社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

複数コードクローン検出結果の比較・表示法

松島　一樹†　井上　克郎†

† 大阪大学大学院情報科学研究科　〒560–0871 大阪府吹田市山田丘 1–5
E-mail: †{k-matusm,inoue}@ist.osaka-u.ac.jp

あらまし　コードクローンを自動的に検出するための様々な手法が提案されてきており，コードクローン検出ツールとして実装されてきた．しかし，同一のソースコード集合に対してであっても，コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが既存研究から明らかとなっている．そこで本研究では，同一ソースコード集合を対象とした複数のコードクローン検出結果に対してクローンペア間の対応に注目し，定性的・定量的な評価を容易にする比較・表示法を提案する．

キーワード　コードクローン，ソフトウェア保守

A Method for Comparison and Visualization of Code Clone Detection Results

Kazuki MATSUSHIMA† and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University　1–5, Yamadaoka, Suita, Osaka, 560–0871, Japan
E-mail: †{k-matusm,inoue}@ist.osaka-u.ac.jp

Abstract　Many techniques for automatic code clone detection have been proposed and implemented as clone detection in the past. However, existing studies show that the result of code clone detection changes drastically in accordance with the tool and its detection parameters. In this study, we propose a method for comparison and visualization of code clone detection results focused on the correspondence of clone pairs.

Key words　code clone, software maintenance

1. まえがき

コードクローンとは，ソースコード中に存在する「全く同一のコード片」もしくは「類似したコード片」のことである．コードクローンの発生原因として，主に既存のソースコードのコピー・アンド・ペーストによる再利用や，コード生成ツールによる自動生成が挙げられる [1]．もし，欠陥が含まれたコード片にコードクローンが存在する場合，それらすべてにも欠陥が含まれている可能性が高い．開発者はすべてのコードクローンに対して修正を行うか検討する必要があり，欠陥の修正に大きなコストが必要となる．そのため，コードクローンはソフトウェア開発の保守工程における大きな問題の 1 つとして指摘されている．

開発者がコードクローンの管理や修正を行うことで，潜在的な欠陥の増大の抑制やソースコード量の削減につながり，ソフトウェアの保守性を高く保つことができる．そのため，開発者がコードクローンに関する情報を認識することは重要である．

コードクローンを自動的に検出するための様々な手法が開発されてきており，コードクローン検出ツールとして実装されてきた [2] [9]．一方 Bellon ら [3] や Wang ら [4] の研究から，同一のソースコード集合に対してであっても，コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが明らかになっている．そのため，様々なコードクローン検出ツールやパラメータでコードクローン検出を行い，得られた結果の共通部分や差異を知ることは重要である．

しかし，これらの研究を始めとしてコードクローン検出結果はベンチマークに対する再現率や適合率といった基準で比較されてきたが，含まれているコードクローンの共通部分や差異といった基準での比較は行われていなかった．

そこで，本研究では同一ソースコード集合を対象とした複数のコードクローン検出結果に対してクローンペア間の対応に注目し，定性的・定量的な評価を容易にする比較・表示法を提案する．

また，提案手法を実際のソフトウェアに適用し，コードクローン検出結果を比較することで重要なコードクローンを発見できるかについて実験を行った．

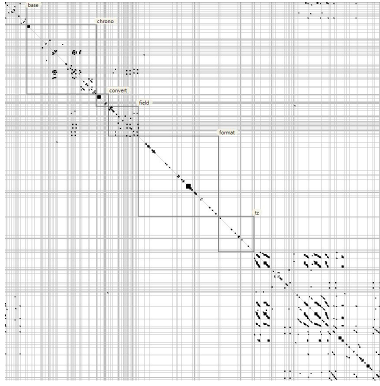

図 1 GemX の散布図

以降，2 章では，本研究の背景としてコードクローンと主な検出手法，コードクローン分析における問題について述べる．3 章では，提案手法について述べる．4 章では，提案手法の適用実験について述べる．最後に，5 章では，まとめについて述べる．

2. 背 景

2.1 コードクローン

コードクローンとは，ソースコード中に存在する「全く同一のコード片」もしくは「類似したコード片」のことである．コードクローンは，主に既存のソースコードのコピー・アンド・ペーストによる再利用やコード生成ツールによる自動生成によって生じる．一般的に互いにコードクローンとなる 2 つのコード片の組をクローンペアと呼び，コードクローンの同値類をクローンセットと呼ぶ [6]．

