情報処理学会論文誌 Vol.54 No.2 835–844 (Feb. 2013)

大規模なソフトウェア群を対象とするメソッド単位でのコードクローン検出

石原 知也[1,a)] 堀田 圭佑[1,b)] 肥後 芳樹[1,c)] 井垣 宏[1,d)] 楠本 真二[1,e)]

受付日 2012年5月14日, 採録日 2012年11月2日

概要：ソフトウェア間にまたがるコードクローンを検出することは，多くのプロジェクトに頻出する処理のライブラリ化による開発効率の向上やライセンスに違反したソースコード流用の特定などの観点から有益である．しかし，既存の研究ではこのようなコードクローンの検出に多大な時間を必要とし，また高速に検出を行うファイル単位の検出手法でもファイルの一部がコードクローンである場合は検出できないという問題点をかかえている．本研究では，大規模なソフトウェア群からメソッド単位のコードクローンを高速に検出する手法を提案する．実験の結果，提案手法は約 3 億 6 千万行のソースコードから約 4.45 時間でコードクローン検出を終了し，検出したコードクローンの 40%はファイル単位の手法では検出できないことが確認できた．

キーワード：コードクローン，ソースコード流用，ソフトウェア保守

Method Clone Detection for a Large Number of Software Systems

Tomoya Ishihara[1,a)] Keisuke Hotta[1,b)] Yoshiki Higo[1,c)] Hiroshi Igaki[1,d)] Shinji Kusumoto[1,e)]

Received: May 14, 2012, Accepted: November 2, 2012

Abstract: Detecting code clones across software systems is useful from the viewpoint that we can discover source code license violations or improve work efficiency by merging common functions into libraries. However, existing methods need much time to detect code clones from software systems and file-based code clone detection, quickly detects code clones from software systems, cannot detect partially-duplicated files. In this research, we propose a method that detects method-based code clones in a huge data set. As a result of experiments, it took about 4.45 hours to detect code clones from about 360 million lines of source code by using the proposed method. Also, we found that 40% of code clones cannot be detected by using file-based code clone detection.

Keywords: code clone, source code plagiarism, software maintenance

1. はじめに

コードクローンとは，ソースコード中に存在する同一，あるいは類似したコード片のことである．ソースコードの流用などの理由により，複数のソフトウェアにまたがるコードクローンが多数存在することが予測されている [1], [2]．このような複数のソフトウェアにまたがるコードクローンを検出することは，多数のソフトウェア間に頻出する処理のライブラリ化による開発効率の向上やライセンスに違反して流用されているソースコードの特定などの観点から有益である．しかし，既存のほとんどのコードクローン検出手法 [3], [4] は単一のソフトウェア内のコードクローンを見つけることを目的として，ソースコードを文や字句などの細粒度で比較している．そのため，大規模なソフトウェア

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] t-ishihr@ist.osaka-u.ac.jp
[b)] k-hotta@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp
[d)] igaki@ist.osaka-u.ac.jp
[e)] kusumoto@ist.osaka-u.ac.jp

群を対象としたコードクローン検出には膨大な時間的・空間的コストを必要とする．

この問題を改善し，大規模なソフトウェア群からより低い調査コストでコードクローンを検出する手法として，ファイル単位のコードクローン検出手法が提案されている [5], [6]．ファイル単位のコードクローン検出手法は高速にコードクローンを検出できる反面，ファイルの一部がコードクローンである場合はその部分をコードクローンとして検出できない．すなわち，ファイルのある一部分のみが流用されている場合，ファイル単位のコードクローン検出手法では流用部分をコードクローンとして検出できない．

そこで本研究では，メソッド単位でのコードクローン検出手法を提案する．メソッド単位のコードクローン検出手法には以下の 2 つのメリットが存在する．

- 大規模なソフトウェア群に対して短時間で検出できる．
- ファイルの一部のメソッドがコードクローンである場合でもそのコードクローンを検出できる．

実験の結果，約 3 億 6 千万行のソースコードから約 4.45 時間でコードクローンを検出できた．また，検出されたコードクローンを分析した結果，複数のソフトウェア間に共通する処理のライブラリ化やソースコード流用の特定の支援に有用なコードクローンの存在を確認できた．

2. 準備

2.1 ソフトウェア間にまたがるコードクローン

本研究では，ソフトウェア間にまたがるコードクローンに着目する．ソフトウェア間にまたがるコードクローンを検出する目的として，以下の点があげられる [5], [6]．

複数のソフトウェアに存在する共通処理のライブラリ化

複数のソフトウェアにまたがってコードクローンとして検出されたコード片が行う処理は，複数のソフトウェアで共通する処理である．複数のソフトウェアで共通する処理は，今後のソフトウェア開発でも必要とされる可能性が高い．このような複数のソフトウェアで共通する処理をライブラリ化することで，後発のソフトウェア開発プロジェクトで新たに処理を記述する必要がなくなり，開発効率の向上が期待できる．

他のソフトウェアからライセンスに違反して流用されたソースコード特定の支援

ソフトウェア間にまたがるコードクローンの生成要因の 1 つとして，他のソフトウェアからのソースコード流用があげられる．流用されたソースコードの中には，ライセンスに違反したソースコードが含まれている可能性がある．ソフトウェア間にまたがるコードクローンを検出することで，ライセンスに違反して流用されているソースコードの特定を支援できる．

