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.

CRDの類似度に基づくコードクローン追跡手法

堀田　圭佑†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市

あらまし　これまでにソースコードの開発履歴を分析し，コードクローンの追跡を行う手法がいくつか提案されている．コードクローンの追跡はコードクローンに関する様々な研究に応用される重要な基盤技術の一つである．しかし，これまでに提案されている手法にはいくつかの課題点が存在する．そこで本稿では，既存のコードクローン追跡手法であるCRDを用いた手法に改良を加え，これらの課題点を解決したコードクローン追跡手法を提案する．提案手法をツールとして実装し，オープンソースソフトウェアに対して実験を行った．その結果，既存手法では追跡に失敗するが，提案手法を用いることで追跡することができるという事例が多数存在することを確認した．また，新たに追跡できた事例のうち約91%は正しいものであることが確認できた．

キーワード　コードクローン，コードクローンの追跡，ソフトウェア進化，CRD

Clone Tracking based on Similarity of CRD

Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871, Japan

Abstract　Clone tracking is a fundamental technique of research on code clones. There exists some techniques to track clones across version histories, but they still remain some issues. This paper proposes a new technique, which is an enhancement of the existing CRD-based clone tracking. We have developed a tool as an implementation of the proposed method, and conducted an experiment to confirm the improvement of accuracy of clone tracking compared to the existing technique. The experimental results showed a lot of instances newly tracked by the proposed method, and approximately 91% of newly detected tracking are correct.

Key words　Code Clones, Tracking Clones, Software Evolution, CRD

1. まえがき

コピーアンドペーストによるコードの複製は，ソフトウェア開発において習慣的に行われている既存コードの再利用方法である[1]．この再利用方法は極めて容易に実施可能であり，またその実施に際し既存コードに手を加える必要がない．このような利点から，コードの複製は高い頻度で行われている[2]．

コードの複製は頻繁に行われている習慣的な再利用手法であるが，その一方でコードクローンがソースコード中に作り込まれる主たる要因として嫌忌されている．コードクローンとはソースコード中に存在する同一の，あるいは類似するコード片のことをいい，ソフトウェアの保守を困難にする要因の一つとして指摘されている．その理由として，同様の修正を繰り返し行わなければならないことによる保守作業量の増大や，修正漏れに起因する不具合の混入，及びその残存の危険性が挙げられる．このような背景から，コードクローンに対する研究が盛んに行われており，ソースコード中に存在するコードクローンを自動的に検出する手法や，コードクローンを除去する手法，あるいはその混入を防止する手法などが多数提案されている[3]〜[5]．

コードクローンに関する研究は多岐にわたるが，それらの基盤となる技術の一つとしてコードクローンの追跡が挙げられる．コードクローンの追跡とは，あるバージョンのソースコード中に存在するコードクローンが，その次のバージョンのソースコードのどこに存在するのかを特定し，その対応付けを行う技術である．コードクローンの追跡によって，個々のコードクローンがどのような進化を辿っているのかを分析することや，コードクローンに対する修正漏れを検出することが実現可能となる．

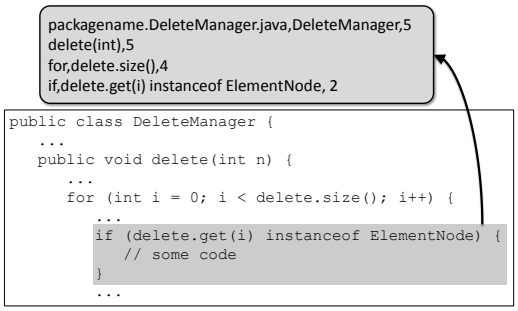
これまでにいくつかのコードクローン追跡手法が提案されている．それらの多くは以下のいずれかの方法を採用している．

コードクローンの対応付けを行う手法：　前後のバージョンに対してコードクローンの検出を行い，バージョン間でコードクローンを比較して対応付けを行う[6], [7].

