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.

オープンソースソフトウェアにおける コードクローンの消失に関する調査

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
E-mail: †{k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェアの保守を困難にするおそれのある要因の一つとして，コードクローンの存在が指摘されている．その一方で，コードクローン生成の主要因である既存コードのコピーアンドペーストによる再利用は，必要な機能を高速に実装できるという理由から，ソフトウェア開発においてしばしば行われている．そのため，ソフトウェア中にコードクローンを存在させないことは現実的ではない．すなわち，コードクローンを管理する技術が求められているといえる．コードクローンの管理を効率的に行うためには，コードクローンが持つ特性を明らかにする必要がある．このような背景に基づき，コードクローンの発生，並びに進化に関する研究が盛んに行われている．しかしながら，コードクローンの消失に関する研究はこれまでに行われていない．コードクローンの消失理由，及び消失したコードクローンが持つ特性が明らかになれば，負の影響を持つコードクローンの特定などの促進が期待される．そこで本稿では，高速にコードクローンの消失を特定する手法を提案し，それを用いてオープンソースソフトウェアを対象とした調査を行った．調査の結果，コードクローンの消失はソフトウェアの開発期間全体を通して多数発生していることや，消失したコードクローンはそうでないコードクローンと比較して複雑でなく，かつコードクローンを構成するコード片の数が少ない傾向にあることがわかった．

キーワード　コードクローン，ソフトウェア進化，ソフトウェア保守，オープンソースソフトウェア

An Empirical Study of Clone Disappearances on Open Source Software Projects

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

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871, Japan
E-mail: †{k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Code clones have been well studied because the presence of clones is regarded as a bad smell for software maintenance. On the other hand, creating code clones has a positive aspect that reusing existing code by copy-and-paste operations can realize rapid development of software systems. Thus, it is not realistic to rid software systems of clones. That is to say, an efficient clone management is required. Based on this background, many researchers have studied on clone appearances or clone evolution. However, there is no research that reveals how clones were gone. To reveal why clones were gone or what characteristics disappeared clones have could promote the efficient clone management. This paper proposes an investigation method for clone disappearances, and conducted an empirical study on open source software systems. Our experimental results showed that clone disappearances occur many times. Moreover, it was found that disappeared clones tend to consist of less code fragments and be less complex than non-disappeared ones.

Key words　Code clone, Software evolution, Software maintenance, Open source software projects

1. ま え が き

ソフトウェアの保守を困難にするおそれのある要因の一つとして，コードクローンの存在が指摘されている．コードクローンとはソースコード中の互いに類似するコード片のことをいい，主にコピーアンドペーストによる既存コードの再利用がその発

生要因であるといわれている．あるコード片に対して不具合の除去や機能の追加等の理由により修正が必要となった場合，そのコード片とコードクローン関係にある他のコード片についても同様の修正を検討する必要がある．しかし，開発者がコードクローンとなっているコード片を把握できていない場合，修正漏れの発生につながる危険性がある．このような背景から，これまでにコードクローンを自動的に検出する技術，あるいはコードクローンの除去を支援する技術に関する研究が盛んに行われている [1], [2].

一方で，コピーアンドペーストによる既存コードの再利用には必要な機能を高速に実現可能であるという利点がある．このため，ソフトウェアの開発現場においてコピーアンドペーストによる再利用は頻繁に行われており，大規模なソフトウェアにおけるソースコードのうち 10%〜15%がコードクローンであるという報告もされている [3]．開発に使用できる資源が限られているにも関わらず，大規模化の一途を辿っているソフトウェア開発の現状を考えれば，コピーアンドペーストによる再利用を禁止してコードクローンを生成させないことは現実的に難しいといえる．したがって，これまでにコードクローンの管理を支援するツールが提案されている [4].