2.2 ファイル単位のコードクローン検出手法

行や字句単位などの既存のコードクローン検出手法は，主に単一のソフトウェアを検出対象としている．そのため，このような既存の手法は詳細にコードクローンを検出すべくソースコードを細粒度で比較している．しかし大規模なソフトウェア群を対象に検出を行う場合，細粒度のコードクローン検出手法ではその検出処理に膨大な時間的・空間的コストを必要とする．このため，大規模なソフトウェア群から短時間でコードクローンを検出することは困難である．

この問題を解決するために，ファイル単位のコードクローン検出手法が提案されている [5], [6]．ファイル単位のコードクローン検出手法では，ファイルを単位として比較を行い，一致する場合のみコードクローンと見なす．この手法は，行単位や字句単位のような細粒度の検出手法と比べてソースコードの比較回数が小さくなるため，高速な検出が可能である．しかし，ファイルの一部がコードクローンである場合は検出できない．

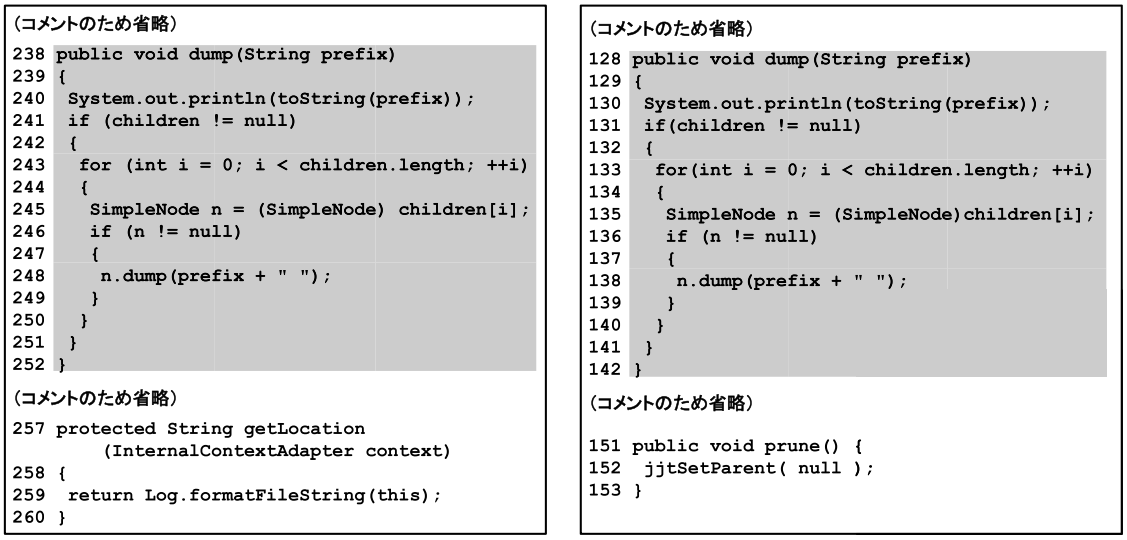
以降，ファイル単位のコードクローンを**ファイルクローン**，メソッド単位のコードクローンのことを**メソッドクローン**と呼ぶ．

3. 研究動機

3.1 既存研究

佐々木らは，ファイルクローン検出ツール FCFinder を開発し，ファイルクローンの性質を調査した [5]．FCFinder はファイルをハッシュ値に変換しファイルクローンを検出する．また，FCFinder はコメント文の削除や字句解析を行うため，コメント文やインデントの違いを吸収できる．佐々木らは，総行数約 4 億行の大規模なソフトウェア群である FreeBSD Ports Collection に対し FCFinder を適用した結果，17 時間ほどで検出を終了し，FreeBSD Ports Collection の約 68%がファイルクローンであったことを報告している．また，検出されたファイルクローンのうち 27%はコメント文やインデントの違いであり，ファイルサイズの分布はファイルクローンであるファイルとそうでないファイルとで差異がなかったとも報告している．

Ossher らは，ファイルクローン検出手法を提案し，ファイルクローンが発生する状況を調査した [6]．Ossher らの手法は，exact，FQN，fingerprint の 3 つの要素を組み合わせてファイルクローン検出を行う．exact は，各ソースファイルを 1 つの文字列と見なしその文字列をハッシュ値に変換し，ハッシュ値が一致したファイルをファイルクローンとして検出する．FQN は，クラスの完全限定名が一致しているファイルをファイルクローンとして検出する．fingerprint は，ソースファイル中のメソッド名とフィールド名がどの程度等しいかを調査し，ある閾値を超えて一致しているファイルをファイルクローンとして検出する．



```
（コメントのため省略）
238 public void dump(String prefix)
239 {
240  System.out.println(toString(prefix));
241  if (children != null)
242  {
243   for (int i = 0; i < children.length; ++i)
244   {
245    SimpleNode n = (SimpleNode) children[i];
246    if (n != null)
247    {
248     n.dump(prefix + " ");
249    }
250   }
251  }
252 }
（コメントのため省略）
257 protected String getLocation
        (InternalContextAdapter context)
258 {
259  return Log.formatFileString(this);
260 }
```

(a) apache.velocity　SimpleNodeクラス

```
（コメントのため省略）
128 public void dump(String prefix)
129 {
130  System.out.println(toString(prefix));
131  if(children != null)
132  {
133   for(int i = 0; i < children.length; ++i)
134   {
135    SimpleNode n = (SimpleNode)children[i];
136    if (n != null)
137    {
138     n.dump(prefix + " ");
139    }
140   }
141  }
142 }
（コメントのため省略）
151 public void prune() {
152  jjtSetParent( null );
153 }
```

(b) Jedit　SimpleNodeクラス

図 1　ファイル単位の検出手法では検出できないコードクローン
Fig. 1　Motivating example.