修正情報を用いる手法：　最初のバージョンに対してのみコードクローン検出を実施し，加えられた修正の位置情報を用いて以

<CRD> ::= <file> <class> <CM> [<method>]
<method> ::= <signature> <block>*
<block> ::= <btype> <anchor> <CM>
<btype> ::= 'for' | 'while' | 'do' | 'if' | 'else' | 'switch'
| 'try' | 'catch' | 'finally' | 'synchronized'

(a) 定義



(b) 例

図 1　CRD の定義と例
Fig. 1　The Definition and An Example of CRD

降のバージョンにおけるコード片の位置情報を特定する [8], [9].

しかし，これらの追跡手法にはそれぞれ，コードクローンに大きな修正が加えられた場合や，コードクローンが追跡対象期間の途中から出現した場合に，適切に追跡ができないという課題点がある．CRD(Clone Region Descriptors) を用いたコードクローン追跡は，これらの課題点を克服した追跡手法である [10]．この手法は，コードクローンを構成するコード片の位置情報を追跡に用いることで，これまでの手法では追跡に失敗するような事例に対しても適切に追跡を行うことを可能にしている．しかしながら，この手法を用いた場合でも，あるメソッドの一部分を別のメソッドとして抽出するなどの，コード片の位置が変わるような修正が加えられた場合にはコードクローンの追跡ができない．

そこで本研究では，これまでに提案されている追跡手法が持つ課題点を解決するために，CRD を用いたコードクローン追跡手法の改良を行う．提案する手法は，コード片の位置情報を基準とした類似度を算出し，高い類似度を持つコード片を対応付けることで，コードクローンの追跡を実現する．これにより，位置情報が変化するような修正が加えられた場合でも，適切に追跡を行うことができる．

提案手法を実装し，オープンソースソフトウェアを対象として既存手法との比較実験を行った．その結果，提案手法を用いることで新たに対応付けが行われたコード片のうち，約 91%が正しい対応付けであったことが確認された．さらに，提案手法を用いた場合でも対応付けが行われなかったコード片を分析したところ，そのうちの約 92%が対応付けを行わなかったことが正しいことが確認できた．

2. 提 案 手 法

2.1 CRD

CRD(Clone Region Descriptors) は Duala-Ekoko と Robillard によって提案された，コードクローンを構成するコード片

```
+ try {
      session = openSession();
      if (command != null) {
         log("cmd : " + command, Project.MSG_INFO);
         executeCommand(command);
      } else {
         try {
            BufferedReader br = new BufferedReader(
              new InputStreamReader(commandResource.getInputStream()));
            …
            FileUtils.close(br);
         } catch (IOException e) {
            throw new BuildException(e);
         }
      }
+ } catch (JSchException e) {
+    …
+ }
```

図 2　CRD が変化する修正
Fig. 2　An Actual Modification that Changes CRD

のおおよその位置情報を表す表現形式である [10]．CRD の定義を図 1(a) に示す．なお CRD は Java を対象に定義されているため，図 1(a) の定義には Java 特有の文が出現している．

CRD は着目するコード片を完全に包含するブロックのうち，最もネストの深いものを基準に定義される (以降本稿では，CRD の算出の基準となるブロックを，そのコード片の**所有ブロック**と呼ぶ)．CRD には所有ブロックとその外側にあるすべてのブロックの情報が含まれている．図 1(b) に CRD の例を示す (注1)．図中の `if` 文を基準に定義される CRD は図の上部に示すような文字列となる．

CRD を用いたコードクローン追跡では，コードクローンに含まれる各コード片について CRD を算出する．その後，次のバージョンに同じ CRD を持つコード片が存在するか否かを調べ，存在すれば二つのコード片を対応付けることでコードクローンの追跡を行う．

しかしこの手法には，CRD が変化するような修正が加えられた場合に追跡に失敗するという課題点がある．図 2 にそのような修正の例を示す．この修正は Ant に実在する修正の例であり，`if` 文の外側に `try` 文が挿入されている．修正前のバージョンにおいて `else` 節全体がコードクローンとなっていた．この修正により，このコードクローンの CRD が変化する．したがって，CRD を用いたコードクローン追跡手法を使用した場合，この修正の前後におけるコードクローンの追跡を適切に行うことができない．本稿では，この例のように CRD に変化が生じるような修正が加えられた場合においても適切に追跡が可能となるように，CRD を用いたコードクローン追跡手法に改良を加える．

