projects release their versions in multiple programming languages (*e.g.*, Java and C#). They typically have many code snippets that implement similar functionalities, *i.e.*, cross-language clones. Programmers often need to track and modify cross-language clones consistently to maintain similar functionalities across different language implementations. In literature, researchers have proposed approaches to detect cross-language clones, mostly for languages that share a common intermediate language (such as the .NET language family) so that techniques for detecting single-language clones can be applied. As a result, those approaches cannot detect cross-language clones for many projects that are not implemented in a .NET language. To overcome the limitation, in this paper, we propose a novel approach, *CLCMiner*, that detects cross-language clones automatically without the need of an intermediate language. Our approach mines such clones from revision histories, which reflect how programmers maintain cross-language clones in practice. We have implemented a prototype tool for our approach and conducted an evaluation on five open source projects that have versions in Java and C#. The results show that *CLCMiner* achieves high accuracy and point to promising future work.

CCS Concepts

•**Software and its engineering** → *Software libraries and repositories; Software maintenance tools;*

Keywords

cross-language clone, *diff*, revision history

1. INTRODUCTION

Due to various considerations, many projects are implemented in different programming languages. For example, ANTLR [1] releases its versions in Java, C#, JavaScript and Python. As another example, Lucene [2] release its versions in Java and C#. When maintaining such projects, if a code snippet is modified, programmers often copy their modifications to proper locations in other language versions, and conduct further editions, according to the syntactic and semantic requirements of the target programming language. As a result, released versions can have similar code snippet in different programming languages. In literature, Kraft *et al.* [15] call such code snippets as *cross-language code clones*.

Cross-language clones can be inevitable and beneficent for a project [13], even though sometimes code clones may be harmful and could be removed [6]. It also becomes necessary for programmers to locate and maintain cross-language clones. For example, after a developer D1 develops a cross-language project, another developer D2, who is not familiar with the source code, joins the project. If D2 modifies a code snippet in a programming language, all the clone instances of the code snippet in another language may require similar modifications. In particular, when a bug is reported in a programming language, D2 often needs to check versions in other languages. It can be tedious for D2 to locate the clones manually. An automated cross-language clone detection tool can be useful for D2 and reduce overlooks.

Researchers [9, 12, 10, 11] have proposed various detection approaches for code clones in one programming language. Recently, researchers [15, 3] start to detect cross-language code clones for the .NET language family. However, their approaches are limited to the languages that share a common intermediate language, while many projects are implemented in other programming languages that cannot be addressed by existing approaches. Without a common intermediate language, it becomes more challenging to detect cross-language clones. In this paper, we need to overcome the following challenges to detect such clones:

Challenge 1. Existing approaches [15, 3] can detect cross-language clones for the .NET language family, which is built on the Microsoft intermediate language. These approaches assume that different programming languages share a common intermediate language. As a result, it is feasible to reduce source code to the intermediate language and to detect clones based on such intermediates. However, most languages do not have such a common intermediate language, which makes the task challenging.

Challenge 2. Different programming languages have different grammars and APIs. As a result, even if code snippets in different programming languages implements the same functionality, their structures (even their lines of code) can be different. It becomes more challenge to determine cross-language clones than clones in a single language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970363

```
1   @@ -129,11 +129,11 @@ public class MachineProbe {
2   if (!t.isEpsilon() && !t.label.getSet().and(label).isNil() &&
          next.contains(t.target)) {
3       if (p.associatedASTNode != null) {
4 -           antlr.Token oldtoken = p.associatedASTNode.token;
5 +           Token oldtoken = p.associatedASTNode.token;
6           CommonToken token = new CommonToken(oldtoken.getType(), oldtoken.getText());
7           token.setLine(oldtoken.getLine());
8 -           token.setColumn(oldtoken.getColumn());
9 +           token.setCharPositionInLine(oldtoken.getCharPositionInLine());
10          tokens.add(token);
11          break nfaConfigLoop; // found path, move to next
12                               // NFAState set
13 ……
```