より効率的なコードクローンの管理を目指す上で，コードクローンが持つ特性を明らかにすることは非常に重要である．この考えに基づき，これまでにコードクローンの発生，並びに進化に関する調査が行われている [5], [6]．しかしながら，“コードクローンの消失” に着目した調査はこれまでに行われていない．コードクローンの消失理由，及び消失したコードクローンが持つ特性を調査することで，コードクローンがソフトウェアの保守にどの程度悪影響を与えているのか，また悪影響を与えているコードクローンがどのような特徴を持つのかが明らかになることにつながると期待される．

本稿では開発履歴情報をもとにコードクローンの発生から消失までを追跡するツール **CloneDisapperanceFinder**(以降, **C22R**) を開発するとともに，オープンソースソフトウェアを対象にコードクローンの消失に関する調査を行う．本稿で調査する内容は以下の通りである．

RQ1： コードクローンの消失はどの程度発生しているのか？
RQ2： コードクローンの消失理由は何か？
RQ3： 消失したコードクローンが持つ特徴は何か？

調査の結果，コードクローンの消失は 8〜13 リビジョンの間に一度の頻度で発生していることがわかった．また，修正漏れなどの意図的でない一貫性のない修正によるコードクローンの消失が少なからず起きていることがわかった．さらに，消失したコードクローンはそうでないコードクローンと比較してコードクローンを構成するコード片の数が少なく，また複雑度が低い傾向にあることが確認された．

2. 調 査 手 法

本節ではコードクローンの消失を特定するツールである C22R について述べる．

```
<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

```
public class DeleteManager {
  …
  public void delete (int n) {
    …
    for(int i = n ; i < delete.size() ; i++ ) {
      …
      if (delete.get(i) instanceof ElementNode) {
        // some code
      }
    …
```

A

(b) コード例