2.2 改 良 方 法

CRD を用いた既存のコードクローン追跡手法では，前後のバージョンに存在する二つのコード片の CRD が一致する場合に，二つのコード片を対応付ける [10]．2.1 節で述べたとおり，この方法でコード片の対応付けを行った場合，CRD が変化するような修正が加えられた場合に適切な対応付けを行うことができない．

そこで本稿では，CRD を文字列とみなしたときの類似度を算出し，高い類似度を持つコード片を対応付ける改良方法を提案する．本稿で提案する手法におけるコード片の対応付けの流

(注1)：この例は文献 [10] で用いられている例と同一である．

れは以下の通りである．

手順 1： 前後のバージョンに存在するすべてののコード片について，CRD を算出する．

手順 2： 前後のバージョンにおけるコード片の組み合わせのうち，対応付けの条件を満たすものについて，CRD の類似度を算出する．

手順 3： 手順 2 で算出した CRD の類似度を用い，前後のバージョン間のコード片の対応付けを行う．

以降，手順 2 及び手順 3 について詳細に述べる．

2.3 手順 2：類似度の算出

提案手法は文字列の類似度を用いることで CRD が完全に一致しない場合でもコード片の対応付けを行う．提案手法では，誤った対応付けを行う確率を下げるために，対応付けの候補となるコード片の組に対していくつか制約を設ける．

b_α，b_β を二つのコード片の所有ブロックとする．このとき，b_α，b_β が以下の条件をすべて満たす場合のみ，二つのコード片の対応付けが可能であるとみなす．

制約 A： b_α，b_β の種類が同じである．

制約 B： b_α 及び b_β が条件式を持つブロックである場合，b_α と b_β の条件式が変数名，リテラルの違いを除いて同じである．

制約 C： b_α 及び b_β がメソッドまたはコンストラクタである場合，メソッド名が同じであるか，引数の型及び名前がすべて同じであるかのいずれかを満たす．

これらの制約をすべて満たすコード片の組に対し，CRD の文字列を基にした類似度を算出する．類似度の算出には，文字列の類似度を算出するあらゆる手法が適用可能である．

2.4 手順 3：対応付けの導出

手順 2 で算出した類似度を用い，コード片の対応付けを行う．このとき，互いに現在対応付けられているコード片よりも類似度の高い組が存在しないような対応付けを求める．

B_{before}，B_{after} をそれぞれ修正前，修正後のバージョンに存在する所有ブロックの集合であるとし，$sim(b_{before}, b_{after})$ を二つの所有ブロック b_{before}，b_{after} の CRD から導出された類似度であるとする．このとき，対応付けられているコード片の組 $p = (b_{before}, b_{after})$ すべてについて，以下の式 (1) が成り立つような対応付けを求める．

$$\forall b'_{before}, \forall b'_{after}$$
$$[sim(b_{before}, b'_{after}) \leqq sim(b_{before}, b_{after})$$
$$\vee\ sim(b'_{before}, b_{after}) \leqq sim(b_{before}, b_{after})] \quad (1)$$

3. 実　　装

提案手法を組み込んだコードクローン追跡ツールを Java を用いて開発した．現在のところ，このツールは Java で記述されており，かつバージョン管理システム Subversion を用いて管理されているソフトウェアのみを対象としている．

本実装はコード片の単位としてブロックを採用している．その理由は以下の通りである．

```
boolean validate(int a, int b) {
   if (a == 0) {
      if (b == 0) {
         return false;
      }
   }
   return true;
}
```

(a) ソースコード

```
boolean validate(int a, int b) {
   IF (a == 0)
   return true;
}
```

```
if (a == 0) {
   IF (b == 0)
}
```

```
if (b == 0) {
   return false;
}
```

(b) 置き換え後のブロック

図 3　サブブロックの置き換え
Fig. 3　An Example of Replacing Sub-blocks

- CRD はブロックを基準に算出されており，CRD を用いたコードクローン追跡手法は実質的にはブロックの対応関係を特定しているため．
- バージョン管理システムによって管理されている開発履歴の各リビジョンに対してコードクローン検出を行う必要があり，長い開発期間を有するソフトウェアに対して適用可能とするためには高速なコードクローン検出手法が必要となるため．