(a) MachineProbe.java

```
1   @@ -143,11 +148,11 @@ namespace Antlr3.Analysis
2   {
3       IToken oldtoken = p.associatedASTNode.Token;
4       CommonToken token = new CommonToken(oldtoken.Type, oldtoken.Text);
5 -       token.Line = (oldtoken.Line);
6 -       token.CharPositionInLine = (oldtoken.CharPositionInLine);
7 +       token.Line = oldtoken.Line;
8 +       token.CharPositionInLine = oldtoken.CharPositionInLine;
9       tokens.Add(token);
10 -      goto endNfaConfigLoop; // found path, move to next
11 -                             // NFAState set
12 +      // found path, move to next NFAState set
13 +      goto endNfaConfigLoop;
14 }
15 ……
```

(b) MachineProbe.cs

Figure 1: A Pair of Matched Diffs

In this paper, we propose a new approach, *CLCMiner*, which detects cross-language clones without intermediate languages. Our approach is based on comparing revision histories that are recorded in repository logs. Here, *Diff* is a change-log tool that is widely used in Version Control Systems (VCS) such as Git, SVN, and Mercurial. In this paper, we also call its generated delta as a *diff*. Each *diff* describes changes of a code fragment in the source code.

The rationale for our approach is that, in multi-language projects, versions in different languages can have similar *diffs* since different versions should have similar functionalities and developers may change all versions in similar ways (i.e., *diffs*) to perform similar tasks. Based on this insight, our approach detects cross-language clones through comparing the similarity among pieces of *diffs* in different programming languages and aligning each *diff* with the most similar one, which is called *diff* matching. Meanwhile, as a *diff* contains both its changed lines of code and surrounding code lines, it becomes easier to determine the granularity of cross-language clones based on *diffs*.

This paper makes the following contributions:

- To the best of our knowledge, we proposed the first approach that detects cross-language clones for programming languages that do not have an intermediate language. Our approach is based on comparing change histories, and thus reduces cross-language clone detection into a *diff* matching problem.
- We conducted an evaluation on five open source projects that release versions in Java and C#. Our results show that our approach achieves an average precision of 87% and recall of 93%.

2. RUNNING EXAMPLE

Figure 1 shows an example of two matched *diffs* in Java and C# code fragments. We use the example to illustrate the problem and how our approach works. The *diff* on the left records two lines of changes in an `if-block` in Java class `MachineProbe`, while the one on the right records four lines of changes in a `block` in C# class `MachineProbe`.

The matched *diff* pair indicates a cross-language clone, which has similar functionality. Both of the code fragments intend to set the fields (*i.e.*, `line` and `charPositionInLine`) of the object `token`. The Java code achieves this through method invocations (*i.e.*, `setLine()` and `setCharPositionInLine()`), while the C# code achieves this through assigning them directly. In addition, the Java jumps out of the `if-block` through a `break` statement, while the C# code uses a `goto` statement. Our approach extracts all the *diffs* from the project (in both Java and C#), and matches each *diff* in Java code to a *diff* in C# code according to the class name (*e.g.,* `MachineProbe`) and the text similarity (*e.g.,* the identifier names and the words). Thus, our approach is able to detect the cross-language clone in Figure 1. The detailed algorithm to match the *diffs* will be presented in Section 3.