2.2 コードクローン検出と分析

2.2.1 主なコードクローン検出手法

コードクローンを自動的に検出するための様々な手法が提案されている [2] [9]．主な手法として，テキストベースの検出やトークンベースの検出などが提案されている．

2.2.2 コードクローン分析における問題

開発者はコードクローン検出結果を分析することで，コードクローンに対して保守作業を検討する．

保守作業の例として以下の 2 つが挙げられる [2]．

集約
クローンセット中のコード片と同様の処理を実装するサブルーチンを作り，各コード片をそのサブルーチンの呼び出し文に置き換えることである．集約により，コードクローンの数とソースコード量を削減する．

同時修正
あるコード片を修正する際に，そのコード片のコードクローンに対しても一貫した修正を行うことである．

このように，コードクローン検出結果を分析しコードクローンに対して保守作業を行うことで，ソフトウェアの品質を向上させることができる．しかし，同一のソースコード集合に対してであっても，コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが既存研究から明らかになっている．

Bellon らは論文 [3] において，CCFinder のデフォルトパラメータと調整したパラメータで検出されるコードクローンの正解集合に対する適合率と再現率を比較した．その結果，調整したパラメータはデフォルトパラメータと比べ，再現率が変化しないまま適合率が 3 倍以上向上していた．

また，Wang らは論文 [4] において，6 つのコードクローン検出結果を行単位で比較した．その結果，ある 1 つのコードクローン検出ツールで検出されたコードクローンの多くはほかのコードクローン検出ツールでは検出されないものであった．また，全コードクローン検出ツールで共通して得られるコードクローンは 10 % 程度であった．

このように，ただ 1 つのコードクローン検出結果だけでは着目すべきコードクローンが含まれていない可能性がある．着目すべきコード片がコードクローン検出結果に含まれていなければ，開発者がソースコードに対する適切な保守を行うことは困難である．

2.2.3 コードクローン検出結果の可視化手法

コードクローン検出結果を可視化する既存の手法として，クローン散布図が挙げられる．クローン散布図とは，ソースコード中のどの位置にコードクローンが存在するかをプロットした図である [10]．

CCFinderX(注1) に付属するコードクローン分析ツール GemX で実際に用いられるクローン散布図を図 1 に示す．この図では，左上角を原点として，横軸・縦軸共にソースコードのトークン列を表す．そして，両軸の対応するトークンが一致している場合に点が描画されている．また，描画される点は左上角から右下角への対角線を軸として線対称となっている．クローン散布図を見ることで，ソースコード中のどの位置に，どの程度の大きさのコードクローンが存在しているかを容易に理解することができる．

3. 提 案 手 法

本章では，提案手法の処理概要とその各ステップの詳細について述べる．

まず，提案手法の処理概要を図 2 に示す．

提案手法は図 2 に示すように，4 つのステップで処理が行われる．

（1） 対象プロジェクトからコードクローンを検出
（2） コードクローン検出ツールから出力されたコードクローン検出結果を共通フォーマットへ変換
（3） 同じ部分を指すクローンペア同士の対応付け
（4） 比較結果を可視化し，開発者に提示

以降の節でそれぞれについて詳細を述べる．

3.1 ステップ 1: コードクローン検出

まず，対象プロジェクトに対してコードクローン検出を行う．開発者は，使用するコードクローン検出ツールとそのパラメータの組を複数設定することができる．そして，入力されたパラ