このツールの処理の流れは以下の通りである．

STEP1： ブロックの特定．

STEP2： ブロック単位でのコードクローン検出．

STEP3： コードクローンの追跡．

以降，それぞれの STEP について述べる．

3.1 STEP1：ブロックの特定

この STEP では，各リビジョンのソースファイルを取得し，それぞれに対して字句解析並びに構文解析を行うことで，ブロックの特定を行う．また，それぞれのブロックに対する CRD の算出も併せて行う．特定したブロックの情報はデータベースに格納する．

なお，この STEP はブロックの長さに閾値を設けており，閾値を超えないブロックはデータベースに格納しない．これは，後のコードクローン検出における誤検出の低減と検出速度の高速化を目的とした処理である．

3.2 STEP2：コードクローンの検出

この STEP ではそれぞれのリビジョンに対して，ブロック単位でのコードクローン検出を行う．検出の単位をブロックとすることで，行やトークンなどのより細かい粒度で検出を行う手法と比較して，高速なコードクローンの検出が可能となる．

ブロックの比較にはブロックの文字列から算出したハッシュ値を用いる．ハッシュ値の算出の前に，それぞれのブロックの文字列に対して以下の正規化を行う．

- 変数名，リテラルを特殊文字に置き換える．
- サブブロックを特殊な文字列に置き換える．

サブブロックの置き換えに用いる文字列には，ブロックの種

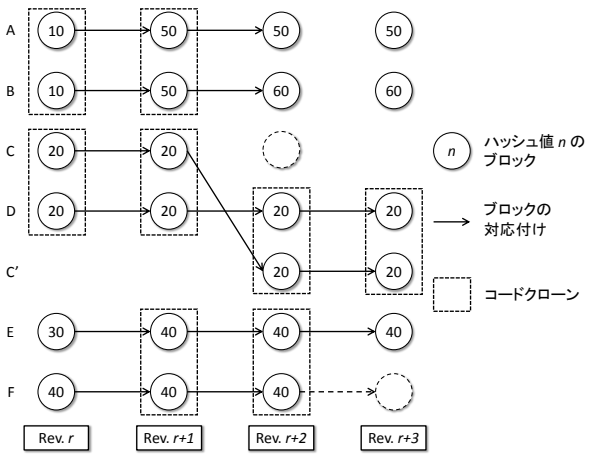


図 4　提案手法によるコードクローン追跡の例
Fig. 4　An Example of Clone Tracking with the Proposed Method

類や条件式などの情報が含まれている．図 3 にサブブロックの置き換えの例を示す．このような正規化により，ブロック内で行われている処理の流れは同じであるがサブブロックで行われている処理が異なるような場合でも，それらのブロックをコードクローンとして検出することができる．

正規化後の文字列からハッシュ値を算出し，比較することでコードクローンの検出を行う．同じハッシュ値を持つブロックの集合が一つのコードクローンとして検出される．

3.3　STEP3：コードクローンの追跡

この STEP では，2. 章で述べた手法を用いてコード片の対応付けを行い，コードクローンの追跡を行う．なお，本実装では CRD の類似度の算出にはレーベンシュタイン距離 [11] を用いた．レーベンシュタイン距離は一方の文字列をもう一方の文字列に変換するために必要な編集距離を表しており，その値が小さいほど文字列間の類似度が高いことを示している．

図 4 にコードクローン追跡の例を示す．この例では，リビジョン $r+1$ から $r+2$ への遷移の際に，コード片 C が別の位置に移動され，コード片 C' となっている．既存手法を用いた場合は，コード片 C に対応するコード片がリビジョン $r+2$ に存在しないと判断されるが，提案手法を用いることでコード片 C とコード片 C' の対応付けを特定することができる．また，リビジョン $r+2$ に存在するコード片 F は，提案手法を用いた場合でもリビジョン $r+3$ に対応するコード片が発見されなかったため，いずれのコード片とも対応付けられていない．

なお，リビジョン k と $k+1$ の間のコード片の対応付けを行う際，提案手法はリビジョン k においていずれかのコードクローンに含まれるコード片のみを対応付けの対象とする．したがって，図 4 中のリビジョン $r+2$ と $r+3$ の対応付けを行う際に，二つのコード片 A，B は対応付けの対象とならない．