```
packagename.DeleteManager.java,DeleteManager,5
delete(int),5
for,delete.size(),4
if,delete.get(i) instanceof ElementNode,2
```

(c) A の `if` 文に対する CRD

図 1 Clone Region Descriptor
Fig. 1 Clone Region Desciptor

2.1 入 出 力

C22R は解析対象となるソフトウェアのソースコードリポジトリを入力として必要とする．与えられたリポジトリを解析し，各リビジョンにおいて消失したコードクローンを検出する．

2.2 制 限

現在のところ，C22R は Java で記述されたソースコードのみを解析することができる．また，バージョン管理システム Subversion で管理されているソフトウェアのみを対象としている．しかし，これらの対象言語並びにバージョン管理システムを拡張することは容易である．

2.3 特 徴

本調査では，リポジトリの各リビジョンにおけるソースコードを逐次解析する必要がある．したがって，C22R に求められる大きな要件の一つとして，解析の高速さが挙げられる．C22R は高速にコードクローンの消失を特定するために，以下に挙げるアプローチを採用している．

ブロック単位のコードクローン検出： C22R は各リビジョンから高速にコードクローンを検出するために，ブロック単位でのコードクローンの検出を行っている．これにより，多くのコードクローン検出手法で用いられている行単位やトークン単位での検出を行うよりも大幅に小さい計算量でコードクローンを検出することができる．

増分的なソースコード解析： C22R は各リビジョンのソースコードを解析する際，前のリビジョンから変更の無いファイルについては前のリビジョンを解析した際に得られたデータを再利用し，変更のあったファイルのみを新たに解析する．これにより，冗長なソースコード解析を削減することができる．

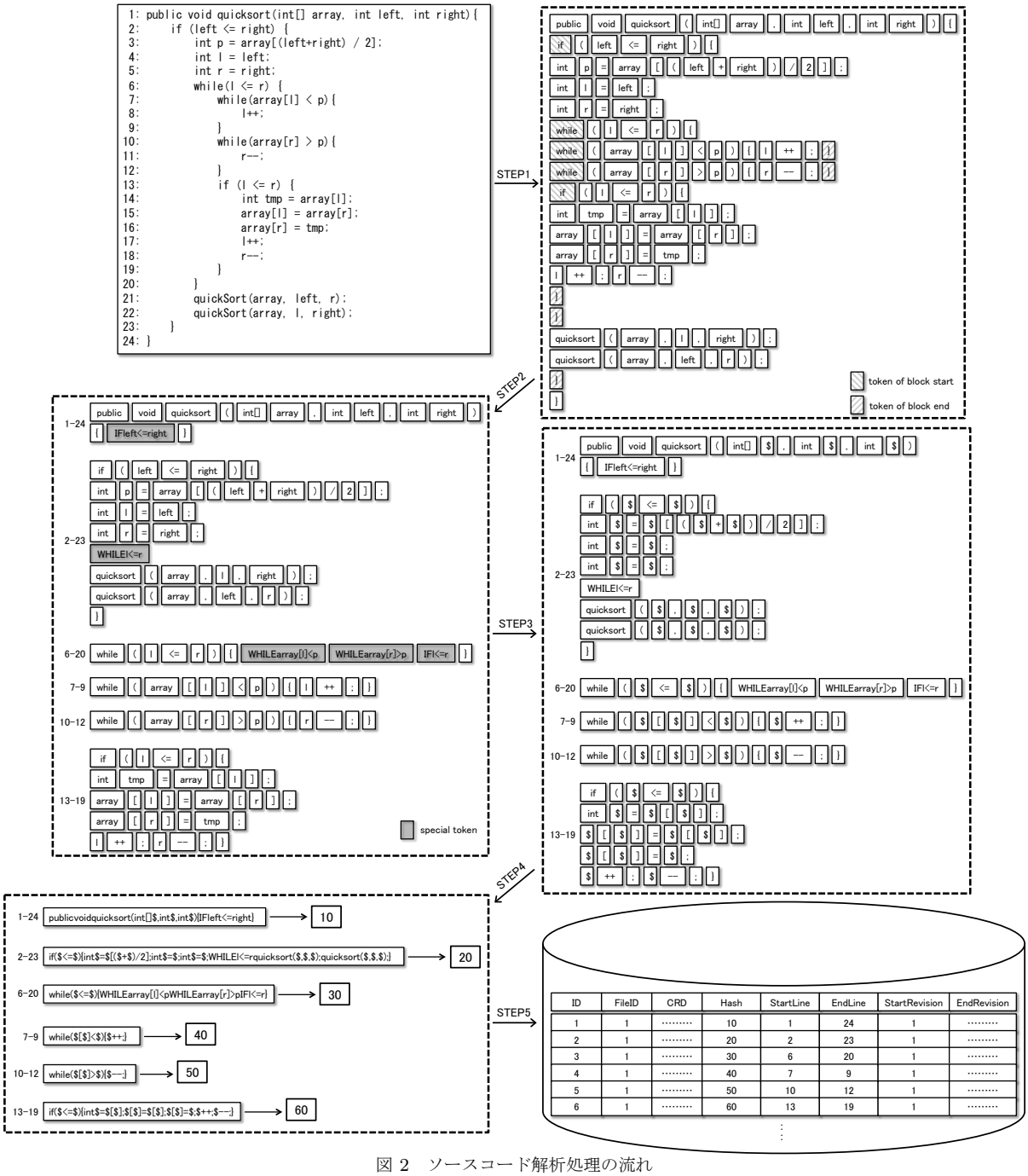


図 2　ソースコード解析処理の流れ

Fig. 2　Intuitive example how hash values are measured from source code

2.4　処 理 内 容

C22R は以下の二つの処理を行う．

ハッシュ値の生成　各リビジョンのソースコードを解析して，すべてのブロックを抽出し，それぞれのブロックに対してハッシュ値を計算する．ハッシュ値は各ブロックのソースコードをもとに算出され，同じハッシュ値を持つブロック同士が互いにコードクローン関係にあるとみなされる．さらに，それぞれのブロックについて CRD [7] と呼ばれる文字列を算出する．この CRD はリビジョン間でブロックを追跡するために使用する．各リビジョンのソースコードを解析した後，これらの情報をデータベースに格納する．