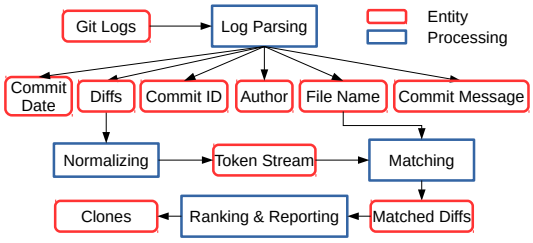


Figure 2: Approach Overview

3. APPROACH AND IMPLEMENTATION

3.1 Overview

The same functionality implemented in different languages may diverge in the syntax, but the functionality in one language (*e.g*, Java) can be used as a reference for implementation in another language (*e.g.*, C#). As a result, similar variable or method names can be used in such cases. To detect cross-language clones, *CLCMiner* adapts natural language processing (NLP) techniques to calculate the similarity among pieces of *diffs* in different programming languages and selects the most similar one for each *diff* as a pair of matched *diffs*. Each pair of matched *diffs* refers to a pair of potential clones. Base on the most similar one, we expect that other similar ones can be further detected by single-language clone detection tools. Therefore, *CLCMiner* so far does not report the second most similar or other similar ones for each *diff*. Finally, *CLCMiner* ranks the matched pairs of *diffs* according to their *diff* similarity and reports top ones as potential cross-language clones.

Figure 2 shows an overview of *CLCMiner*. Each blue rectangle represents a processing step, and each red rounded rectangle represents an entity. The input of *CLCMiner* is git logs, and its output is a ranked list of detected potential cross-language clones. The approach has four main steps:

1. **Log Parsing.** This step extracts *diffs* and their attributes from revision logs.
2. **Normalizing.** This step normalizes *diffs* and prepares for the comparison in the next step.
3. **Diff Matching.** This step matches *diffs* in different languages by comparing their similarity values. For each *diff*, its matched one is the most similar one.
4. **Ranking & Reporting.** This step ranks matched *diffs* according to their similarity and reports cross-language clones.

Table 1: Attributes of Example *Diffs*

FN	`MachineProbe.java`	`MachineProbe.cs`
CID	7288ec550b52a1b969ce6f1db62377241c36ed66	e589c63956a9e06aec08b146c2871211c13b1d56
CA	Sharwell	Sharwell
CD	Mon Mar 28 15:33:44 2011 -0800	Tue May 3 20:16:15 2011 -0800
CM	Convert all Tool grammars to ANTLR v3. The only remaining dependency on v2 is the StringTemplate 3.2's use of the v2 runtime	(C# 3) Code cleanup
TS	if t is epsilon t label get set and label is nil next contains t target if p associated ast node null antlr token oldtoken p associated ast node token token oldtoken p associated ast node token common token token new common token oldtoken get type oldtoken get text token set line oldtoken get line token set column oldtoken get column token set char position in line oldtoken get char position in line tokens add token break nfa config loop	i token oldtoken p associated ast node token common token token new common token oldtoken type oldtoken text token line oldtoken line token char position in line oldtoken char position in line token line oldtoken line token char position in line oldtoken char position in line tokens add token goto end nfa config loop goto end nfa config loop

3.2 Log Parsing

In a Version Control System (VCS), repository logs record code evolution histories. For example, the structure of git logs is organized as follows: a git log consists of several *commits*; each commit is related to one or more files; each file is related to one or more *diffs*; each *diff* records one or more change hunks that occur in a code fragment [5].

Log parsing is a preparation step that extracts useful information from repository logs. *CLCMiner* parses a log into a list of *diffs*, and attaches each *diff* with a set of *attributes*, including *Commit Date* (CD), *Commit Author* (CA), *Commit ID* (CID), *File Name* (FN), and *Commit Message* (CM). For example, Table 1 lists the attributes of the *diffs* in Figure 1. Some attributes (*e.g.*, *FN*) are useful for matching *diffs*, and others (*e.g.*, *CID*) help to uniquely locate the code.

3.3 Normalizing

Normalizing is to remove uninteresting contents from the *diffs* and transform the rest contents into normalized comparison units. *CLCMiner* uses the token streams of the *diffs* as the comparison unit, and normalizes them as follows:

1. **Removing Comments.** To relieve the impact of comments in natural language, CLCMiner removes the comments from the *diff* firstly.
2. **Lexing.** CLCMiner employs a lexer to lex the code in the *diff* without comments into a token stream.
3. **Removing Punctuations.** Punctuations and numbers are removed from the token stream, as they often do not indicate significant semantics.
4. **Post Processing.** Camel case tokens are split by the uppercase letters and tokens with underscores are split by the underscores. After that, all tokens are transformed to lowercases. This step paves difference between programming styles.