Ossher らは，約 1 万 3 千の Java ソフトウェアに対し上記の 3 つの要素を組み合わせて実験したところ，全ファイルの約 10%超がファイルクローンとして検出されたと報告している．またファイルクローンが発生する状況として，同じライブラリを使用していることや新たなソフトウェア開発を始めるときにそれ以前に開発されたソフトウェアの再利用を行うことなどがあげられるとも報告している．

Livieri らは，分散処理技術を応用して大規模なソフトウェア群から短時間でコードクローンを検出する手法を提案し，その手法を D-CCFinder として実装した [1]．D-CCFinder は既存のコードクローン検出ツール CCFinder に分散処理技術を加えている．対象とするソフトウェア群を一定のサイズで分割し，分割した各要素をそれぞれ別のコンピュータに割り振り，そこで CCFinder を実行する．Livieri らは，総行数約 4 億行の大規模なソフトウェア群である FreeBSD Ports Collection に対し D-CCFinder を適用した結果，CCFinder では 45 日を要すると予測されていたコードクローン検出処理が 51 時間で終了したと報告している．Livieri らの手法と本研究ではコードクローン検出の高速化を実現するアプローチが異なる．Livieri らは高コストである検出処理を分散させて多くの PC を利用することで高速化を図っているが，我々は検出単位の粒度を大きくすることで検出コストを削減し，1 台の PC で高速化を実現している．

3.2　既存研究の課題点

図 1 は，ソフトウェア間にまたがるコードクローンの例である．ハイライトされている 2 つのメソッドは木構造の表示のために使用される dump メソッドである．このメソッドを含むファイルは，ファイル全体としてはコードクローンにはなっていなかった．つまり，ファイルクローン検出手法ではこのメソッドをコードクローンと判断できない．そこで本研究では，この課題点を解決するためにより細かい粒度であるメソッド単位のコードクローン検出手法を提案する．

3.3　課題設定

本研究では，メソッドクローンに関する以下の論点を調査する．

RQ1　ファイル単位の検出手法で検出できなかったコードクローンがメソッド単位の検出手法で新たにどの程度検出されるか．

RQ2　メソッド単位の検出手法は大規模なソフトウェア群に対して短時間で検出を終了できるか．

RQ3　メソッド単位の検出手法で新たに検出されたコードクローンの発生原因にはどのようなものが存在するか．

RQ4　メソッド単位の検出手法で新たに検出されたコードクローンは複数のソフトウェアに存在する共通処理のライブラリ化に有用であるか．

RQ5　メソッド単位の検出手法で新たに検出されたコードクローンはライセンスに違反したソースコード流用の特定の支援に有用であるか．

4.　提案手法

この章では，提案するメソッドクローン検出手法について説明する．提案手法は対象とするソースファイル群を入力として受け取り，どのメソッドがコードクローンと判断されたかという情報が入っているデータベースを出力として返す．**図 2** に提案手法の概要を示す．以降，各ステップについて詳細に説明する．

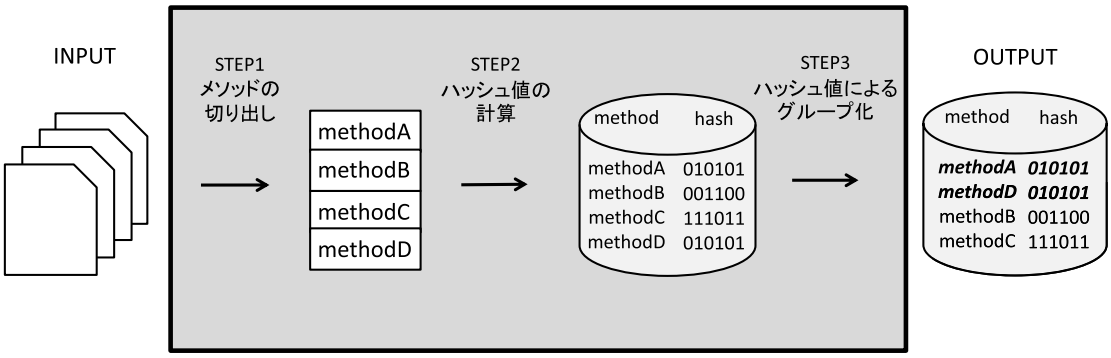


図 2 提案手法の概要
Fig. 2 Overview of proposed method.

4.1 各 STEP の説明

STEP1：メソッドの切り出し 与えられたソースファイルからメソッドを切り出す．ソースファイルから抽象構文木を作成することでメソッドの切り出しを実現する．抽象構文木を作成することで，改行や空白回数の違いなどが吸収される．

STEP2：ハッシュ値の計算 STEP1 で切り出されたメソッドに対してハッシュ値を算出する．具体的には，メソッドを 1 つの文字列と見なしその文字列に対してハッシュ値の計算を行う．等しい記述を持つメソッドはハッシュ値が等しくなるため，ハッシュ値が等しいメソッドどうしはコードクローンの関係にある．本研究では，MD5 [7] を用いてハッシュ値を計算し，算出されたハッシュ値はメソッドごとにデータベースに格納する．ただし，元の記述が異なっているにもかかわらず計算されたハッシュ値が等しくなってしまう「ハッシュ値の衝突」が発生した場合は正しくコードクローンを検出できない．ハッシュ値の衝突がどの程度起こるかについては 7 章で論じる．

