情報処理学会論文誌 Vol.56 No.2 580–592 (Feb. 2015)

粗粒度なコードクローン検出手法の精度に関する調査

堀田 圭佑[1,a)] 楊 嘉晨[1,b)] 肥後 芳樹[1,c)] 楠本 真二[1,d)]

受付日 2014年5月7日, 採録日 2014年11月10日

概要：コードクローン（ソースコード中の重複箇所）に関する研究が近年活発に行われており，コードクローンを自動的に検出する手法が多数提案されている．また近年，単一のプロジェクトだけでなく，多数のプロジェクトからなる巨大なデータセットや過去のバージョンのソースコードに対してコードクローン検出手法を適用し，ライブラリ候補の検出や修正漏れの検知を行う研究が行われている．しかし，コードクローン検出手法の多くはソースコードを詳細に分析する細粒度な検出方法を採用しており，大規模なデータセットを対象としたときに膨大な実行時間を要するという課題がある．その解決方法として，ソースコードに対して粗い解析を行う粗粒度なコードクローン検出手法を用いるという方法がある．しかし，粗粒度なコードクローン検出手法がどの程度の精度でコードクローンを検出できているのかは明らかになっていない．そこで本研究では，粗粒度なコードクローン検出手法をツールとして実装し，7 つの細粒度なコードクローン検出ツールとの比較を行った．その結果，粗粒度なコードクローン検出手法が細粒度なものと比較して高速であり，かつ細粒度なものに劣らない検出精度を有していることを確認した．

キーワード：コードクローン，ソフトウェア進化，ソフトウェアリポジトリマイニング

An Empirical Study on Accuracy of Coarse-grained Clone Detection

Keisuke Hotta[1,a)] Jiachen Yang[1,b)] Yoshiki Higo[1,c)] Shinji Kusumoto[1,d)]

Received: May 7, 2014, Accepted: November 10, 2014

Abstract: There exist many state-of-the-art code clone detectors because of much effort of researchers. Researchers adopt clone detectors not only on a single state of a single project but also on multiple projects or multiple revisions, which aims to identify library candidates or to detect inconsistencies of modifications on clones. However, it has been still challenging to detect clones on large corpora of code even with such successful detectors because they require much time to complete clone detection. An alternative of using such detectors will be using a coarse-grained detector. It has high scalability compared to fine-grianed ones, but it is unclear how accurate such a coarse-grained approach is. This paper evaluates the accuary of it compared to seven fine-grained detectors and discusses with a coarse-grained detector implemented by us. The experimental results revealed the high scalability and the high accuracy of the coarse-grained approach.

Keywords: code clones, software evolution, mining software repositories

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，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主たる生成要因であると考えられている．コピーアンドペーストによる既存コードの再利用には，既存機能と類似した機能を高速に，かつ容易に実装することができるという利点がある．そのため，コピーアンドペーストによる既存コードの再利用は多くの開発者によって頻繁に行われており [1]，その結果生じるコードクローンは多くの開発者の関心を集めている [2]．この

[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)] k-hotta@ist.osaka-u.ac.jp
[b)] jc-yang@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp
[d)] kusumoto@ist.osaka-u.ac.jp

ような理由から，コードクローンに関する研究がさかんに行われている [3], [4], [5].

コードクローンに関する研究の中で最も注目を集めているトピックとして，コードクローンの検出技術があげられる．コードクローンの検出技術とは，与えられたソースコード中に存在するコードクローンを自動的に特定する技術を指す．コードクローンの自動検出はコードクローンに関する様々な研究の基盤となる技術である．したがって，コードクローンに関する研究の黎明期から現在に至るまで，コードクローン検出技術に関する研究は多数の研究者によって精力的に行われており，これまでに多数のコードクローン検出手法やコードクローン検出ツールが提案，開発されている [6], [7].

コードクローン検出技術の最たる利用目的として，単一プロジェクトの最新のソースコードに対してコードクローン検出を行い，その結果を品質評価やソースコード修正前のチェックに用いることがあげられる．それに加え，近年，複数プロジェクトを対象にコードクローン分析を行う試みや，開発履歴情報を用いて過去に遡ってコードクローン分析を行う試みが行われている．これにより，ソースコード剽窃の検出 [8] やライブラリ候補の特定 [9]，コードクローンの履歴分析 [10], [11] などが可能となる．

しかし，これらの研究は以下にあげる課題に直面している．

検出に要する時間：一般的に，単一のプロジェクトからのコードクローン検出には数秒から数分程度の時間を要する．したがって，コードクローン検出ツールを単純に大規模なソースコード集合に適用した場合，実用的な時間内に検出を終えることは難しい．