消失の特定　二つの連続するリビジョンについて，CRD を用いてリビジョン間でブロックの対応付けを行う．また，変更前のリビジョンについて同じハッシュ値を持つブロックをグループ化することでコードクローンを特定する．その後，これらの情報を用いてコードクローンの消失を特定する．

本小節では，はじめに CRD について述べた後，それぞれの処理についてその詳細を述べる．

2.4.1　Clone Region Descriptor

Clone Region Descriptor(**CRD**) とは，Duala-Ekoko らによって提案された，コードクローンのおおよその位置情報を表す表現形式である [7]．図 1 に CRD の定義，及び例を示す．CRD はクラス，メソッド，あるいはブロックごとに定義され，ファイルパス，クラス名，メトリクス値，条件式の内容などを

表 1　調査対象ソフトウェア
Table 1　Overview of Target Software

Software	Start revision (date)	End revision (date)	# of target revisions	LOC of start revision	LOC of end revision
ArgoUML	15,880 (2008-10-04)	19,794 (2011-11-17)	2,222	329,170	362,604
Ant	268,587 (2001-02-05)	904,537 (2010-01-30)	5,143	57,124	211,855

もとに算出される．例えば，図 1(b) 中の A の `if` 文に対して CRD を算出すると，図 1(c) のようになる．C22R は文献 [7] に記述されている方法に従い，リビジョン r のあるブロックとリビジョン $r+1$ のあるブロックが同じ CRD を持つ場合，それらが同一のブロックであると判定する．

2.4.2　ハッシュ値の生成

この処理では，与えられたリポジトリから各リビジョンのソースコードを取得し，そこからすべてのブロックの抽出を行うとともに，ハッシュ値及び CRD の算出を行う．各リビジョンにおけるソースコードの解析処理は以下の 5 つのステップから成る (図 2)．なお，前リビジョンから変更のなかったソースファイルについては解析を行わず，すでにデータベースに登録されている情報を再利用することで処理の高速化を目指している．

STEP1(構文解析)：　ソースコードに対して構文解析を行い，ブロックを特定する．このとき，C22R は抽出するブロックが持つトークン数に閾値を設けており，閾値以下のトークン数から成るブロックは解析の対象から除外される．C22R は “30” を規定値としている．

