Comparison and Visualization of Code Clone Detection Results

Kazuki Matsushima
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
k-matusm@ist.osaka-u.ac.jp

Katsuro Inoue
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—Many techniques for code clone detection have been proposed and implemented as clone detectors in the past. These studies show that a result of code clone detection changes drastically for different tools and/or their detection parameters. Therefore, it is important to apply different clone detectors or different parameters and to identify the different or common parts of the obtained detection results. In this paper, we propose a method for comparison and visualization of detection results based on the correspondence of clone pairs. It enables developers to compare detection results by different tools and/or their detection parameters. Using this method, we will show the comparison results of an OSS using two code clone detectors, CCFinderX and NiCad.**

***Index Terms*—Code Clone, Visualization, Software Maintenance**

I. Introduction

Code clones are some fragments of source code which are identical or similar to each other. One of the factors of the appearance of code clones is copying and pasting existing code [2], [7]. For instance, when a cloned code fragment contains a bug, it is highly likely that all of its cloned fragments contain the same bug [6] [10] [11]. Developers must check all of them for the bug and it takes a high cost. Therefore, it has been pointed out that code clone is one of the factors of increasing cost of software maintenance [5], [9], [15].

Managing and modifying code clones lead to prevent fault-proneness of systems and to decrease the number of lines of code [1]. Therefore, developers can keep high maintainability of systems. For this reason, it is important that developers identify and understand information on code clones.

Today, many techniques for automatic code clone detection have been proposed and implemented as code clone detectors. However, according to studies conducted by Roy et al. [14] and Wang et al. [20], a result of code clone detection changes drastically for different tools and/or their detection parameters.

Therefore, it is important to apply different clone detectors or different parameters and identify the different or common part of the obtained detection results. Although code clone detection results have been compared in view of recall and precision in these studies, few studies have focused on the different or common part of obtained detection results.

In this paper, we propose a method for comparison and visualization of clone detection results based on the correspondence of clone pairs. Also, we conducted an experiment to apply our method to *JEdit* source code [23] using two code clone detectors, *CCFinderX* [22] and *NiCad* [13] with their default parameters. From the result of the comparison, 63% of the clone pairs detected by *CCFinderX* is not detected by *NiCad*. Conversely, 43% of the clone pairs detected by *NiCad* is not detected by *CCFinderX*. In addition, we found a clone pair which was detected by *CCFinderX* but not detected by *NiCad* contains a bug.

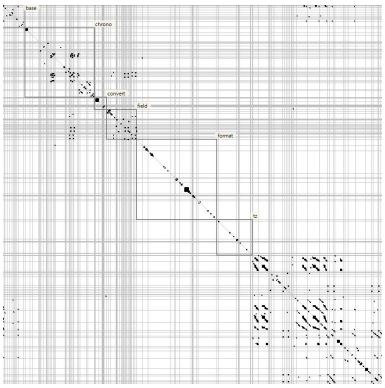


Fig. 1. Example of a scatter plot generated by *CCFinderX*

This paper is organized as follows. In Section II, we introduce background and motivation of this research. Section III describes a method to compare and visualize code clone detection results. In Section IV, we show a conducted experiment and discuss its result. Section V discusses on our approach with other works, and Section VI mentions limitations and threats to validity. Finally, we summarize this study in Section VII.

II. Background

A. Visualization of a Code Clone Detection Result

One of the popular methods for visualization of a code clone detection result is the scatter plot (dot plot) [19]. A scatter plot is a figure which shows the location of code clone in source code.

Fig. 1 shows an example of a scatter plot generated by *CCFinderX* with an associated GUI *GemX* [19]. Both X and

TABLE I
PARAMETERS FOR CCFINDERX

Minimum Clone Length	50
Minimum TKS(Token Kinds)	12
Shaper Level	2 - Soft shaper
P-match Application	Use P-match

Y axes in this figure represent the sequence of tokens of source code, and dots are plotted if tokens on X and Y axes are the same. By looking at a scatter plot, developers can comprehend the location and length of code clones in source code easily.