In Table 1, Column "TS" lists the two normalized token streams of the *diffs* in the running example.

3.4 Diff Matching

Diff matching is the process to align a *diff* in a language (*e.g.*, Java) to the *diff* in the other language (*e.g.*, C#), according to their similarity. Bag of Words (BOW) [8] represents a piece of text as a bag (multiset) of its words, disregarding grammar and the ordering of words. *CLCMiner* adopts BOW to build a characteristic vector, each dimension of which represents the number of times that a word appears in the token stream of a *diff*, to calculate the similarity between two *diffs*. Table 2 shows the characteristic vectors for the token streams in Table 1. Column "Token" lists the words appearing in the token streams. Columns "Java" and "C#" list the numbers of times that each word occurs in the *diff* of `MachineProbe.java` and `MachineProbe.cs` respectively. Column "Difference" lists the absolute value of the difference between the numbers of occurrence. For example,

Table 2: Characteristic Vectors

Token	Java	C#	Difference
#add	1	1	0
#and	1	0	1
#antlr	1	0	1
#associated	3	1	2
#ast	3	1	2
#break	1	0	1
...	...	...	...
#token	11	10	1
#tokens	1	1	0
#type	1	1	0
Total	80	59	61

token "break" appears in the *diff* of Java code once but does not appear in the C# code, and the difference is 1 ($|1-0|$).

We use the distance between two vectors to measure the similarity of two *diffs*. For two vectors, $V_i(v_{i1}, v_{i2}, \ldots, v_{in})$ and $V_j(v_{j1}, v_{j2}, \ldots, v_{jn})$, their distance is defined as:

$$Distance(V_i, V_j) = \frac{\sum_{k=1}^{n} |v_{ik} - v_{jk}|}{\sum_{k=1}^{n} (v_{ik} + v_{jk})}$$

In the example, the distance is $61/(80+59) = 0.4388$. The smaller the distance is, the more similar two *diffs* appear.

Algorithm 1 shows the details for matching *diffs*. It takes as input two lists of *diffs*, each of which represents changes of the code fragments in a programming language. The output is a list of matched *diff* pairs, each of which is from different input lists. *CLCMiner* compares the sizes of the two *diff* lists and sets the small one and the large one as *source* and *target* respectively (Lines 1–2). The *diffs*, whose file names are the same, are called *neighbors* of each other. For each *diff* in *source* (d_s), *CLCMiner* searches *target* for its *nearest neighbors* by comparing the distances from d_s to all of its *neighbors* in *target* (Lines 3–18). The shortest distance indicates the nearest one. As long as there exists a *neighbor* in *target* for d_s, d_s can be matched; otherwise, it cannot.

CLCMiner only matches a *diff* to its nearest *neighbor* to report clone *pairs*, instead of reporting all its top-k nearest *neighbors* to form clone *groups*. This takes into consideration that, with the nearest neighbor, the other top-k nearest neighbors and even clones in files with different names can be detected by a single-language clone detector to build more comprehensive clone groups. Section 5 discusses more about this setting for future work.

3.5 Ranking and Reporting

Each pair of matched *diffs* is called clone candidates. We rank all such pairs according to their distances. The pairs whose *diff* distances are lower than 0.5 are to be reported as code clones because it is empirically determined (*cf.* Section 4) that such short distance pairs of *diffs* are highly likely to be cross-language clones.