STEP3：ハッシュ値によるグループ化 STEP2 で算出されたハッシュ値の等しいメソッドをグループ化する．等しいハッシュ値を持つメソッドは等しい記述を持つ．そのため，ハッシュ値が等しいメソッドをグループ化することで，互いにコードクローンであるメソッドどうしが 1 つのグループを構成する．これにより，コードクローンの関係にあるメソッドがどれか判別できるようになる．

4.2 Java に対する実装

本研究では，Java を対象として実装を行った．以下に Java での実装方法を記す．

- 本研究では，抽象構文木の作成に JavaDevelopment-Tools [8]（以降 JDT と呼ぶ）を使用している．JDT ではメソッドの内容はメソッド宣言ノード以下に作成されるため，メソッド宣言ノード以下の部分木を抽出することによってメソッドの切り出しを実現している．
- 変数名の違いやコメント文の有無などソースコードの振舞いに影響を与えない箇所の違いを吸収するために正規化処理を行う．本研究では，以下に示す正規化処理を行っている．
  - 変数名，文字列リテラル，メソッド宣言部のメソッド名は特殊文字列に置換する．
  - 修飾子，アノテーション，コメント文は削除する．
- Java 言語で記述されたソースファイルには，getter メソッドや setter メソッドのように処理が単純でありかつ短いメソッドが多くのソースファイルに存在する．このようなメソッドは多くが return 文や簡単な代入文のみで構成されているため，大量にコードクローンとして検出されるおそれがある．このようなコードクローンは流用の特定に有用でなく，処理が簡単に記述できるためライブラリ化する価値が小さい．したがって，処理が単純かつ短いメソッドに対してのコードクローンの検出は不要である．そこで，このようなメソッドはハッシュ値の計算を行わないようにフィルタリングする．本研究では，メソッド内の複文が 0 であるメソッドのハッシュ値は計算しない．ここで複文とは，複数の文から成り立つブロックを持つ文のことを指し，do，for，if，switch，synchronized，try，while 文を複文と定義する．

5. 実験の準備

5.1 メトリクスの定義

ここでは実験の際に使用するメトリクスを定義する．本研究では以下の 2 つのメトリクスを使用する．

NoS メソッドクローンとして検出されるメソッドが分布しているソフトウェアの数を NoS（the Number Of Software system）と定義する．多くのソフトウェアで使用されているメソッドは，メソッドクローンとして検出されたときに NoS の値が大きくなる．したがって，NoS の値が大きいメソッドクローンはライブラリ化すべきであると考えられる．

メソッドクローン共有数，メソッドクローン共有率 ある

2ファイル間で共通するメソッドクローンの数を2ファイル間におけるメソッドクローン共有数と定義し，一方のファイルにある全メソッド数に対するメソッドクローン共有数の割合をメソッドクローン共有率と定義する．メソッドクローン共有率の高いファイルのペアは，ファイルに存在するほとんどのメソッドがコードクローンの関係にあることを示している．つまり，一方がもう一方のファイルを流用した後少し修正を加えた可能性が高いと考えられる．

5.2 実験方法

本研究では，提案手法の有効性を示すために2つの実験を行った．

実験1 メソッド単位の提案手法とファイル単位の手法でコードクローン検出を行い，検出結果や検出時間を比較してどの程度違いがあるかを調査する．

実験2 提案手法で検出したコードクローンを分析し，流用の特定や処理のライブラリ化に有用なコードクローンがどの程度存在するかを調査する．

実験1ではファイル単位のコードクローン検出が必要であるため，本研究ではファイルクローン検出も行った．ファイルクローン検出は，解析対象となるファイルそれぞれに対してその文字列をもとにMD5によりハッシュ値を算出し，ハッシュ値に基づいてグループ化することによって実現される．このとき，解析対象のファイルには4章で説明した正規化処理に加えて，以下の正規化処理を行っている．

- パッケージ文，インポート文の削除
- クラス，インタフェース宣言部の名前定義部分を特殊文字列に置換

提案手法ではメソッド切り出しを行っているため，メソッド単位のコードクローン検出のみを行う場合上記の正規化は必要ない．本研究では，実験の際にファイル単位のコードクローン検出を必要とするため，これらの正規化を加えている．これらの正規化を行うことでパッケージ宣言などの違いが吸収され，より多くのファイルがファイルクローンとして検出される．

実験2では5.1節で定義したメトリクスをもとに検出したメソッドクローンの分析を行う．

5.3 実験対象

本研究では，実験対象として“UCI source code data sets” [6], [9]（以降 *UCIdatasets* と呼ぶ）を使用した．*UCIdatasets* はweb上に存在するオープンソースソフトウェアを含むデータセットである．本研究ではソフトウェア間にまたがるコードクローンの検出を目的としており，できる限り多くのソフトウェアを含む大規模なデータセットに対し提案手法を適用する必要がある．そのため，我々が調査した中で最も多くソフトウェアを含んでいたこのデータセットを実験対象とした．

表1 実験対象の構成
Table 1 Structure of the target data set.