B. Problems on Code Clone Analysis

Analyzing code clone detection results and performing maintenance activities enable developers to improve the quality of their software [4]. However, existing studies show that a code clone detection result changes drastically based on characteristics of code clone detectors or its detection parameters [14], [18].

For instance, Bellon et al. [3] compared two code clone detection results, one by the default parameters and the other by the optimized parameters of *CCFinder* [8] with respect to the recall and precision for the reference set of code clones. The result showed that the precision of the detection result by the optimized parameters was three times or higher than the one by the default parameters. In contrast, there is no change in the recall.

Wang et al. [20] also compared the six code clone detection results from different six code clone detectors line by line. The result showed that the lines reported as cloned by a code clone detector were not reported as cloned by most other code clone detectors. Furthermore, the number of lines reported as cloned by just six code clone detectors was about one-tenth of the number of lines reported as cloned by one or more code clone detectors.

In addition to the difference of clone detectors, we experience the difference of parameter setting for a clone detector. For example, for Apache Ant [21] 1.9.14 as a target system and *CCFinderX* as a code clone detector, we confirm substantial difference of the code clone detection results by difference of the parameters. With the parameters shown in Tab. I, *CCFinderX* reports 579 clone sets and generated the scatter plot shown in Fig. 2. Next, we altered the Minimum TKS (the number of different kinds of token in a detected clone) parameter from 12 to 20 and detected clones. *CCFinderX* reports 154 clone sets and generated the scatter plot shown in Fig. 3. Comparing these detection results, the drastic reduction in the number of clone sets reported by *CCFinderX* was caused by an increase in the Minimum TKS parameter. The number of files containing any code clones also decreased.

Thus, developers may miss important code clones by analyzing only a single code clone detection result. If important code clones are not included in it, it is difficult for developers to consider and perform appropriate maintenance activities.

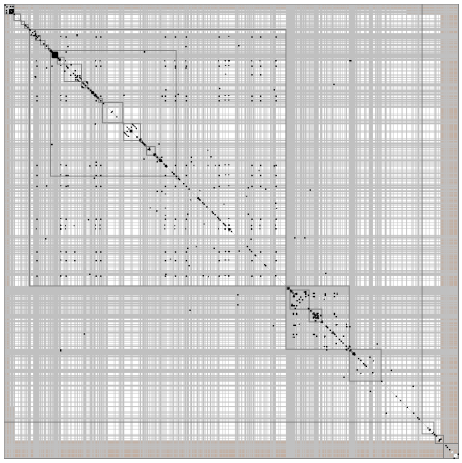


Fig. 2. Detection result of Apache Ant

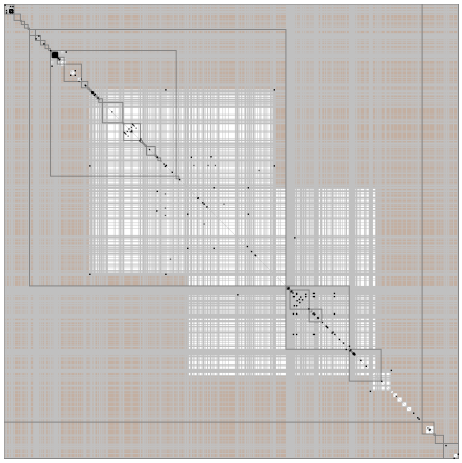


Fig. 3. Detection result of Apache Ant with the altered parameters

III. PROPOSED METHOD

In this section, we describe an overview of the proposed method.

As shown in Fig. 4, our proposed method is realized in the following four steps; (A) Detect code clones, (B) Map clone pairs, (C) Calculate mismatch rate, (D) Visualize the difference. We explain these steps in more detail.

A. Step A: Detect Code Clones

First, code clones are detected from a target system. Developers can configure multiple detectors to use with and multiple sets of parameters of each detector at the same time.