(注1)：http://www.ccfinder.net/ccfinderxos-j.html

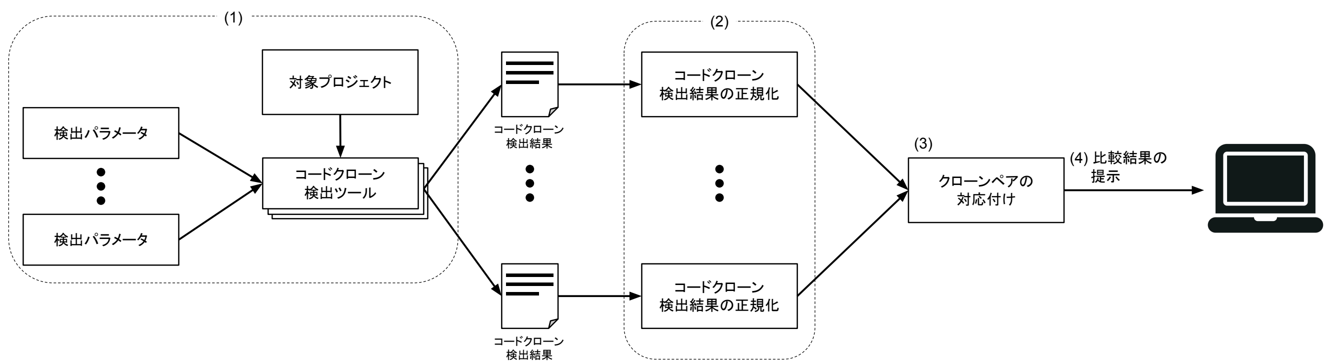

図 2　提案手法の概要図

メータを開発者が指定したコードクローン検出ツールに引き渡し，コードクローン検出を行う．提案手法で対応しているコードクローン検出ツールは，CCFinderX，NiCad [5]，CCVolti [7]，CCFinderSW [8] の 4 つである．

3.2　ステップ 2: コードクローン検出結果の正規化

ステップ 2 ではコードクローン検出ツールから出力されたコードクローン検出結果を共通フォーマットへ変換する．

コード片は一般的に，<ファイル，開始位置，終了位置> の 3 つの要素から定義される．しかし，これらの出力形式はコードクローン検出ツールにより様々である．例えば CCFinderX であれば <ファイル ID，開始トークン番号，終了トークン番号> を用いて出力される．一方，NiCad であれば，<ファイルパス，開始行番号，終了行番号> を用いて出力される．

そこで，提案手法では比較を簡単にするために，各コードクローン検出結果を共通フォーマットと呼ぶ <ファイル ID，開始行番号，終了行番号> の 3 つ組の列に変換する．

以降，コード片 f のファイル ID を fid，開始行番号を b，終了行番号を e とし，それぞれ $f.fid$，$f.b$，$f.e$ のように表す．

更に，コード片 f_1，f_2 に対して次のような順序関係を定める..

$$
\begin{aligned}
f_1 < f_2 \Leftrightarrow & (f_1.fid < f_2.fid) \lor \\
& (f_1.fid = f_2.fid \land f_1.b < f_2.b) \lor \\
& (f_1.fid = f_2.fid \land f_1.b = f_2.b \land f_1.e < f_2.e)
\end{aligned}
$$

そして，任意のクローンペア p を構成する 2 つのコード片 f_1，f_2 について，常に $p.f_1 < p.f_2$ が成り立つものとする．この順序関係が成り立たないクローンペアについては，条件を満たすように f_1 と f_2 の入れ替えを行う．

ステップ 1 により，すべてのコードクローン検出結果を正規化し，統一された形式で扱うことが可能になる．

3.3　ステップ 3: クローンペアの対応付け

ステップ 3 では複数の検出結果間でクローンペア同士の対応付け（クローンペアマッピング）を行う．

対応するクローンペアの判定には，クローンペアを構成するコード片の完全一致ではなく *ok* 値と *good* 値に基づいて行う [3].

```
for each q in r' do
    for each 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
```

図 3　r から r' へのクローンペアマッピングのアルゴリズム

コードクローン検出ツールの空行や改行の扱いの違いにより，同じ部分を指しているクローンペアであってもコード片の範囲がずれることがある．*ok* 値と *good* 値に基づいてマッピングを行うことで，範囲がずれているコード片で構成されたクローンペア同士も対応していると判定することが可能となる．

以降，コード片 f に対して，$lines(f)$ が f に含まれるソースコード行の集合を表し，$|lines(f)|$ が $lines(f)$ の要素数，つまり f の行数を表す．