検出されるコードクローンの数：数千行程度の小規模なシステムに対してコードクローン検出ツールを適用した場合でも，すべて手作業で確認することが困難な数のコードクローンが検出される．ゆえに，大規模データセットに対してコードクローン検出ツールを適用すると，さらに膨大な数のコードクローンが報告されると想定され，その検出結果を利用したり，あるいは解析することはきわめて困難であると考えられる．

この問題を解決する方法として，ソースコードに対して簡易的な解析処理のみを行う，粗粒度なコードクローン検出手法を用いることがあげられる．ここでいう “粗粒度なコードクローン検出手法” とは，ソースコードからクラスやメソッド，ブロックなどを抽出し，それらを比較することでコードクローンを検出する技術を指す．これまでに提案されているコードクローン検出手法の多くは，字句単位や文単位などの細かな粒度でソースコードを比較することで，高い精度でのコードクローン検出を目指している．これに対し粗粒度なコードクローン検出手法は，これまでの細粒度な検出手法よりも粗い粒度でソースコードを比較するため，コードクローンの検出に要する時間的コストが大幅に小さいという利点がある．さらに，粗粒度なコードクローン検出手法はクラスやメソッド，ブロックなどの単位でコードクローンを検出するため，細粒度なコードクローン検出手法と比較して検出されるコードクローンの数が小さくなる．このような理由から，粗粒度なコードクローン検出手法は大規模なソースコード集合に対するコードクローン検出に有用であると考えられる．

しかし，粗粒度なコードクローン検出手法がどの程度の精度でコードクローンの検出を行うことができるのかは明らかにされていない．粗粒度なコードクローン検出手法は細粒度のものと比較して簡易的な解析処理しか行っていないため，細粒度なコードクローン検出手法と比較して検出の精度が低い可能性がある．粗粒度なコードクローン検出手法の検出精度が低ければ，粗粒度な手法が持つ利点を考慮に入れても，有用であるとはいい難い．一方，粗粒度なコードクローン検出手法が細粒度なコードクローン検出手法と比較して十分な精度で検出を行うことができれば，粗粒度なコードクローン検出手法は大規模なソースコード集合に対する手法として有用であると考えられる．

そこで本研究では，粗粒度なコードクローン検出ツールを実装し，その検出精度や検出に要する時間を細粒度なコードクローン検出ツールと比較した．その結果，粗粒度なコードクローン検出手法は細粒度なものと比較して，大規模なソースコード集合に対して高速に検出を行うことができ，かつ検出されるコードクローンの数が少ないことを確認した．さらに，適合率と再現率を総合して考えると，粗粒度なコードクローン検出手法の精度は細粒度なものに劣らないことが明らかになった．これらの実験結果から，粗粒度なコードクローン検出手法は大規模ソースコード集合に対するコードクローン検出手法として有用であるという結論を得た．

以降の本稿の構成は以下のとおりである[*1]．2 章で本研究の研究動機である大規模データセットに対するコードクローン検出に触れるとともに，関連研究を紹介する．3 章では本研究における粗粒度なコードクローン検出手法の定義ならびにその実装について説明する．4 章で本研究で調査する調査項目を導入する．5 章で本研究で行う実験の方法を説明し，6 章で実験結果を述べる．7 章で実験結果に対する考察を行い，最後に 8 章で本稿をまとめる．

2. 研究動機

本章では，はじめに本研究の背景となる大規模データセットに対するコードクローン検出技術の適用例について述べる．その後，これまでに行われたコードクローン検出技術の精度評価に関する関連研究に触れる．

[*1] 以降本稿では，コードクローン検出手法のことを**検出手法**と表記し，コードクローン検出ツールのことを**検出ツール**と表記する．

2.1 大規模データセットに対するコードクローン検出

本節では，大規模データセットに対するコードクローン検出技術の適用例として，複数プロジェクトからなるソースコード集合に対する適用事例と，開発履歴を用いた過去のソースコードに対する適用事例を紹介する．

複数プロジェクトに対する検出

Koschke はプロジェクト間のコードクローンを字句単位で高速に検出するための手法を提案している [8]．この手法は，ある 1 つのプロジェクトを入力とし，そのプロジェクトのソースコードとあらかじめ指定されているソースコード集合との間に存在するコードクローンを検出することを目的としている．これにより，あるコード片が他のプロジェクトから流用されたか否かを識別することができると Koschke は述べている．すなわち，このようなプロジェクト間のコードクローンを検出する手法を用いることで，ライセンスに違反した不正なソースコード流用などを特定することが可能になる．