4.　評　　価

4.1　準　　備

提案手法を実装したツールを用い，既存の CRD を用いたコードクローン追跡手法との比較実験を行った．

この実験における評価項目は以下の二点である．

項目 1：　既存手法では特定できない適切な対応付けを，提案手法を用いることで特定することができるか．

項目 2：　提案手法を用いた場合でも対応付けが行われなかったコード片は，対応付けられなかったことが適切であるか．

比較を行うために，文献 [10] の記述に従って，著者らが既存手法の実装を行った．なおこの実装では，文献 [10] で述べられている工夫点はすべて実装されている．

実験対象のソフトウェアを表 1 に，ツールの実行に要した時間を表 2 にそれぞれ示す．これらのソフトウェアを選定した基準は以下の通りである．

- Java を用いて記述されており，かつ Subversion を用いて管理されていること．
- 長い開発期間を有し，かつ広く用いられているソフトウェアであること．

なお今回の実験では，ArgoUML については “*/trunk/src*” 以下を，Ant については “*/ant/core/trunk/src/main*” 以下を，それぞれ調査対象としている．これは，テストケースなどを分析対象から除外することを目的とした措置である．

また，今回の実験では検出するブロックの最小トークン数を 30 としている．

4.2　実験結果 (項目 1)

提案手法を用いたコードクローンの追跡結果と，既存手法を用いたコードクローンの追跡結果を比較した．表 3 に，それぞれの手法について，後のリビジョンに存在するコード片と対応付けが行われなかったコード片の数を示す．本実験では,『既存手法を用いた場合は対応付けができたが，提案手法を用いた場合には対応付けができなかった』という事例は存在しなかった．すなわち，提案手法を用いることで，既存手法では特定できなかった対応付けが，ArgoUML については 202，Ant については 236，それぞれ新たに特定された．

新たに特定された対応付けが適切なものであるか否かを判別するため，著者らのうち二名が目視による調査を行った．すべての対応付けを確認することは現実的ではないため，ArgoUML についてはバージョン “0.32” の開発期間 (リビジョン 18,179〜リビジョン 18,964)，Ant についてはバージョン “1.8” の開発期間 (リビジョン 554,389〜リビジョン 904,537) に，それぞれ

表 2　実 行 時 間
Table 2　Elapsed Time

ソフトウェア	STEP1 & 2 [m]	STEP3 [m]	計 [m]
ArgoUML	132	43	175
Ant	100	50	150

表 3　対応付けされなかったコード片の数
Table 3　The Number of Code Fragments Not Tracked by Each Method

ソフトウェア	提案手法	既存手法	Δ
ArgoUML	537	739	202
Ant	345	581	236

表 1　実験対象ソフトウェア
Table 1　Target Software Systems

名称	開始リビジョン (日付)	終了リビジョン (日付)	調査対象リビジョン数	LOC(終了リビジョン)
ArgoUML	15,880 (2008-10-04)	19,794 (2011-11-17)	2,222	362,604
Ant	268,587 (2001-02-05)	904,537 (2010-01-30)	5,143	211,855

対象期間を限定して調査した．

表 4 に調査結果を示す．表中の “**T**” は対応付けが適切であったことを，“**F**” は対応付けが不適切であったことをそれぞれ表している．また，各項目の意味はそれぞれ以下の通りである．

T1　コード片，もしくはコード片を包含するコードが新たなメソッドとして抽出された．

T2　コード片の外側に新たなブロックが挿入された．

T3　コード片の外側にあるブロックのうち少なくとも一つについて，その条件式が変化した．

T4　コード片を含むメソッドが別のクラスに移動した．

T5　コード片を含んでいる `try` 文に新たな `catch` 節が追加された．

T6　コード片を含むメソッドが別のメソッドの中に展開された．

F1　コード片が削除されたにも関わらず，別のコード片と誤って対応付けられた．

F2　コード片に修正が加えられた結果，コード片の大きさが閾値を下回ったため，別のコード片と対応付けられた．

表 4 より，提案手法によって新たに特定された対応付けのうち約 91%が適切なものであることがわかる．

図 5 に新たに特定された対応付けの例を示す．この例は Ant で発見された，T1 に分類される対応付けである．図中のハイライトされている二つの `if` 文が新たに対応付けられたブロックである．この例では，修正前は同じメソッドに含まれていたブロック文が，修正によってそれぞれ異なるメソッドに抽出されている．