全ファイル数	3,963,896
検出対象ファイル数	2,072,490
ソフトウェア数	13,193
検出対象メソッド数	5,953,165
検出対象ファイルの総行数	361,663,992
全容量	30.6 GByte

データセット内には，subversionで管理されているtrunk，tags，branchesを同時に含むソフトウェアやバージョンの異なる同一のソフトウェアが存在する．ソフトウェアは処理の追加，修正，削除を行うことで新しいバージョンに進化していくが，すべてのソースファイルを変更するわけではない．そのため，バージョンの異なる同一のソフトウェアに対してコードクローン検出を行うと，変更されていないソースファイルはコードクローンとして検出されてしまう．このように不必要に多くのコードクローンを検出してしまうため，本研究ではデータセット内でtrunk，tags，branchesを同時に含むソフトウェアはtrunk内のソースファイルのみを，バージョンの異なる同一のソフトウェアは最新バージョンのソフトウェアのみを検出対象とした．

また，いくつかのソフトウェアにはソースコード自動生成ツールによって生成されたソースファイルが存在していた．同じツールによって生成されたソースコードはコードクローンとして検出されるが，このようにして検出されたコードクローンは処理のライブラリ化や流用の特定に有用でない．なぜならば，同じツールによって作成されるため，流用特定の証拠としてのコードクローンにはならずライブラリ化する価値が低いからである．以上の理由により本研究では，コメントに“generated by”が含まれているソースファイルは自動生成ツールで生成されたソースファイルと見なして検出対象から除外した．

さらに，検出時間の短縮を図るために，データセット内にある拡張子.javaを持つファイル以外のすべてのファイルをデータセットから削除した．このような前処理を行うことで，解析対象ではないファイルに対して解析対象であるかどうかの判定処理を行わなくなるため処理が高速化する．

表1は，拡張子.javaファイル以外のファイルを削除した後の *UCIdatasets* の構成を示している．上記処理の結果，検出対象ファイル数が約半数に減少している．

6. 実験結果

6.1 検出したコードクローン数

実験1の結果，メソッドクローンとして検出されたメソッド数は2,937,047，クローンセット（互いにメソッド

表 2 検出されたメソッドクローンの数
Table 2 Number of detected method clones.

	ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
ファイルクローンの一部	1,407,338（24%）	365,150（6%）	1,772,448（30%）
ファイルクローンでない	658,500（11%）	506,059（9%）	1,164,559（20%）
合計	2,065,838（35%）	871,209（15%）	2,937,047（49%）

表 3 検出されたコードクローンを含むファイルの数
Table 3 Number of files including code clones.

	ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
検出対象メソッドを含むファイルクローン	288,185（14%）	84,213（4%）	372,398（18%）
検出対象メソッドを含まないファイルクローン	304,779（15%）	114,412（6%）	419,191（20%）
ファイルクローン合計	592,964（29%）	198,625（10%）	791,589（38%）
ファイルクローンではないがメソッドクローンを含む	147,532（7%）	140,668（7%）	288,200（14%）
合計	740,496（36%）	339,293（16%）	1,079,789（52%）

表 4 解析時間
Table 4 Analyzing time.

処理内容	時間
メソッドの切り出し 正規化 ハッシュ値の計算	2.98 h
ハッシュ値のグループ化	1.47 h
合計	4.45 h

クローン関係にあるメソッドの集合）数は 814,391 となった．814,391 個のクローンセットのうち，約 60%にあたる 490,206 個のクローンセットが複数のソフトウェアにまたがっていた．

表 2，**表 3** は，それぞれメソッドクローンとコードクローンを含むファイルの分析結果である．表中の百分率はそれぞれ全検出対象メソッド数と全検出対象ファイル数に対する割合である．表 2 より，ファイルクローンでないファイルに存在している 1,164,559 個のメソッドクローンは，検出された全メソッドクローン数 2,937,047 の 40%を占めていることが分かる．また表 3 より，ファイルクローンではないがメソッドクローンを含む 288,200 個のファイルは，コードクローンを含む全ファイル数 1,079,789 の 27%であることが分かる．

6.2 解析速度

実験 1 において，1CPU，4core（2.00 GHz），メモリ 8 GByte の環境で提案手法を *UCIdatasets* に適用した結果，4.45 時間でコードクローンの検出が完了した．**表 4** に各処理に要した時間を示す．このような検出時間でコードクローン検出が完了した理由として，データベースを SSD 上においているためデータベースアクセス速度が高速であることや，*UCIdatasets* から今回解析対象とする拡張子が.java であるファイル以外を削除していることがあげられる．比較対象としてファイルクローン検出の検出時間を測定したところ，同じ環境，対象で 2.70 時間で検出を終了している．アルゴリズム上提案手法のほうが検出時間はかかるものの，十分に短時間で検出が完了したといえる．

また，既存の字句単位のコードクローン検出ツールである CCFinder [3] との解析速度の比較も行った．ただし，本研究で実験対象としたデータセットは非常に大規模であるため，データセットをサブセットに分割し，サブセットのペア 1 つにおける CCFinder の検出時間を測定することで，CCFinder の全データセットに対する検出時間を概算し比較している．実際にデータセットを 1,000 ファイルごとに分割し，約 2,000 のサブセットを構築した．組合せの総数は $2{,}000 \times 2{,}000 \div 2 = 2{,}000{,}000$ 通りであり，ランダムに 150 ペアを抽出し各ペアに CCFinder を適用したところ，1 つのペアに対する検出時間の平均は 41 秒であった．このことから全データセットに対する CCFinder の検出時間は約 950 日と推定される．この結果から，字句単位の検出手法と比較して提案手法は大規模なソフトウェア群から非常に高速にコードクローン検出を終えていることが分かる．