Here, a fragment *f* of source code is formally denoted by a tuple $\langle file, begin, end \rangle$ where *file* represents the name of the source file containing the fragment, *start* represents the start position of the fragment in the file, and *end* represents the end position of the fragment in the file. *NiCad* outputs code-clone fragments consisting of the name of the source file, the start line, and the end line. In contrast, *CCFinderX* outputs code clone fragments consisting of the ID of the source file, the ID of the start token, and the ID of end token.

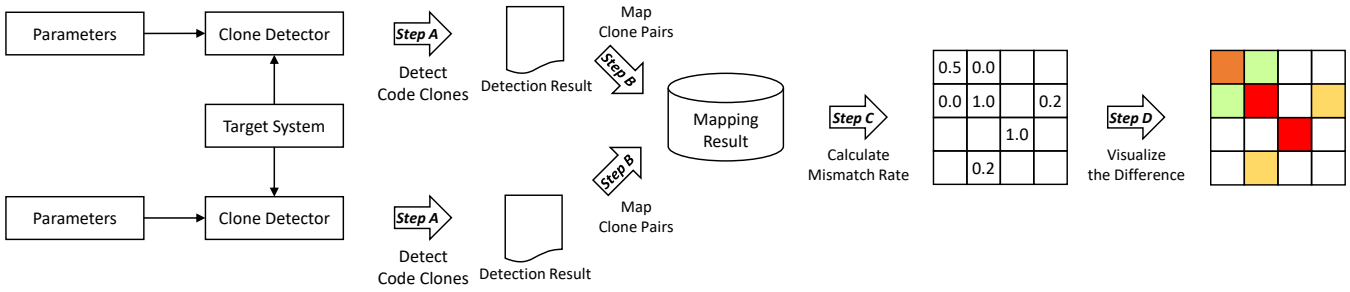


Fig. 4. Overview of our proposed method

Thus, all code clone detection results obtained are converted into a standard format where any code fragment consists of the ID of the source file, the start line number, and the end line number. Furthermore, we introduce an order relation on two code fragments f_1 and f_2 as follows:

$$\begin{aligned} f_1 < f_2 \Longleftrightarrow\ & (f_1.fileId < f_2.fileId) \vee \\ & (f_1.fileId = f_2.fileId \wedge \\ & \ f_1.beginLine < f_2.beginLine) \vee \\ & (f_1.fileId = f_2.fileId \wedge \\ & \ f_1.beginLine = f_2.beginLine \wedge \\ & \ f_1.endLine < f_2.endLine) \end{aligned}$$

This means that f_1 precedes f_2 if the file name of f_1 precedes f_2, or so if the start line or end line of f_1 precedes f_2 when they are in the same file. Thus, we assume that two code fragments of any single clone pair p satisfy $p.f_1 < p.f_2$. In case that there is a clone pair not satisfying this order, we swap the order in p.

B. Step B: Map Clone Pairs

Secondly, we map clone pairs between different code clone detection results.

To identify clone pairs to map, we use the method of matching two code fragments based on two metrics, *ok-value* and *good-value*, proposed in [3], instead of using a perfect matching of fragments of each clone pair. This is because we frequently experience difference of handling of comments, empty lines, or braces of code clone detectors, and so the fragments of clone pairs do not match exactly with each other. Matching based on *ok-value* and *good-value* allows mapping of slightly different clone pairs.

Hereinafter, for any code fragment f, $lines(f)$ denotes a set of lines contained in f. Also, $|lines(f)|$ denotes the number of lines in f.

We define two functions $overlap$ and $contained$ used for measuring *ok-value* and *good-value* as follows:

Definitions of $overlap$ and $contained$

For any two code fragments f_1 and f_2,

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|}$$

$$contained(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|}$$

Function $overlap$ measures the ratio of a code fragment overlapping to another one. Function $contained$ measures the ratio of a code fragment containing another one.