まず，*good* 値と *ok* 値を求めるために用いられる関数 *overlap* と *contained* の定義を以下に示す．

$overlap$ と $contained$ の定義

任意のコード片 f_1，f_2 に対して

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

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

overlap は 2 つのコード片同士の重複度を表し，*contained* は f_1 に対する f_2 の包含度を表す．

次に *good* 値と *ok* 値の定義を以下に示す．

$good$ 値と ok 値の定義

任意のクローンペア p_1，p_2 に対して

$$good(p_1, p_2) = min(overlap(p_1.f_1, p_2.f_1), overlap(p_1.f_2, p_2.f_2))$$

$$ok(p_1, p_2) = min(max(contained(p_1.f_1, p_2.f_1), contained(p_2.f_1, p_1.f_1)) max(contained(p_1.f_2, p_2.f_2), contained(p_2.f_2, p_1.f_2))$$

$good$ 値は 2 つのクローンペアを構成するコード片同士の重複度を表し，ok 値は 2 つのクローンペアを構成するコード片同士の包含度を表す．

そして，2 つのコードクローン検出結果 r から r' へのクローンペアマッピングのアルゴリズムを図 3 に示す．図中の $better$ は以下の条件を満たすときに真となる．

$$better = (good \geq t \wedge good > good_max) \vee (good = good_max \wedge ok > ok_max) \vee (ok \geq t \wedge ok_max < t)$$

提案手法では，$better$ 中の閾値 t として論文 [3] で用いられた値 0.7 を使用している．

これらをもとに，2 つのコードクローン検出結果 r_1 と r_2 のクローンペアマッピングを以下の手順で行う．

（A） r_1 から r_2 へのクローンペアマッピングを行う．

（B） r_2 から r_1 へのクローンペアマッピングを行う．

（C） 手順（A）と手順（B）で得られたクローンペア同士の対応関係の和集合を r_1 と r_2 のクローンペアマッピングとする．

以降，この手順でマッピングされたクローンペアのことをマッチしたクローンペアと呼ぶ．

図 4 を例にマッチしたクローンペアを求める．

まず，手順（A）に従って r_1 から r_2 へのクローンペアをマッピングを行う．ok 値と $good$ 値から，p_1 と q_1 を，p_2 と q_2 をそれぞれマッピングする．

次に，手順（B）に従って r_2 から r_1 へのクローンペアをマッピングを行う．ok 値と $good$ 値から，q_1 に p_1 を，q_2 と p_1 をそれぞれマッピングする．

最後に，手順（C）に従って r_1 と r_2 のクローンペア同士の対応関係の和集合を求める．

以上より，p_1 には q_1 と q_2 が，p_2 には q_2 が，q_1 には p_1 が，q_2 には p_1 がマッチする．

3.4 ステップ 4: 比較結果の提示

最後に比較結果を可視化し，開発者に提示する．

第 2.2.3 節において，既存のコードクローン検出結果の可視化手法としてクローン散布図について述べた．

このクローン散布図をもとに提案手法では，各点をミスマッチ率に応じて彩色して描画することで，複数のコードクローン検出結果の差を可視化する．提案手法で出力される散布図を図 5 に示す．

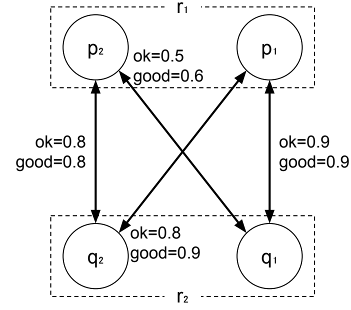

図 4　クローンペアマッピングの例

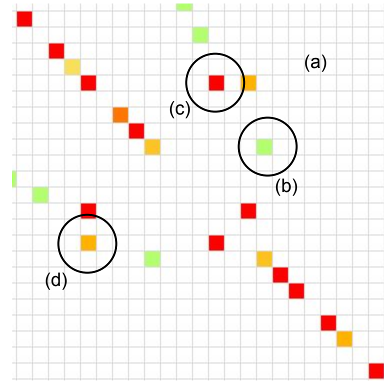

図 5　提案手法の散布図の例

```
( i ) [ > clone pair [0]
        v clone pair [1]
            > Value
            v In <CCFinderX_default> 1 matched     ]
                v clone pair [0]                   ] ( ii )
                    > Value                        ]
            v In <NiCad_default>     1 matched     ]
                v clone pair [0]
(iii)               v Value
                        v Fragment1
                            Path   C:/optional/clearcase/CCMkbl.java
                            Begin  130
                            End    156
                        v Fragment2
                            Path   C:/optional/clearcase/CCCheckin.java
                            Begin  132
                            End    157
```