6.3 発生原因

実験 1 の結果をみると検出されたメソッドクローンの数が非常に多いことが分かる．そのため，実験 2 ではすべてのメソッドクローンを分析するのではなく，5.1 節で定義した NoS メトリクスの上位 100 位までのメソッドクローンを著者の 1 人が実際に目で見て調査した．その結果，メソッドクローンの発生原因を以下に示す 3 つに分類できた．**表 5** は発生原因とそれに該当すると判断されたメソッドクローン数を示している．該当数の合計が 100 を超えて

いるが，これは発生原因に重複があるためである．

抽象クラスの継承，インタフェースの実装 抽象クラスを継承するクラスやインタフェースを実装するクラスでは，すべての抽象メソッドをオーバライドしなければならない．またこのようなクラスでは，新たに処理を記述する際に別々のソフトウェアであっても処理が類似することがある．その結果，多くのソフトウェアにまたがるコードクローンが発生する．実際に見つかった例では，FileFilter クラスをオーバライドしたクラスがこれに該当する．FileFilter クラスは抽象クラスであり，そこで定義されている抽象メソッドをオーバライドしたメソッドが NoS 値の高いメソッドクローンとして検出された．

ソースコードの流用 ソフトウェア開発では，web 上などで公開されているソースコードを開発中のソフトウェアに使用することがある．このとき，自分の作成したソースコードに対応させるといった理由で，使用するソースコードに変更を加えることがある．その結果，ソースコードの一部のメソッドがコードクローンとして検出される．実験 2 の結果，ファイル内の全メソッドのうち 9 割以上のメソッドがコードクローンであり一部のメソッドのみが追加，修正，削除されているファイルを発見した．このようなファイルは流用を行ったために生成されたファイルであると判断した．実際に見つかった例では，64 bit-encoder,decoder を実装したクラスがこれに該当する．このクラスはどのソフトウェアにおいても開発者名が等しかったため，ソースコードの流用を行ったのではないかと判断した．

汎用的な処理を行うメソッド size や close といったメソッドは多くのクラスで定義されている．このようなメソッドは，特定の変数が null や 0 であるといった条件判定と併用されやすい．そのため，4.2 節で述べたフィルタリング処理を通過しコードクローンとして検出される．NoS メトリクスの上位 100 位までのメソッドクローンには，実際に size メソッドや close メソッドをコードクローンと判断しているものが複数存在した．

表 5 NoS メトリクス上位 100 のメソッドクローンの分類
Table 5 Classification of the top 100 method clones in NoS ranking.

発生原因	代表的な例	該当数
抽象クラスの継承 インタフェースの実装	FileFilter クラス	34
ソースコードの流用	64 bit-encoder,decoder	42
汎用的な処理を行うメソッド	size メソッド	44

7. 議論

7.1 検出したメソッドクローンの有用性

提案手法によって検出されたコードクローンを調査したところ，流用した可能性の高いファイルやライブラリ化すべきメソッドを検出できた．以下に検出した有用なコードクローンの一例を示す．いずれの場合もファイル全体がコードクローンにはなっていないのでファイル単位の検出手法では検出できない．

複数のソフトウェアに存在する共通処理のライブラリ化

汎用的な処理を行うメソッドは多くのクラスで使用されるためライブラリ化することが難しい．そのため，6.3 節で調査した 100 のメソッドクローンのうち，汎用的な処理を行うメソッドである 44 のコードクローンを除く 56 のコードクローンがライブラリ化可能なメソッドであると判断した．**図 3** は実際にライブラリ化された例である．2 つのメソッドはテーブルに関する処理を行うクラスで宣言されているソートを行うメソッドである．swing の JTable におけるソート機能は Java1.6 から実装された機能であるため，それ以前の開発ではソート機能は開発者が自ら実装する必要があった．

他のソフトウェアからライセンスに違反して流用されたソースコードの特定

流用の結果生じたファイルは流用されたファイルとのメソッドクローン共有率が高いと考えられる．そのため，5.1 節で定義したメソッドクローン共有率が 90%を超えるファイルペアを抽出し，ライセンスや開発者が等しいファイルのペアを除いたところ 44 のファイルペアが残った．これらのファイルペアはライセンスに違反する流用を行っている可能性があると判断した．**図 4** はその一例である．

```
244 public void n2sort() {
245  for(int i = 0; i < getRowCount(); i++) {
246   for(int j = i+1; j < getRowCount(); j++) {
247    if (compare(indexes[i], indexes[j]) == -1) {
248     swap(i, j);
249    }
250   }
251  }
252 }
```

(a) sun TableSorterクラス

```
219 public void n2sort() {
220  for (int i = 0; i < getRowCount(); i++) {
221   for (int j = i+1; j < getRowCount(); j++) {
222    if (compare(indexes[i], indexes[j]) == -1) {
223     swap(i, j);
224    }
225   }
226  }
227 }
```

(b) Perham TableSorterクラス

図 3 ライブラリ化すべきメソッドクローンの例
Fig. 3 An example of the method clone which should be merged into libraries.

```
153 public boolean isSorting() {
154  return sortingColumns.size() != 0;
155 }
156
157 private Directive getDirective(int column) {
158  for (int i = 0; i < sortingColumns.size(); i++) {
159   Directive directive =
        (Directive)sortingColumns.get(i);
160   if (directive.column == column) {
162    return directive;
162   }
163  }
164  return EMPTY_DIRECTIVE;
165 }
166
167 public int getSortingStatus(int column) {
168  return getDirective(column).direction;
169 }
170
171 private void sortingStatusChanged() {
172  clearSortingState();
173  fireTableDataChanged();
174  if (tableHeader != null) {
175   tableHeader.repaint();
176  }
177 }
```