Now we define functions *good* and *ok* as follows:

Definitions of *good* and *ok*

For any two clone pairs p_1 and p_2,

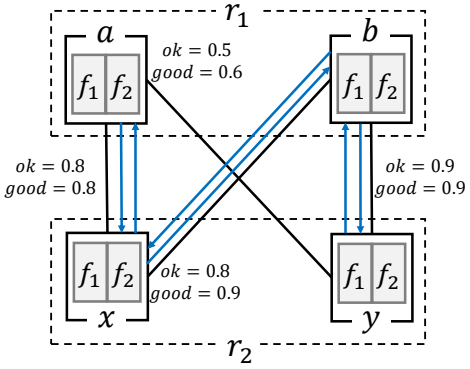
$$\begin{aligned} good(p_1, p_2) = min(& overlap(p_1.f_1, p_2.f_1), \\ & overlap(p_1.f_2, p_2.f_2)) \\ ok(p_1, p_2) = min(& max(contained(p_1.f_1, p_2.f_1), \\ & contained(p_2.f_1, p_1.f_1)), \\ & max(contained(p_1.f_2, p_2.f_2), \\ & contained(p_2.f_2, p_1.f_2))) \end{aligned}$$

Intuitively, $good$ indicates an overlap degree of a clone pair (not a fragment pair), and ok means its containment degree.

Algorithm 1 Map clone pairs in r into those in r'

```
for each clone pair q in r' do
  for each clone pair p in r do
    ok_max ⇐ ok(p, mapped[p])
    good_max ⇐ good(p, mapped[p])
    ok ⇐ ok(p, q)
    good ⇐ good(p, q)
    if better then
      mapped[p] ⇐ q
    end if
  end for
end for
```

Algorithm 1 is an algorithm for mapping clone pairs from a code clone detection result r to r' [3]. However, in our



Black edge: possible mapping
Blue arrow: resulting mapping

Fig. 5. Example of mapping clone pairs

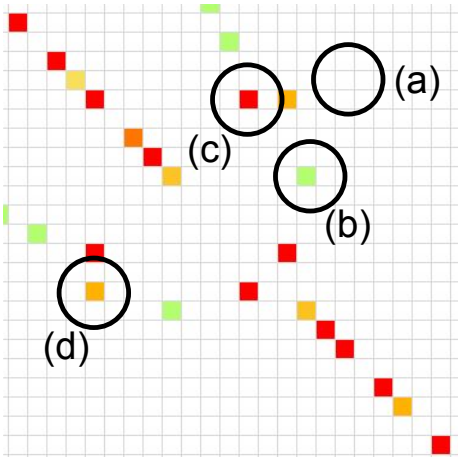


Fig. 6. Example of a scatter plot generated through our proposed method

method, we have slightly simplified the condition $better$ from the original one. The definition of the condition $better$ in our method is as follows:

$$better = (good \geq t \wedge good > good\_max) \vee (ok \geq t \wedge ok > ok\_max)$$

We use 0.7 as a threshold t in the condition $better$, which is the same as [3].

Now mapping $Map(r_1, r_2)$ between two code clone detection results r_1 and r_2 is determined by the following three steps.

(a) Map clone pairs from r_1 to r_2 by Algorithm 1.
(b) Map clone pairs from r_2 to r_1 by Algorithm 1.
(c) Calculate the union of the mapping results by the step (a) and step (b).

Fig. 5 is an example of mapping clone pairs. A code clone detection result r_1 includes two clone pairs a and b. Also, another detection result r_2 includes two clone pairs x and y.

First, clone pairs are mapped from r_1 to r_2 by following step (a). By *ok-value* and *good-value*, a is mapped to x and b is mapped to y.

Then, clone pairs are mapped from r_2 to r_1 by following step (b). By *ok-value* and *good-value*, x is mapped to b and y is mapped to b.

Finally, calculate the union of correspondence of clone pairs obtained with step (a) and step (b) by following step (c). In this step, a is mapped to x, b is mapped to x and y, x is mapped to a and b, and y is mapped to b.