図 6　マッチしたクローンペアリストの例

ミスマッチ率とは，あるコードクローン検出結果 r を基準として，r と異なるコードクローン検出結果 r' との差を表す値であり，その定義を以下に示す．

ミスマッチ率の定義

r に含まれる，ファイル s_1 と s_2 との間のクローンペアの集合を $P_{all}(r, s_1, s_2)$ とする．そのうち r' に含まれるクローンペアとマッチしているものの集合を $P_{match}(r, r', s_1, s_2)$ とする．

このとき，ミスマッチ率 $m(r, r', s_1, s_2)$ は

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

```
210  String checkMarginsMessage = pageSetupPanel.recalculate();
211  if ( checkMarginsMessage != null )
212  {
213     JOptionPane.showMessageDialog(
                      略
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  }
```

図 7　バグを含んでいる jEdit のコード片

この定義から，提案手法の散布図における，コードクローン検出結果 r を基準としたときの点 (s_1, s_2) における色を以下の手順で決定する．

（1） $|P_{all}(r, s_1, s_2)| = 0$ ならば白色 (図 5 中（a)).

（2） $m(r, r', s_1, s_2) = 0$ ならば緑色（b).

（3） $m(r, r', s_1, s_2) = 1$ のときを赤色として，0 に近づくにつれて黄色へと変化するように色を選ぶ（c, d).

このようにミスマッチ率に応じて彩色することで，開発者は複数のコードクローン検出結果の差を直感的に把握することができる．

また，開発者はどのようなクローンペアがマッチしたかリストで確認することが可能である．提案手法で出力されるマッチしたクローンペアリストの例を図 6 に示す．

リストには，基準となるコードクローン検出結果の現在選択中の点に含まれているクローンペアの一覧が表示される（図 6 中（i)).

また，それぞれの子要素にはマッチしたクローンペアがコードクローン検出結果ごとに表示される（ii)．図の例では，基準に含まれる clone pair [1] に対して，コードクローン検出結果 CCFinderX_default に含まれるクローンペア 1 個と，NiCad_default に含まれるクローンペア 1 個がマッチしていることがわかる．

更に，Value ノードを展開することで，各クローンペアを構成するコード片の範囲を確認することも可能である（iii)．図の例では，CCMkbl.java の 130 行目から 156 行目のコード片と CCCheckin.java の 132 行目から 157 行目のコード片が NiCad_default の clone pair [1] を構成していることがわかる．

4. 適 用 実 験

本章では提案手法を用いた適用実験について述べる．

適用実験では，提案手法を用いることで重要なコードクローンを発見できるかについて調査した．具体的には，オープンソースソフトウェア jEdit において実際に修正されたバグを含んだコード片のコードクローンを複数のコードクローン検出結果を比較することで発見する．

表 1　CCFinderX

Minimum Clone Length: 50
Minimum TKS: 12

表 2　NiCad

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

対象となったコード片を図 7 に示す．このコード片はファイル PrinterDialog.java に含まれていた．以降，このコード片をコード片 A と呼ぶ．

適用実験で用いた jEdit の詳細を以下に示す．

- プログラミング言語: Java
- リビジョン: r24577
- LOC: 113,826 行

また，コードクローン検出結果として CCFinderX と NiCad のそれぞれのデフォルトパラメータで得られたものを用いた．それぞれのデフォルトパラメータの詳細を表 1 と表 2 に示す．

まず，CCFinderX のコードクローン検出結果について調査した．

図 8 に CCFinderX のコードクローン検出結果を基準としたミスマッチ率の散布図の一部を示す．中央黒枠で囲まれた部分が点 (PrinterDialog.java，PrinterDialog.java) である．クローンペアリスト（図 9）を確認すると，この点にはクローンペアが 20 個含まれていて，そのうち NiCad で検出されたクローンペアとマッチしたものは 2 つであった．また，コード片 A を含むクローンペアは CCFinderX で 1 個検出された（図 9 中