(a) newsreaver TableSorterクラス

```
175 private Directive getDirective(int column) {
176  for (int i = 0; i < sortingColumns.size(); i++) {
177   Directive directive =
        (Directive)sortingColumns.get(i);
178   if (directive.column == column) {
179    return directive;
180   }
181  }
182  return EMPTY_DIRECTIVE;
183 }
(コメントのため省略)
190 public void setSortingStatus
        (int column, int status){
191  Directive directive = getDirective(column);
192  if (directive != EMPTY_DIRECTIVE) {
193   sortingColumns.remove(directive);
194  }
195  if (status != NOT_SORTED) {
196   sortingColumns.add(new Directive(column, status));
197  }
198  sortingStatusChanged();
199 }
(コメントのため省略)
206 public int getSortingStatus(int column) {
207  return getDirective(column).direction;
208 }
```

(b) vspm TableSorterクラス

図 4 ライセンスに違反した流用特定の例

Fig. 4 An example of the method clone which supports detecting plagiarism.

2 つのソースファイルには GUI におけるテーブルのソート機能が実装されており，ハイライトされている部分が提案手法によってそれぞれメソッドクローンのペアとして検出された．newsreaver は NNTP のニュースリーダ，vspm は従業員の管理システムであり，いずれのソフトウェアも GUI におけるテーブルのソート機能を必要としている．

図 4 の 2 つのソースファイルを調査した結果，以下の 4 点が判明した．

(1) (a) のメソッド数は 15，(b) のメソッド数は 12 であり，11 個のメソッドが 2 ファイル間で共有されるメソッドクローンである．
(2) 2 つのソースファイルの開発者が違う．
(3) (a) は (b) よりも開発日が 1 年ほど早い．
(4) newsreaver のライセンスは LGPL，vspm のライセンスは GPL．

(1)，(2) よりほとんどのメソッドがメソッドクローンであるにもかかわらず開発者が違うため，2 ファイル間で流用が行われたと判断できる．次に，(3) より流用を行ったのは (b) であると分かる．最後に，(4) より (a) のライセンスは LGPL である．LGPL で保護されるソースコードを変更する場合は，それらを変更したということと変更した日時がよく分かるよう，改変されたファイルに告示しなければならない．しかし，開発者名やソースの一部が変更されている (b) はその旨を記述していない．以上の理由により，(b) はライセンスに違反して流用を行っていると判断できる．

7.2 課題に対する回答

3.3 節で述べた RQ に対して得られた考察を述べる．

RQ1 は，ファイル単位の検出手法で検出できなかったコードクローンがメソッド単位で検出を行うことで新たにどの程度検出できるかという課題である．今回対象とした *UCIdatasets* ではファイルクローン検出手法に対し，新たに 1,164,559 のメソッドクローンが検出され，それらは全メソッドクローンの約 40%を占めることが確認された．また，ファイルクローンではないがメソッドクローンを含むファイルが 288,200 存在し，それらがコードクローンを含むファイルの約 27%を占めることが確認された．

RQ2 は，メソッド単位の検出手法は大規模なソフトウェア群に対して短時間で検出を終了できるかという課題である．6.2 節で述べたように，メソッド単位の検出手法は十分に短時間で検出を終了している．

RQ3 は，メソッド単位で検出を行うことで新たに検出されたコードクローンの発生原因にはどのようなものがあるかという課題である．実験の結果，以下のような発生原因が存在した．

- 同じ抽象クラスを継承する複数のクラスを実装する際に処理の内容が類似する．
- ソースコードを流用した後，それぞれのソフトウェアに応じた変更を加える．
- サイズを取得する処理や終了処理などの汎用的な処理のため処理内容が類似する．

RQ4 は，メソッド単位の検出手法で新たに検出されたコードクローンは複数のソフトウェアに存在する共通処理のライブラリ化に有用であるかという課題である．7.1 節で述べたように，メソッド単位のコードクローン検出で新たに検出したコードクローンの中に，複数のソフトウェア間に共通する処理のライブラリ化に有用なコードクローンが複数存在した．さらに，実験の結果新たに検出したメソッドクローンの中に，実際にライブラリ化が行われたメ

ソッドが存在した．

RQ5 は，メソッド単位の検出手法で新たに検出されたコードクローンはライセンスに違反したソースコード流用の特定の支援に有用であるかという課題である．7.1 節で述べたように，メソッド単位のコードクローン検出によって実際にライセンスに違反して流用したと思われるソースファイルを発見した．そのソースファイルはファイル全体がコードクローンではないため，ファイルクローン検出手法では検出できない．したがって，メソッド単位のコードクローン検出で新たに検出したコードクローンを用いることで，ライセンスに違反したソースコード流用の特定の支援に有用であると考えられる．