STEP2(サブブロックの特殊文字列への置換)：　それぞれのブロックについて，そのブロック中に別のブロックが含まれる場合，そのサブブロックを特殊な文字列に置換する．この処理を行うことで，ネストの深い位置にあるブロックに対する修正により，その外側にあるコードクローンのすべてが消失として検出されることを防ぐことができる．置換に用いる特殊な文字列は以下の情報を保持している．

- ブロックの種類 (e.g. `if`，`while`，`for`)
- 条件式 (サブブロックが条件式を持つ場合のみ)

STEP3(正規化)：　変数名及びリテラルを特殊文字列に置換する．

STEP4(ハッシュ値及び CRD 算出)：　各ブロックのトークン列からそのブロックを表す文字列を生成し，その文字列をもとにハッシュ値を算出する．このとき，同時に CRD の算出も行う．

STEP5(データベースへの格納)：　算出したハッシュ値及び CRD をデータベースに格納する．

2.4.3　消失の特定

この処理では，“ハッシュ値の生成” によって作成されたデータベースを用いて，コードクローンの消失を特定する．

はじめに，各リビジョンについて，各ブロックをハッシュ値をもとにグループ化する．このとき，同じグループに属するブロックは互いにコードクローン関係にあるとみなされる．また，それぞれのブロックについて，CRD を用いてブロックの追跡を行い，次のリビジョンのどのブロックと対応するのかを特定する．これらの情報を用い，各リビジョンのすべてのコードクローンについて，要素の消失があったかどうかを調査する．C22R は，少なくとも一つ以上の要素の消失を含むコードクローンを “消失したコードクローン” であると判断する

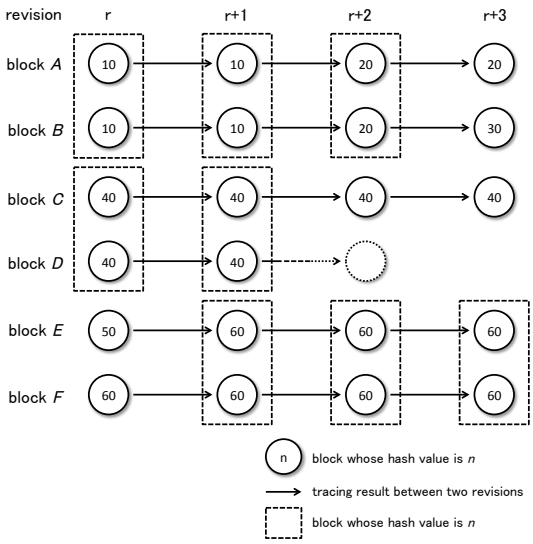


図 3　コードクローンの消失の特定例
Fig. 3　Example of clone disappearances

図 3 にコードクローンの消失の特定例を示す．図中の円は一つのブロックを表しており，円中の数値はそのブロックのハッシュ値を示している．また，矢印は連続するリビジョン間におけるブロックの追跡結果を表している．さらに，図中の長方形は同一のハッシュ値を持つブロックのグループを示している．

この例では，リビジョン r から $r+2$ の間ではブロック A とブロック B がコードクローン関係にあるが，$r+2$ から $r+3$ への遷移の際にブロック B のみに修正が加えられたため，$r+3$ ではこれらのブロックはコードクローン関係ではなくなっている．したがって，このコードクローンは $r+2$ において消失したと判定される．また，ブロック C とブロック D はリビジョン $r+1$ においてコードクローン関係にあるが，リビジョン $r+2$ への遷移の際にブロック D が削除され追跡不能となったため，このコードクローンはリビジョン $r+1$ において消失したと判定される．

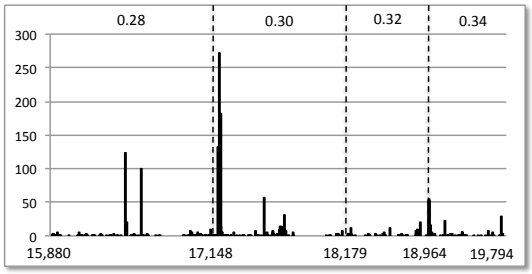