4. EVALUATION

We implemented *CLCMiner*, and conducted evaluations to answer the following research questions:

Algorithm 1: Diff Matching

```
Input: List dList_j dList_cs
Output: List dPair
source = minimumList(dList_j, dList_cs);
target = maximunList(dList_j, dList_cs);
foreach d_s ∈ source do
    distance ← 1;
    foreach d_t ∈ target do
        if d_t.fileName().equals(d_s.fileName()) then
            if Distance(d_s, d_t) == distance then
                pairs.add(d_s, d_t);
            end
            if Distance(d_s, d_t) < distance then
                pairs.clean();
                pairs.add(d_s, d_t);
                distance ← Distance(d_s, d_t);
            end
        end
    end
    dPair.addAll(pairs);
end
return dPair;
```

- **RQ 1.** What is the clone ratio distribution with respect to the *diff* distances?
- **RQ 2.** What is the accuracy of *CLCMiner*?
- **RQ 3.** What is the impact of the other attributes on cross-language clones?

4.1 Setup

In our evaluation, we use five open source projects implemented in both Java and C#, *i.e.*, ANTLR3, FpML, Log4j (Log4net), Spring, Lucene. Table 3 shows the projects and lists LOCs, log sizes, numbers of commits and *diffs*.

We apply our approach to each project to obtain the ranked list of cross-language clone pairs as the report. Column "#Matched Diff Pairs" in Table 3 lists the number of matched *diff* pairs according to the file name and *diff* similarity. Due to the large number of clone candidates and limited manpower, we randomly sampled, in a uniform way, a small percentage of the clone candidates in the reported ranked lists and manually labelled whether they were actual clones. As listed in Table 3, for ANTLR3, FpML, Log4j (Log4net), and Spring, we sampled over 6% of all the reported clone candidates in each project; for Lucene, we sampled about 2%. Two co-authors manually labelled whether they were actual clones separately based on the clone definition of Bellon [4] and the functionality equivalence. If there exists a difference between the labels given by the two co-authors, it will be labelled and decided by a third co-author. We calculated the clone ratio and its distribution w.r.t. the distances, where the clone ratio is defined as $CR = \frac{\#clones}{\#candidates} \times 100\%$.

4.2 Result

4.2.1 RQs 1 & 2. Distribution and Accuracy

Figure 3 shows the clone ratio distribution and the accumulated clone ratio, *w.r.t.*, *diff* distances. The clone ratio distribution in Figure 3(a) indicates: 1) almost all the candidates whose *diff* distances are lower than 0.3 are clones; 2) almost none of the candidates whose *diff* distances are larger than 0.7 is clone; 3) when distances increase from 0.3 to 0.5, the clone ratio decreases gradually; 4) when distances increase from 0.5 to 0.7, the clone ratio decreases greatly.

The accumulated clone ratio in Figure 3(b) also decreases with the increasing of the *diff* distance. When the *diff* distance is lower than 0.5, the clone ratio decreases slowly and when the *diff* distance is larger than 0.5, the clone ratio decreases greatly.

Table 3: Characteristics of Subject Projects

Projects		#LOC	Logs (MB)	#Commit	#Diffs	#Matched Diff Pairs	#Samples
ANTLR3	Java	49,617	32	572	2,839	7,117	710
	C#	97,304	31	648	18,962		
FpML	Java	17,810	244	329	2,736	3,993	259
	C#	16,548	227	183	2,206		
Log4j		30,287	46	2,644	19,172	2,599	166
Log4net		30,885	36	925	7,391		
Spring	Java	551,475	335	11,971	162,739	6,080	400
	C#	224,807	316	1,747	20,160		
Lucene	Java	867,110	821	24,988	286,628	59,377	908
	C#	434,577	883	1,320	43,073		

Based on the above observation, it is reasonable to set 0.5 as the proper threshold distance. If the *diff* distance is lower than 0.5, its related clone candidate is considered as a clone; if the *diff* distance is larger than 0.5, its related clone candidate is not considered as a clone. In other words, we only report as clones the pairs of code fragments in the ranked list whose *diff* distance is lower than 0.5.