4.3　実験結果 (項目 2)

提案手法を用いた場合でも，後のリビジョンに対応するコード片が存在しなかったコード片について，対応するものが見つからなかったことが適切であるか否かを調査した．この調査も項目 1 に対する調査と同様に，著者らのうち二名による目視によるものとなっている．調査対象期間も項目 1 に対するものと同一であり，ArgoUML についてはバージョン “0.32” の開発期間，Ant についてはバージョン “1.8” の開発期間をそれぞれ対象としている．

表 4　新たに特定された対応付けに対する目視調査の結果
Table 4　Manual Investigation on Newly Detected Pairs of Code Fragments

		ArgoUML	Ant
T	適切	3	37
	T1　新たなメソッドとして抽出	0	18
	T2　ブロックのネストが変化	0	9
	T3　条件式が変化	2	5
	T4　メソッドの移動	1	2
	T5　`catch` 節の追加	0	2
	T6　メソッドのインライン展開	0	1
F	不適切	0	4
	F1　ブロックの削除	0	3
	F2　ブロックの縮小	0	1

```
public synchronized void createStreams() {
  if (out != null && out.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
  }

  …

  if (error != null && error.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
  } else if (!(logError || ourputStream == null)) {

  …

}
```

(a) 修正前 (Rev. 567,592)

```
public synchronized void createStreams() {
  outStreams();
  errorStreams();
  …
}

private void outStreams() {
  if (out != null && out.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
  }
  …
}

…

private void errorSteams() {
  if (error != null && error.length > 0) {
    String logHead = new StringBuffer("Output ").append(
        ((append) ? "appended" : "redirected")).append(
        " to ").toString();
    errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
  } else if (!(logError || ourputStream == null)) {
  …
}
```

(b) 修正後 (Rev. 567,593)

図 5　提案手法により新たに対応付けられたコード片の例
Fig. 5　Newly Tracked Code Fragments

表 5 に調査結果を示す．表中の各項目の意味はそれぞれ以下の通りである．

T1　コード片が小さくなり，対応するブロックがデータベースに格納されなかった．

T2　コード片が削除された．

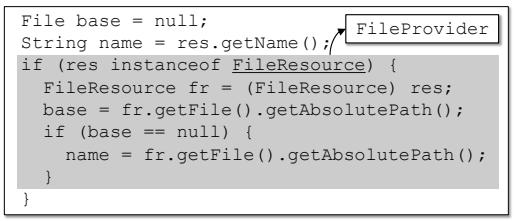
T3　コード片が大幅に変化し，別のものとなった．

F1　条件式に出現する型が変化したため，対応付けられなかった．

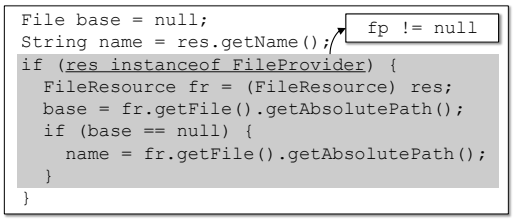
F2　条件式が変化したため，対応付けられなかった．

F3　`catch` 節が追加されたため，対応付けられなかった．

表 5 より，提案手法が対応付けを行わなかったもののうち約



(a) F1 (Rev. 688,723 - Rev. 688,724)



(b) F2 (Rev. 718,386 - Rev. 718,387)

図 6　対応付けに失敗した修正の例
Fig. 6　An Example of Failure of Tracking

92%が適切であることがわかる．

対応付けに失敗したものは，すべて制約 2 を満足させない修正に起因するものであった．図 6 にそのような修正の例を示す．図 6 の例は共に Ant において発見された修正の例である．図 6(a) は F1 に分類される修正であり，条件式中に出現する型が変化したため対応付けが行われなかった．一方，図 6(b) は F2 に分類される修正であり，条件式そのものが変化したため対応付けが行われなかった．