C. Step C: Calculate the Mismatch Rate

Next, mismatch rate is calculated. Mismatch rate for results r and r' represents the difference of r' from r. The definition is as follows:

Definition of the mismatch rate of r' for r

$P_{all}(r, s_1, s_2)$ is a set of clone pairs between two source files s_1 and s_2, which generate code clone detection result r.
$P_{mapped}(r, r', s_1, s_2)$ is a set of clone pairs which are included in r' and are mapped from r into r' by $Map(r, r')$. When comparing r and r' based on r, mismatch rate $m(r, r', s_1, s_2)$ is

$$m(r, r', s_1, s_2) = 1 - \frac{|P_{mapped}(r, r', s_1, s_2)|}{|P_{all}(r, s_1, s_2)|}$$

The mismatch rate is calculated for each pair of source files.

D. Step D: Visualize the Difference

Finally, the difference between two code clone detection results is visualized by colorizing each point depending on its mismatch rate. Here r and r' are code clone detection results and s_1 and s_2 are target source files, and the rule of colorization at the point (s_1, s_2) is as follows:

1) When $P_{all}(r, s_1, s_2)$ equals to zero (meaning no clones in r), the point is white (as shown in Fig. 6 (a).
2) When $m(r, r', s_1, s_2)$ equals to zero (meaning perfect matching), the point is green (b).
3) The other cases, the higher mismatch rate is the redder the point is. Conversely, the lower mismatch rate is the yellower the point is (c, d).

By colorizing each point depending on its mismatch rate, developers can comprehend the difference between code clone detection results intuitively.

IV. Experiment

In this section, we show the comparison results of an OSS using two code clone detectors and discuss it.

We used an open-source software *JEdit* [23] Revision r24577 as a target system. Some existing studies targeted *JEdit* as well [5] [11] [20]. The size of the target is 113,826 Loc in Java. We also used two code clone detectors, *NiCad* and *CCFinderX* as code clone detectors. We have used default

TABLE II
PARAMETERS FOR CCFINDERX

Minimum Clone Length	50
Minimum TKS (Token Kinds)	12

TABLE III
PARAMETERS FOR NICAD

granularity	blocks	rename	blind
threshold	0.3	filter	none
minsize	10	abstract	none
maxsize	2,500	normalize	none
transform	none		

parameters of *NiCad* and *CCFinderX* for this experiment, which are shown in Tab. II and III.

We have compared two code clone detection results with our proposed method.

A. Overview

First, we look over two code clone detection results before comparison. The number of the detected clone pairs and execution time on a workstation with Xeon E5-1620 3.60GHz 4 cores CPU and 32 GB memory are presented in Tab. IV.

We show the scatter plot of the detection result of *CCFinderX* generated by *GemX* in Fig. 7. Scatter plot from the *NiCad*'s detection result cannot be presented here because *GemX* does not support *NiCad*'s output format.

Now we compare these two detection results. It took about 10 milliseconds for the mapping clone pairs (Step B), 4 milliseconds for calculating mismatch rate (Step C), and 0.01 millisecond for visualizing the difference (Step D).

Fig. 8 shows the scatter plot of the comparison result based on the code clone detection result of *NiCad*. Also, Fig. 9 shows the scatter plot of the comparison result based on the one of *CCFinderX*. *CCFinderX* reports 2,771 clone pairs and 63% of them are not mapped to ones reported by *NiCad*. On the other hand, *NiCad* reports 2,518 clone pairs and 43% of them are not mapped to ones reported by *CCFinderX*.