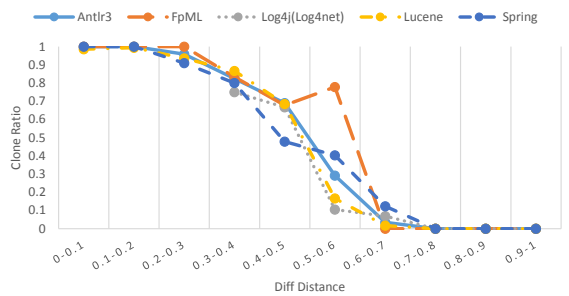
We use precision and recall to evaluate the accuracy of *CLCMiner*. In this way, for ANTLR3, FpML, Log4j (Log4net), Spring and Lucene, w.r.t. the manually labelled clone samples, the report precisions are 86%, 90%, 71%, 68% and 90% respectively and the average precision is about 87%. For the clone candidates in the five projects whose *diff* distance is between 0.5 and 1, the clone ratios are 3%, 8%, 2%, 5%, and 2% respectively. Since it is impossible to know how many actual cross-language clones in the projects, we calculate the recall based on the number of the missed clones whose *distance* is larger than 0.5. In this way, the recalls of the five projects are 90%, 97%, 71%, 69% and 98% respectively and the average recall is about 93%.

4.2.2 RQ3. Impact of More Attributes of Diffs

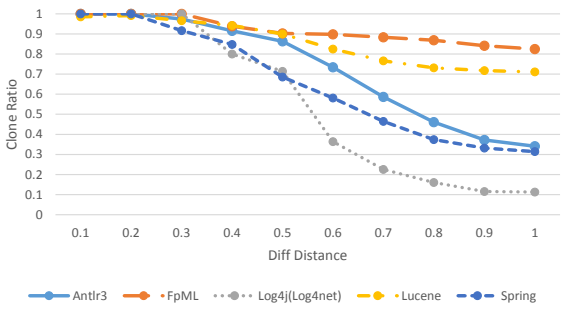
For matching *diffs*, BOW is not the only choice. We identify the following attributes that may be used to improve the effectiveness of matching cross-language clones in future.

Author. As a developer may have a programming style that may persist even across different languages, we hypothesize that a pair of *diffs* from different language versions of a project may be more likely to be clones if they are authored by the same developer. To investigate the hypothesis, we look into the labels for the clone reports sampled in the way mentioned in Section 4.1. Among these five projects, all sampled pairs of *diffs* in Spring and Log4j (Log4net) were committed by different persons; about only 0.5% of the *diff* pairs in Lucene were committed by the same developer, and about only 1% in FpML were committed by the same developer. ANTLR3 has a more pronounced difference: about 74% of *diff* pairs were made by different authors. So for each pair of *diffs* in the sampled reports, we have a variable indicating whether it is clones and another variable indicating whether it is made by the same author. A simple t-test showed that *diff* pairs made by the same author are statistically more likely to be clones than those made by different authors, but the correlation between the two variables is very weak (Pearson's correlation coefficient is about 0.08).

Commit Time. As the functionalities in different language versions of a project are likely to remain consistent, changes in one language version may induce similar changes in another within a short period of time. Similarly, we investigate whether the commit time difference between the



(a) Clone Distribution



(b) Accumulated Clone Distribution

Figure 3: Clone Ratio Distribution

two *diffs* in a reported pair is correlated with whether the pair is a clone with t-test and Pearson's correlation. We noticed that five projects exhibit different correlations between commit time differences and clones. In FpML and Spring, the *diff* pairs with *shorter* time differences are statistically more likely to be clones, but the correlation coefficients between these the two variables are very weak (-0.19 and -0.11). In Log4j (Log4Net), the effect is reversed: the *diff* pairs with *longer* time differences are statistically more likely to be clones, although the correlation is still weak (0.33). In ANTLR3 and Lucene, whether *diff* pairs are clones statistically has no effect on their time differences.