5. 結果の妥当性について

調 査 手 法

本稿で述べた実験は，いずれも著者らによる目視によって行われた．しかし，著者らは本実験で対象としたソフトウェアの開発者ではないため，判断に誤りが混入している可能性がある．なお，今回の実験では，誤りが混入する確率を下げるため，著者らのうち二名が同時に調査を行っている．

実 験 対 象

今回の実験は，二つのオープンソースソフトウェアをその対象としている．しかし，これらはいずれも Java で記述されたソフトウェアであるため，この実験の一般性を向上させるためには，他の言語で記述されたソフトウェアに対して実験を行う必要があるといえる．

表 5　対応付けが行われなかったコード片に対する目視調査の結果
Table 5　Manual Investigation on Code Fragments not Tracked with the Proposed Method

		ArgoUML	Ant
T	適切	23	33
	T1 ブロックの縮小	7	9
	T2 ブロックの削除	8	24
	T3 ブロックの大幅な変化	8	0
F	不適切	5	0
	F1 条件式中の型の変化	2	0
	F2 条件式の変化	2	0
	F3 `catch` 節の追加	1	0

6. あ と が き

本稿では，既存のコードクローン追跡手法である CRD を用いた手法に改良を加えた，新たなコードクローン追跡手法を提案した．提案する手法をツールとして実装し，二つのオープンソースソフトウェアに対して適用実験を行った．その結果，既存手法を用いた場合は追跡に失敗するが，提案手法を用いることで追跡できた事例を多数発見した．また，新たに追跡できた事例についてその正しさを調査したところ，約 91%の事例について追跡が正しいものであったことが確認できた．

本研究の今後の課題として，追跡精度のさらなる改善と，Java 以外で記述されたソフトウェアや商用ソフトウェアを対象とした実験などが挙げられる．

謝辞　本研究は，日本学術振興会科学研究費補助金萌芽研究 (課題番号 : 24650011)，若手研究 (A)(課題番号 : 24680002)，及び特別研究員奨励費 (課題番号 : 25・1382) の助成を得て行われた．

文　　献

[1] M. Kim, L. Bergman, T. Lau, and D. Notokin, “An Ethnographic Study of Copy and Paste Programming Practices in OOPL,” Proceedings of the 3rd International Symposium on Empirical Software Engineering, pp.83–92, Aug. 2004.
[2] G. Zhang, X. Peng, and Z.X.W. Zhao, “Cloning Practices: Why Developers Clone and What can be Changed,” Proceedings of the 28th International Conference on Software Maintenance, pp.285–294, Sep. 2012.
[3] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” Information and Software Technology, vol.55, no.7, pp.1165–1199, July 2013.
[4] 神谷年洋，肥後芳樹，吉田則裕，“コードクローン検出技術の展開,” コンピュータソフトウェア，vol.28，no.3，pp.28–42，2011.
[5] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング,” コンピュータソフトウェア，vol.28，no.4，pp.42–56，Nov. 2011.
[6] T. Bakota, R. Ferenc, and T. Gyim’othy, “Clone Smells in Software Evolution,” Proceedings of the 23rd International Conference on Software Maintenance, pp.24–33, Oct. 2007.
[7] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy, “An Empirical Study of Code Clone Genealogies,” Proceedings of the 13th International Symposium on Foundations of Software Engineering, pp.187–196, Sep. 2005.
[8] S. Thummalapenta, L. Cerulo, L. Aversano, and M.D. Penta, “An empirical study on the maintenance of source code clones,” Empirical Software Engineering, vol.15, no.1, pp.1–34, Feb. 2010.
[9] N. Göde and R. Koschke, “Frequency and Risks of Changes to Clones,” Proceedings of the 33rd International Conference on Software Engineering, pp.311–320, May 2011.
[10] E. Duala-Ekoko and M.P. Robillard, “Clone Region Descriptors: Representing and Tracking Duplication in Source Code,” ACM Transactions on Software Engineering and Methodology, vol.20, no.1, pp.3:1–3:31, June 2010.
[11] V.I. Levenshtein, “Binary Codes Capable of Correcting Deletiong, Insertions, and Reversals,” Soviet Physics Doklady, vol.10, no.8, pp.707–710, 1966.