3.　調 査 結 果

C22R を用いて，オープンソースソフトウェア ArgoUML 及び Ant を対象とした調査を行った．表 1 に調査対象ソフトウェアの概要を示す．

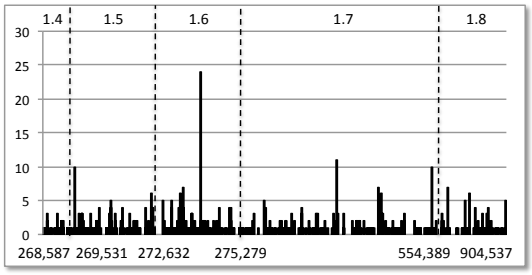
本実験で調査する項目は以下の通りである．

RQ1：　コードクローンの消失はどの程度発生しているのか？

RQ2：　コードクローンの消失理由は何か？



(a) ArgoUML



(b) Ant

図 4　コードクローンの消失数

Fig. 4　Number of disappeared clone groups in every revision

RQ3：　消失したコードクローンが持つ特徴は何か？

以降本節では，C22R の実行時間について触れた後，それぞれの RQ について調査結果を述べる．

3.1　実 行 時 間

表 2 に C22R の実行時間を示す．“消失の特定” については，二つの連続するリビジョンのペアごとに独立して処理が行われるため，それらの最小，最大，及び平均の所要時間を示している．表より，ArgoUML については約 104 分，Ant については約 135 分ですべての処理が完了していることがわかる．この実験結果から，C22R は十分に実用的な時間内に解析を完了することができたものと考えている．

3.2　RQ1 に対する回答

図 4 にそれぞれのプロジェクトにおけるコードクローンの消失数を示す．消失したコードクローンの総数は，ArgoUML では 1,765，Ant については 651 であった．また，少なくとも一つ以上のコードクローンの消失を含むリビジョンの数は，ArgoUML では 263，Ant では 380 となった．すなわち，ArgoUML については 8 リビジョンに一度，Ant については 13 リビジョンに一度の頻度でコードクローンの消失が発生しているという結果となった．

3.3　RQ2 に対する回答

コードクローンの消失理由を明らかにするために，消失したコードクローンを目視で調査した．消失したコードクローンのすべてを調査することは現実的に困難であるため，ArgoUML は図 4 中の “0.32” の期間に，Ant については図 4 中の “1.8” の期間に限定して調査を行った．

表 2　実行時間 (スレッド数 : 8)

Table 2　Timing information (execution with 8 threads)

Software	Hash generation	Disappearance identification			
		total	max.	min.	ave.
ArgoUML	79 mins.	25 mins.	10.5 secs.	1.8 secs.	5.4 secs.
Ant	87 mins.	48 mins.	89.0 secs.	0.5 secs.	4.5 secs.