Commit Message. As a commit message often summarizes the changes in the commit, a *diff* pair may be more likely to be clones if they share similar commit messages. So we also investigate whether the distance between the commit messages of a *diff* pair is correlated with whether the pair is a clone. We calculate the distance between commit messages via the same technique we used for code (Section 3.4), and we check the relationship with t-test and Pearson's correlation. We found mixed results too as many commit messages are empty or very brief and non-informative: in FpML, ANTLR3, and Spring, clone pairs have statistically *shorter* commit message distance, while in Log4j (Log4net) and Lucene, clone pairs have statistically *longer* distance, and the correlation coefficients are all weak.

As a summary, there is no clear deciding attribute for *diff* pairs to be clones, besides the code itself. It could be a combined effect of various attributes, even some contexts beyond *diffs*. In our future work, we plan to investigate whether the combination of more attributes, together with additional ones discussed in Section 5, can be used to improve cross-language clone detection.

5. DISCUSSION AND FUTURE WORK

Using comments in code. In diff normalization (Section 3.3), code comments were removed as we hypothesized that comments in natural language may be too high-level and appear similar even for non-clones and thus are not accurate enough for clone detection. However, during the manual labelling of the sampled *diff* pair reports, we noticed that many clone pairs either contain quite different comments for different parts of the two code fragments in the pair or contain almost exactly the same comments (which may indicate an actual copying-pasting operation). In our future work, we plan to more systematically investigate how comments in code are related with clones.

Relaxing file names. Diff matching (Section 3.4) used a requirement that potentially matched diffs should be from files of the same name, and thus all code in every reported clone pair has the same file name. However, cross-language clones can appear in files with different names, especially if they are from different projects. The requirement was added to reduce the pair-wise matching time for projects involving too many commits; it is a trade-off between efficiency and recall. In the future work, we will optimize our matching algorithm and analyze how the file names impact cross-language clones that may be from different projects.

Detecting clone groups and change propagation. *CLCMiner* matches a *diff* in one language to its nearest neighbors in another language only, as we focus on the feasibility of using *diffs* for detecting cross-language clones. We can change the setting to return all the neighbors of a *diff* whose distance is within a small threshold, which can enable us to detect cross-language clone groups, in addition to pairs. Also, by linking clone groups based on clone transitivity within a threshold and complemented with a single-language detector, we will be able to study how changes are propagated even through different languages, extending similar studies within the same language [20].

Detecting clones beyond revision histories. Based on revision histories, *CLCMiner* is limited to detect cross-language clones that have been changed in the past in the same project. For clones that are never changed, we can explore more language attributes that can identify clone relations (*e.g.*, using deep learning to build vector representation of programs [18]) across languages. This limitation can also be compensated by a single-language detector that can detect cross-project and same-language clones based on certain clone transitivity across projects and languages.

Crossing more languages. Increasing demands for cross-platform mobile applications (*e.g.*, iOS and Android) raise the need for quick development that can reuse code across more diverse kinds of languages (*e.g.*, Objective-C, Swift, and Java). In our future work, we plan to adapt *CLCMiner* to more languages and explore more attributes that can identify co-change relations and be used to detect clones and facilitate code reuse across different languages.

Handling false positives. Although *CLCMiner* reports high precisions, there is still space for improvement. We investigated the false positives and found various characteristics causing "accidental similarity" among *diffs*: 1) a short method is defined in one *diff* but invoked in the other *diff*; 2) the *diffs* contain code that handles exceptions or errors; 3) the *diffs* contain a large number of same string constants used differently; 4) the *diffs* contain a number of different numeric values which were excluded by our normalizing step; 5) the *diffs* contain code that uses the same set of library functions (*e.g.*, File I/O, HttpHeaders) in different ways. In future work, we will refine *CLCMiner* to handle such cases.

Comparing with token-based clone detection. Some token-based clone detection techniques [19], can run in *plain text* mode to detect some cross-language clones. For example, *CCFinder* lexes each line of source files into token sequence and utilizes suffix-tree-based substring matching algorithm to search for similar subsequences. Different from *CCFinder*, *CLCMiner* splits each camel case identifier (*e.g.*, the variable name) and utilizes the statistical method to calculate the distance between *diffs* and search for similar *diffs*. We will compare *CLCMiner* with *CCFinder* in future work.