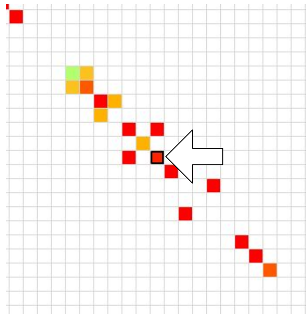

図 8　CCFinderX の検出結果を基準としたミスマッチ率の散布図の一部

```
> clone pair [0]
      ⋮ 略
v clone pair [4]
  v Value
    v Fragment1
        Path    C:/jedit/gjt/sp/jedit/print/PrinterDialog.java
        Begin   210
        End     243
    v Fragment2
        Path    C:/jedit/gjt/sp/jedit/print/PrinterDialog.java
        Begin   260
        End     293
    In <NICAD>  0 matched
> clone pair [5]
      ⋮ 略
> clone pair [19]
```

図 9　CCFinderX の検出結果における点 (PrinterDialog.java, PrinterDialog.java) のクローンペアリスト

青枠部分）が，これとマッチするクローンペアは NiCad では検出されていないことがわかった．

同様に，点 (PrinterDialog.java, PrinterDialog.java) と同じ列，または同じ行にある点についても調査したが，コード片 A を含むクローンペアは存在しなかった．

次に，NiCad のコードクローン検出結果について調査した．

図 10 に NiCad のコードクローン検出結果を基準としたミスマッチ率の散布図の一部を示す．図 8 と同様に，中央黒枠で囲まれた部分が点 (PrinterDialog.java, PrinterDialog.java) である．この点は緑色で表示されていることから，すべて CCFinderX で検出されたクローンペアとマッチしており，コード片 A の新たなコードクローンは含まれていない．

同様に，(PrinterDialog.java, PrinterDialog.java) と同じ列，または同じ行にある点についても調査したが，すべてクローンペアの存在しない点であった．

これらの調査からコード片 A を含むコードクローンは，NiCad のデフォルトパラメータでは検出できないことが明らかとなった．また，CCFinderX で検出されたコードクローンは，実際にリビジョン r24577 から r24578 への更新においてコード片 A と同時に修正されていた．

よって，提案手法により 2 つのコードクローン検出結果と比較することで，重要なコードクローンを見逃すことなくより正確な分析を行うことが可能となった．

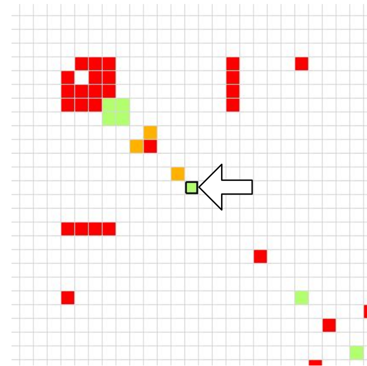

図 10　NiCad の検出結果を基準としたミスマッチ率の散布図の一部

5. まとめと今後の課題

本研究では，複数のコードクローン検出結果間でクローンペア同士の対応付けを行うクローンペアマッピングを定義した．また，ミスマッチ率を定義することでそれらの差を可視化する手法を提案した．

そして，提案手法を実際のソフトウェアに適用し 2 つのコードクローン検出結果を比較したところ，一方だけでは検出できなかった重要なコードクローンを発見できることを確認した．

今後の課題として，他のコードクローン検出結果のクローンペアとマッチしないクローンペアの中から重要なものを優先的に開発者に提示することが挙げられる．

謝辞　本研究は JSPS 科研費 18H04094 の助成を受けた．

文　献

[1] 井上 克郎, 神谷 年洋, 楠本 真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

[2] 肥後 芳樹, 楠本 真二, 井上 克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. TSE, Vol. 31, No. 10, pp. 804–818, 2007.

[4] 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, Aug. 2013.

[5] 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.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. TSE, Vol. 28, No. 1, pp. 654–670, 2002.

[7] 横井 一輝, 崔 恩瀞, 吉田 則裕, 井上 克郎. 情報検索技術に基づく細粒度ブロッククローン検出. コンピュータ ソフトウェア, Vol. 35, No. 4, pp. 16-36, 2018.

[8] Y. Semura, N. Yoshida, E. Choi, and K. Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In Proc. APSEC 2017, pp. 654-659, 2017.

[9] 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.

[10] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code Clone Analysis Tool. In Proc. ISESE2002, Vol. 2, pp. 31-32, 2002.