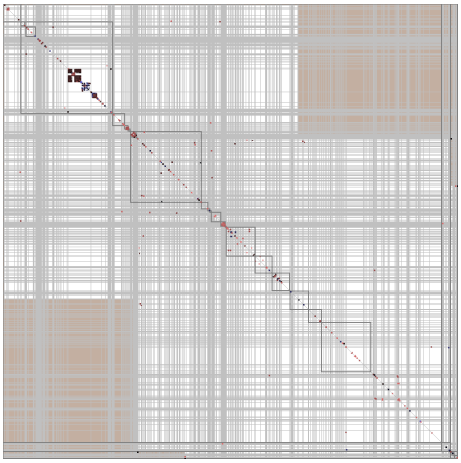


Fig. 7. Scatter plot of the detection result of *CCFinderX* generated by *GemX*

TABLE IV
NUMBER OF DETECTED CLONE PAIRS AND EXECUTION TIME

Detector	# Clone Pairs	Detection Exe. Time (sec.)
CCFinderX	2,771	72
NiCad	2,518	28

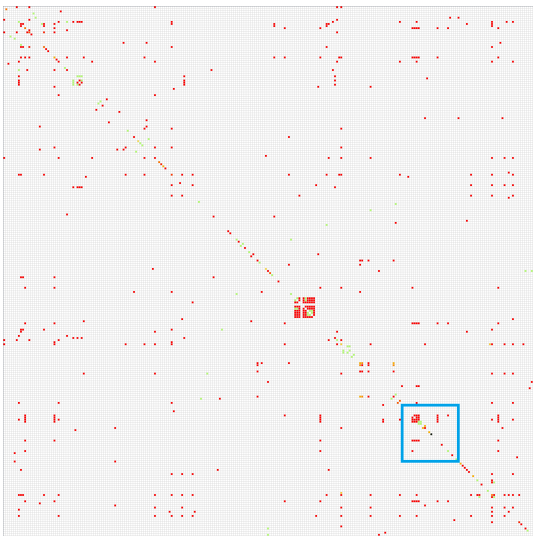


Fig. 8. Scatter plot of the comparison result based on the detection result of *NiCad*

As you can see and identify easily, both figures contains many red points meaning existence of high rate of mismatching, and also there are smaller number of green points showing complete matching. This means that, although the number of the detected clone pairs are similar around 25 hundreds, there are a lot of clone pairs in both results, which cannot be mapped

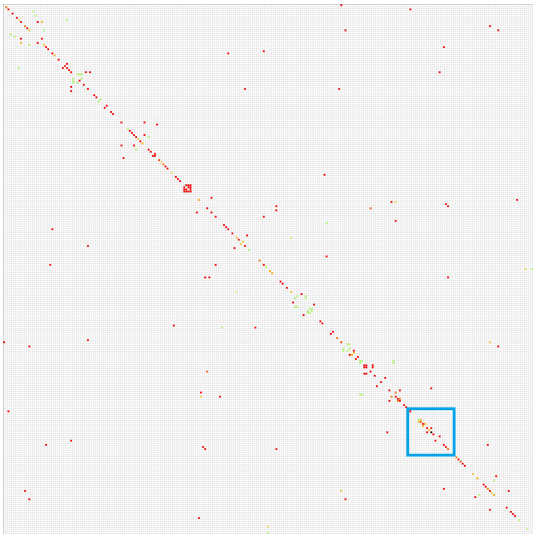


Fig. 9. Scatter plot of the comparison result based on the detection result of *CCFinderX*

```
210  String checkMarginsMessage = pageSetupPanel.recalculate();
211  if ( checkMarginsMessage != null )
212  {
213      JOptionPane.showMessageDialog(
 :
230  PageRanges pr = ( PageRanges )PrinterDialog.this.attributes.get( PageRanges.class );
231  try
232
233
234  {
235      pr = mergeRanges( pr );
236      PrinterDialog.this.attributes.add( pr );
237  }
238  catch ( PrintException e )
239  {
240      e.printStackTrace();
241      JOptionPane.showMessageDialog( PrinterDialog.this, jEdit.getProperty("print-error.message", new
          String[]{e.getMessage()} ), jEdit.getProperty( "print-error.title" ),JOptionPane.ERROR_MESSAGE );
242      return;
243  }
```