6. RELATED WORK

Cross-language clone detection. The number of various software systems implemented in multiple languages is increasing considerably [14], but cross-language clone detection is limited. Kraft *et al.* [15] conduct the first study on code clones that span over multiple languages. They implemented a tool called C2D2 based on the CodeDOM library in the Microsoft .NET framework, which uses NRefactory Library to generate the Unified CodeDOM graph for both C# and VB.NET. Al-omari *et al.* [3] present a clone detection approach for the .NET language family too, based on the Common Intermediate Language (CIL). It can detect cross-language clone pairs in C#, J#, and VB.NET. Compared with these work, our approach focuses on detecting cross-language clone detection on different platforms without common intermediate languages.

Data mining in VCS. There are considerable studies of data mining in Version Control Systems (VCS). Zimmermann *et al.* [21] apply data mining on version histories to recommend related syntactic changes. Gîrba *et al.* [7] apply concept analysis on VCS to identify groups of co-changes. McIntosh, *et al.* [16] mine source and test code for accompanying build changes. Meng *et al.* [17] mine revision histories to identify updated API interfaces. We mine VCS for a different purpose, *i.e.*, detecting cross-language clones.

7. CONCLUSION

This paper proposes a novel approach, *CLCMiner*, that detects cross-language clones without common intermediate languages. Our key idea is to utilize *diff* similarity. We have implemented and evaluated its prototype on five open source projects. The results show that *CLCMiner* can detect many cross-language code clones with a high precision of 87% and recall of 93% on average (*w.r.t.* distance threshold 0.5).

To improve *CLCMiner* in our future work, we plan to refine the handling of false positives, detect more cross-language clones not captured in revision histories by incorporating in single-language clone detectors, and detect more clone groups across more languages (*e.g.*, Objective-C, Swift, and Java) as described in Section 5.

8. ACKNOWLEDGMENTS

This work is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102) and the grant of Science and Technology Commission of Shanghai Municipality (No. 15DZ1100305). This work is performed during Xiao Cheng's visit to Singapore Management University (SMU) and partially supported by SMU. Also, we thank all anonymous reviewers for their feedbacks.

9. REFERENCES

[1] Antlr. http://www.antlr.org.
[2] Lucene. http://lucene.apache.org.
[3] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *Proc. WCRE*, pages 405–414, 2012.
[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *TSE*, 33(9):577–591, 2007.
[5] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10, 2009.
[6] R. Fanta and V. Rajlich. Removing clones from the code. *J. of Software Maintenance*, 11(4):223–243, 1999.
[7] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and D. Ratiu. Using concept analysis to detect co-change patterns. In *Proc. ESEC/FSE*, pages 83–89, 2007.
[8] Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.
[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.
[10] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. ISSTA*, pages 81–92, 2009.
[11] E. Jürgens, F. Deissenboeck, and B. Hummel. Clonedetective - A workbench for clone detection research. In *Proc. ICSE*, pages 603–606, 2009.
[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transaction on Software Engineering*, 28(7):654–670, 2002.
[13] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *ESME*, 13(6):645–692, Dec 2008.
[14] K. Kontogiannis, P. K. Linos, and K. Wong. Comprehension and maintenance of large-scale multi-language software applications. In *Proc. ICSM*, pages 497–500, 2006.
[15] N. A. Kraft, B. W. Bonds, and R. K. Smith. Cross-language clone detection. In *Proc. SEKE*, pages 54–59, 2008.
[16] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In *Proc. ICSME*, pages 241–250, 2014.
[17] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, pages 353–363, 2012.
[18] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *Knowledge Science, Engineering and Management (KSEM)*, pages 547–553, 2015.
[19] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.
[20] S. Wang, D. Lo, and L. Jiang. Understanding widespread changes: A taxonomic study. In *17th CSMR*, pages 5–14, 2013.
[21] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. ICSE*, pages 563–572, 2004.