7.3 結果の妥当性

本研究の結果の妥当性に関して，以下にあげる点に留意する必要がある．

ハッシュ値の衝突 提案手法では 2 つのメソッドがコードクローンであるかの判定にハッシュ値を使用している．そのためコードクローン検出の際に異なるメソッドのハッシュ値が偶然一致するハッシュ値の衝突が起こった場合，本来コードクローンでないメソッドがコードクローンとして検出されるおそれがある．しかし，本研究では検出の対象となるメソッド数が約 600 万であり，かつハッシュ値の計算に 128 bit の MD5 アルゴリズムを使用しているため衝突確率はきわめて低い [10]．また，本研究では実際にハッシュ値の衝突が起こっているかどうかを，検出したすべてのコードクローンに対して調査し，ハッシュ値の衝突がなかったことを確認した．ただし，多くのソフトウェアに適用していけば，いずれハッシュ値の衝突が起こる可能性がある．

正規化，フィルタリング処理の選択 本研究では，4.2 節や 5.2 節で述べた正規化やフィルタリング処理を行っている．しかし，型名の正規化やメソッドサイズでのフィルタリングなど正規化やフィルタリング処理の方法を変更することによって本研究で得られた結果と異なる結果が得られる可能性がある．

バージョンの異なる同一ソフトウェアの存在 5.3 節で述べたように，*UCIdatasets* にはバージョンが異なる同一のソフトウェアが複数含まれている．本研究では，実験を行う前に可能な限り最新バージョンの以外のソフトウェアを解析対象から除外しているが完全ではなく，バージョンが異なる同一ソフトウェアが検出対象に含まれている可能性がある．そのため，不必要に多くコードクローンを検出している可能性がある．

8. おわりに

本研究では，大規模なソフトウェア群からメソッド単位のコードクローン検出を行った．実験の結果，検出したメソッド単位のコードクローンのうち約 40%，コードクローンを含むファイルのうち約 27%がメソッド単位の検出手法で新たに検出されたコードクローンであるという結果が得られた．また，複数のソフトウェア間に共通する処理のライブラリ化やライセンスに違反したソースコード流用の特定の支援に有用であるコードクローンが存在することが確認された．

本研究の今後の課題は以下のとおりである．

- 本研究とは別の正規化やフィルタリング処理を実装して検出されるコードクローンに違いがあるかを調査する．
- 対象となるファイルを Java 以外にも拡張して，言語によって検出されるコードクローンに違いがあるかを調査する．

参考文献

[1] Livieri, S., Higo, Y., Matushita, M. and Inoue, K.: Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder, *Proc. 29th International Conference on Software Engineering*, pp.106–115 (2007).

[2] Brown, A.W. and Booch, G.: Reusing open source software and practices: The impact of open source on commercial vendors, *Proc. 7th International Conference on Software Reuse*, pp.123–136 (2002).

[3] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).

[4] Mayrand, J., Leblanc, C. and Merlo, E.M.: Experiment on the automatic detection of function clones in a software system using metrics, *Proc. 12th IEEE International Conference on Software Maintenance*, pp.244–253 (1996).

[5] 佐々木裕介，山本哲男，早瀬康裕，井上克郎：大規模ソフトウェアシステムを対象としたファイルクローンの検出，電子情報通信学会論文誌 D，Vol.J94-D, No.8, pp.1423–1433 (2011).

[6] Ossher, J., Sajnani, H. and Lopes, C.: File cloning in open source java projects: The good, the bad, and the ugly, *Proc. 27th International Conference on Software Maintenance*, pp.283–292 (Sep. 2011).

[7] Rivest, R.: The MD5 message-digest algorithm, RFC 1321 (Informational) (Apr. 1992). available from ⟨http://www.ietf.org/rfc/rfc1321.txt⟩.

[8] Java development tools, available from ⟨http://www.eclipse.org/jdt/⟩.

[9] Lopes, C., Bajracharya, S., Ossher, J. and Baldi, P.: Uci source code data sets, available from ⟨http://www.ics.uci.edu/~lopes/datasets/⟩.

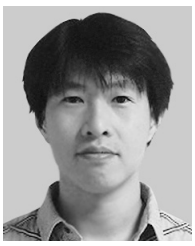
[10] Hummel, B., Jürgens, E., Heinemann, L. and Conradt, M.: Index-based code clone detection: Incremental, distributed, scalable, *Proc. 26th International Conference on Software Maintenance*, pp.1–9 (2010).

石原 知也

平成 24 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．コードクローン分析に関する研究に従事．

堀田 圭佑

平成 22 年大阪大学基礎工学部情報科学科卒業．平成 24 年同大学大学院情報科学研究科博士前期課程修了．現在，同研究科博士後期課程在学中．コードクローン分析に関する研究に従事．IEEE 会員．



肥後 芳樹 （正会員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院情報科学研究科博士後期課程修了．日本学術振興会特別研究員を経て，平成 19 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．コードクローン分析，リファクタリングに関する研究に従事．IEEE 会員．

井垣 宏 （正会員）

平成 12 年神戸大学工学部電気電子工学科卒業．平成 14 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．平成 17 年同研究科博士後期課程修了．同年同研究科特任助手．平成 18 年南山大学数理情報研究科講師．平成 19 年神戸大学工学部工学研究科情報知能学専攻特命助教．平成 22 年東京工科大学コンピュータサイエンス学部助教．平成 23 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻特任准教授．博士（工学）．ソフトウェア工学教育，サービス指向アーキテクチャ，ホームネットワークシステム，Web サービス，ソフトウェアプロセス等の研究に従事．IEEE，ACM，IEICE 各会員．

楠本 真二 （正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同学科講師．平成 11 年同学科助教授．平成 14 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教授．平成 17 年同専攻教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IEEE，JFPUG 各会員．