Fig. 10. Buggy code fragment included in *JEdit*

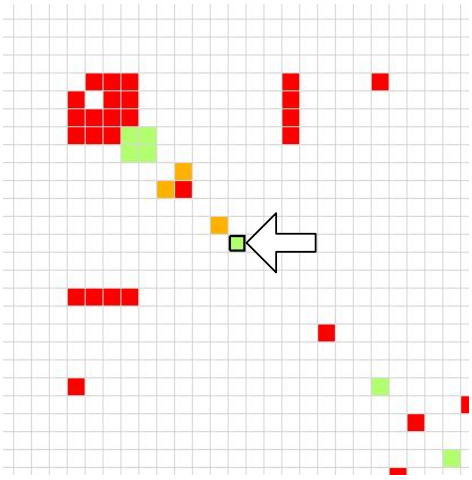


Fig. 11. Partially enlarged figure within the blue box Fig. 8

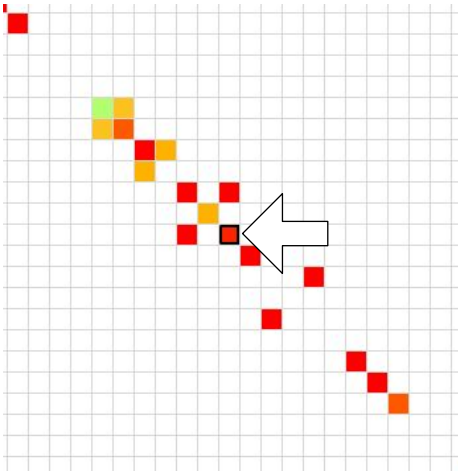


Fig. 12. Partially enlarged figure within the blue box in Fig. 9

each other are needed to be analyzed further.

B. Details of the Comparison Result

Next, we investigated the comparison result focusing on small areas with red points.

Fig. 11 extends the blue square area of Fig. 8. The black-edged point located in the center of Fig. 11 represents the mismatch rate at the point $(PrinterDialog.java, PrinterDialog.java)$ and its color is green. The meaning of this is that all of clone pairs which are composed of two code fragments in $PrinterDialog.java$ detected by *NiCad* are also detected by *CCFinderX*. At this point, *NiCad* reported one clone pair and it was also reported by *CCFinderX*

Fig. 12 extends the blue square area of Fig. 9. The black-edged point located in the center of Fig. 12 also represents the mismatch rate at the point $(PrinterDialog.java, PrinterDialog.java)$ and its color is red. The meaning of this is that almost all clone pairs which are composed of two code fragments in $PrinterDialog.java$ detected by *CCFinderX* are not detected by *NiCad*. At this point, *CCFinderX* reported 20 clone pairs and only one of them was also reported by *NiCad*.

By investigating remaining 19 clone pairs, we have found that one of them contained a buggy code fragment as shown in Fig. 10. Due to the reuse of variables and the lack of a check for a null parameter, the code fragment causes NullPointerException. Actually, the code fragment and its code clone were fixed simultaneously at Revision r24578 which is the next to the target revision in this experiment.

Through this investigation, we have recognized importance of analyzing code clone differences in detail.

V. Related Works

As presented in Section II, Bellon et al. [3] have investigated the results of various code clone detectors. They have

introduced the mapping algorithm we have used here, and calculated the recall and precision values for the reference set, which would help to evaluate the detectors quantitatively. We have taken a similar approach, but our goal is to identify difference of code-clone pairs for different detectors or parameters, and visualize the difference for intuitive understanding of the risk of the detection results.

Wang et al. [20] presented a search-based approach to find a better configuration of clone detection techniques. They have developed a system named EvaClone, and have performed a large-scale experiment with 6 detectors and 8 subject systems. Their objectives are also quantitative evaluation of detection results and improvement of configuration with respect to the fitness function. On the other hand, we are interested in intuitive understanding of the difference of the detection results at the levels of file to file or clone to clone mapping and matching.