```
…        …
265      Hashtable ret = new Hashtable();
266      for (Enumeration e = System.getProperties().propertyNames();
267          e.hasMoreElements();) {
268          String name = (String) e.nextElement();
269          ret.put(name, System.getProperties().getProperty(name));
270      }
271      return ret;
…        …
```

(a) org/apache/tools/ant/types/PropertySet.java, revision 669,848

```
…        …
139      sys = System.getProperties();
140      Properties p = new Properties();
141      for (Enumeration e = sys.propertyNames(); e.hasMoreElements();) {
142          String name = (String) e.nextElement();
143          p.put(name, sys.getProperty(name));
144      }
145      p.putAll(mergePropertySets());
…        …
```

revision 671,018

```
String value = sys.getProperty(name);
if (name != null && value != null) {
    p.put(name, value);
}
```

(b) org/apache/tools/ant/types/CommandLineJava.java, revision 669,848

図 5　意図的でない一貫性のない修正の例

Fig. 5　Actual code where an unintended inconsistency occured

調査結果を表 3 に示す．“Refactoring” はソースコードにリファクタリングが適用されたことによって消失したコードクローンの数を示しており，括弧内の数値はそのうちコードクローンの除去を目的としたリファクタリングによって消失したコードクローンの数を示している．“Different evolution” はコードクローンを構成する要素のいくつかのみに対して修正が加えられ，コードクローン関係にあったコード片同士がそれぞれ独自の進化を辿ったものを指す．“Unintended inconsistency” は修正漏れなどの意図的でない一貫性のない修正によって消失したコードクローンの数を表している．“Unneeded code deletion” は不要となったコードの削除によって発生した消失の数を示しており，“Shrinking” はブロックに修正が加えられた結果トークン数が閾値を下回ったことで消失と判断されたコードクローンの数を示している．また “CRD limitation” は，条件式部分に変更が加わった場合など，CRD の比較では不一致となりブロックの追従ができなくなったために消失とみなされたコードクローンの数を示している．

表 3 より，コードクローンの除去を目的としたリファクタリングはほとんど行われていないことがわかる．また，意図的でない一貫性のない修正による消失も少なからず存在していることがわかる．このような消失は不具合の混入を招く危険性が高いといえる．

図 5 に，Ant から発見された，意図的でない一貫性のない修正の例を示す．この例で示されているコードクローンは図中の二つのブロックから構成される要素数が二のコードクローンである．リビジョン 671,018 において一方のブロックには変数が `null` でないことを確認するための処理が追加されているが，もう一方のブロックにはこの処理が追加されていなかった．しか

表 3　コードクローンの消失理由
Table 3　Reasons why clones were gone

Software	Refactoring	Different evolution	Unintended inconsistency	Unneeded code deletion	Shrinking	CRD limitation	Total
ArgoUML	18 (5)	17	11	58	3	6	115
Ant	31 (3)	14	13	13	0	37	107

し，これらの二つのブロックが意味的に同じ処理を行っていることから，もう一方のブロックに対しても変数が `null` でないことを確認する処理を追加する必要があると考えられる．

3.4　RQ3 に対する回答

消失したコードクローンの特性を調査するために，以下の項目について消失したコードクローンとそうでないコードクローンの間で統計的に優位な差が存在するか否かを調査した．

- コードクローンを構成するトークン数
- サイクロマチック数
- メソッド呼び出しの回数
- コードクローンを構成する要素数

調査の結果，サイクロマチック数並びに構成要素数については，消失したコードクローンの方がそうでないコードクローンより統計的に小さいという結果となった．すなわち，消失したコードクローンはそうでないコードクローンと比べて複雑ではなく，かつ構成要素数は小さい傾向にあることがわかった．

4.　調査の妥当性について

ビッグコミットの存在

本調査で対象としたソフトウェアのリポジトリには，多数のファイルが変更されている，いわゆる “ビッグコミット” がいくつか存在していた．例えば，MoveClass リファクタリングが行われたリビジョンなどは，リファクタリングによって修正されたソースファイルすべてについて CRD によるブロックの追跡が不可能となるため，それらのソースファイルに含まれるコードクローンがすべて消失として報告される．このようなビッグコミットを除外することで誤検出が大幅に削減できる可能性がある．

CRD によるブロックの追跡

本調査ではリビジョン間のブロックの追跡を行う上で，既存手法である CRD を用いた．しかし調査の結果，消失したコードクローンとして報告されたもののうち 19%が，実際は消失していないにも関わらず CRD による追跡ができなかったことによって消失として報告された．しかしながら，CRD による追跡の失敗は消失の誤検出を引き起こすものの，消失の検出漏れを引き起こすものではないため，この問題点は深刻なものではないと考えている．

目視による調査

本調査では，コードクローンの消失理由を特定する際に著者らによる目視調査を行った．しかし，著者らは対象ソフトウェアの開発者ではないため，この目視調査に誤りが含まれる可能性がある．したがって，今回の調査では誤りの混入を低減するために，著者のうち二人が共同して目視調査を行った．

対象ソフトウェア

本調査では二つのオープンソースソフトウェアに対しての調査を行った．しかし，対象ソフトウェアの数は十分とはいえず，また対象言語も Java のみとなっている．より多くのソフトウェアに対する実験や，異なる言語で記述されたソフトウェアに対する実験を行うことで，本調査で得られた結果と異なる結果が導かれる可能性がある．

5.　あ と が き

本稿ではコードクローンの消失を特定する手法を提案し，オープンソースソフトウェアに対する調査を行った．調査の結果，コードクローンの消失は多数発生していることが確認できた．また，不具合の混入につながる危険性が高い，意図的でない一貫性のない修正によるコードクローンの消失が少なからず起きていること，及びコードクローンの除去を目的としたリファクタリングはほとんど行われていないことがわかった．さらに，消失を含むコードクローンはそうでないコードクローンと比較して要素数が少なく，かつ複雑度が低い傾向にあることがわかった．

今後の課題として，より多くのソフトウェアに対する調査，及び Java 以外の言語で記述されたソフトウェアに対する調査を行うことを考えている．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究 (A)(課題番号 : 21240002)，萌芽研究 (課題番号 : 23650014, 24650011)，若手研究 (A)(課題番号 : 24680002) の助成を得た．

文　　献

[1] 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.
[2] 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，Aug. 2011.
[3] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proc. of the 2nd Working Conference on Reverse Engineering, pp.86–95, July 1995.
[4] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen, “Clone-aware configuration management,” Proc. of the 24th International Conference on Automated Software Engineering, pp.123–134, Nov. 2009.
[5] C.J. Kapser and M.W. Godfrey, ““cloning considered harmful” considered harmful: patterns of cloning in software,” Empirical Software Engineering, vol.13, no.6, pp.645–692, Dec. 2008.
[6] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” Proc. of ESEC/FSE-13, pp.187–196, Sept. 2005.
[7] 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, July 2010.