Stephan et al. [16] presented methods for assessment of model clone detectors and techniques. They have created rules which convert a tool's native output format into a normalized format as we did and their rules are for models. In contrast, our rules are for codes and intended for visualization of detection results.

Svajlenko et al. [17] evaluated the recall of 11 clone detectors using 4 benchmark frameworks. They have extended the definition of the recalls of Murakami's type 3 aware one [12], and applied it to those benchmarks, finding anomalies in Bellon's benchmark [3]. These suggest important improvement of our mapping method based on the original Bellon's approach, and we will further elaborate more accurate and practical mapping as future studies.

VI. Limitation and Threats to Validity

In the experiment, we have conducted a very limited analysis. We have found a bug-prone clone in a red point, but further investigate is needed for interpreting the meaning of the red points. Also, We need to apply our approach with various clone detectors, parameters, and targets, to understand the risks of the red, yellow, and other points.

We have proposed the mismatch rate based on one of two clone detection results. This approach generates two different scatter plots of the mismatching rate as shown in Figure 8 and 9. Since the interpretation of two plots might not be straightforward, we would need to consider a method of merging them.

We have used 0.7 as threshold value t, but further investigation is needed for a better value.

VII. Conclusions

In this paper, we have presented a method for comparison and visualization of different code clone detection results, focusing on the mapping of clone pairs. Developers can identify different or common parts of the obtained clone detection results intuitively with scatter plots generated by our method. We have conducted an experiment to apply our method to *JEdit* using *CCFinderX* and *NiCad*. From the result of the comparison, we have confirmed that a large number of clone pairs reported by one tool is not reported by the other one.

We will extend the comparison method to allow more than two clone detection results for different tools or parameters.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP18H04094.

References

[1] L. Barbour, F. Khomh, and Y. Zou, "An empirical study of faults in late propagation clone genealogies," Journal of Software: Evolution and Process, vol. 25, pp. 1139-1165, 2013.
[2] I. D. Baxter, A. Yahin, L. Moura, M Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in Proc. ICSM 1998, pp. 368-377, 1998.
[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, vol. 33, No. 9, pp. 577-591, 2007.
[4] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring support based on code clone analysis," Product Focused Software Process Improvement, pp. 220-233, 2004.
[5] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study of software bugs in clone and non-clone code," in Proc. SEKE 2017, pp. 436-443, 2017.
[6] P. Jablonski and D. Hou, "CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in Proc. ETX 2007, pp. 16-20, 2007.
[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in Proc. ICSE 2009, pp. 485-495, 2009.
[8] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol. 28, pp. 654–670, 2002.
[9] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in Proc. ICSM 1997, pp. 314-321, 1997.
[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in Proc. OSDI 2004, vol. 6, pp. 289-302, 2004.
[11] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in Proc. ICSME 2017, pp. 227-237, 2017.
[12] H. Murakami, Y. Higo, and S. Kusumoto, "A dataset of clone references with gaps," in Proc. MSR 2014, pp. 412–415, 2014.
[13] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in Proc. ICPC 2008, pp. 172-181, 2008.
[14] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, No. 7, pp. 470–495, 2009.
[15] N. Saini, S. Singh, and Suman, "Code clones: Detection and management," Procedia Computer Science, vol. 132, pp. 718-727, 2018.
[16] M. Stephan and J. R. Cordy, "MuMonDE: A framework for evaluating model clone detectors using model mutation analysis," Software Testing Verification and Reliability, vol. 21, no. 1-2, p. e1669, 2018.
[17] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools", in Proc. ICSME 2014, pp. 321-330, 2014.
[18] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in Proc. ICSME 2015, pp. 131-140, 2015.
[19] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Code clone analysis tool," in Proc. ISESE 2002, pp. 31-32, 2002.
[20] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in Proc. ESEC/FSE 2013, pp. 455-465, 2013.
[21] Apache Ant. https://ant.apache.org/.
[22] CCFinderX. http://www.ccfinder.net/ccfinderxos.html.
[23] JEdit. http://www.jedit.org/.