Ishihara らは，複数プロジェクトからなるソースコード集合に対してメソッド単位でのコードクローン検出を行うことで，ライブラリとして再利用可能なメソッドを特定する手法を提案している [9]．この手法では，Java で記述されたプロジェクトの集合を対象とし，その中に存在するメソッドを抽出して比較することで，メソッド全体が重複している箇所を特定する．この手法を用いることで，多くのプロジェクトで共通して記述されている，ライブラリとして整備すべき処理を特定することができる．

また Ishihara らは，複数プロジェクトからなるソースコード集合に対してのコードクローン検出結果を用いることで，ソースコード補完を行う手法を提案している [12]．ソースコード補完とは，利用者が途中まで記述したコードを入力とし，未記述の部分を自動的に補完する技術のことをいう．この手法では，“同じコード片が多数出現するならば，そのコード片は頻繁に再利用されていることを意味し，再利用候補として有力である” という考えに基づき，コードクローンとして多くのプロジェクトに散在するコード片を再利用候補として提示する．このように，複数プロジェクト間のコードクローン検出は，ソースコードの再利用支援技術としても用いることができる．

開発履歴を遡った検出

開発履歴情報を用い，過去に遡ってコードクローンを検出する大きな目的として，コードクローンの履歴分析があげられる．これらの研究の目標は，“コードクローンがソフトウェアの開発・保守に悪影響を与えているのか” を定量的に明らかにすることである．これらの調査結果は，どの程度のコストをかけてコードクローンを管理すべきかを検討する上で重要な指標となりうる．

これまでに多数の研究者によって調査が行われてきたが，コードクローンがソフトウェア開発・保守に悪影響を与えているか否かについては，調査結果が分かれている．コードクローンがソフトウェア開発・保守を阻害していると主張する報告もあれば [10]，必ずしもそうとはいえないとする調査報告もある [11]．これらの調査結果を総合すると，すべてのコードクローンではなく，一部のコードクローンがソフトウェア開発・保守を阻害している可能性が高いといえる．したがって今後，どのようなコードクローンがソフトウェア開発・保守を阻害するのかという点に関して研究が進められていくと考えられる [5]．開発履歴を遡ったコードクローン検出は，この研究が今後進められるうえで重要な役割を担う基盤的技術である．

また，過去に遡ったコードクローン検出のこのほかの利用用途として，修正漏れの検知があげられる．現時点に存在するコードクローンが，過去にどのような変遷をたどってきたのかを明らかにすることで，生成された時点ではコードクローンに含まれていたが，その後コードクローンとは見なされなくなったコード片が存在するか否かを知ることができる．そのようなコード片を分析することで，修正漏れがなかったかを確認することが可能になる．このように，コードクローンが過去にどのような変遷をたどってきたのかを特定する技術はコードクローンの追跡と呼ばれ，これまでにいくつかの手法が提案されている [13], [14], [15]．

2.2 関連研究

Bellon らは 8 つのソフトウェアを対象とし，6 つの検出ツールを用いた精度比較を行った [16]．その結果，トークン単位の検出手法が高い再現率を有することや，抽象構文木の比較に基づく検出手法が高い適合率を有することを明らかにした．また Bellon らは実験に用いたデータセットをすべて公開しており，そのデータはコードクローン検出ツールの精度評価を行う際に広く用いられている．

Bellon らの研究以外にコードクローン検出手法の精度を評価した実験として，Roy らの実験がある [17]．Bellon らの実験ではコードクローンを目視で確認して正解か否かを判別しているが，Roy らの手法はコードクローンを構成するコード片に対して変数名の変更や文の挿入などの修正を加えたものを作成し，それを検出できるか否かを評価することで，検出精度の評価を行っている．

3. 粗粒度な検出手法

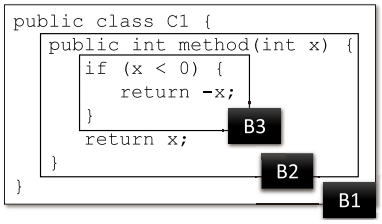
本章では，本研究における粗粒度な検出手法の定義を与えるとともに，細粒度な検出手法と比較した長所，短所を述べる．さらに，本研究で用いる粗粒度な検出ツールについても言及する．

3.1 定義

粗粒度な検出手法とは，ソースコードをファイルやメソッド，ブロックなどの粗い単位で比較することでコー

```
public class C1 {
   public int method(int x) {
      if (x < 0) {
         return -x;
      }
      return x;
   }
}
```

(a) ソースコード



(b) ブロック

図 1 ブロックの例
Fig. 1 Example of blocks.

```
 1: List<Employee> employees =
 2:                     new ArrayList<Employee>();
 3:
 4: String line;
 5: while ((line = br.readLine()) != null) {
 6:     String[] splitLine = line.split(",");
 7:     int id = Integer.parseInt(splitLine[0]);
 8:     String name = splitLine[1];
 9:     int age = Integer.parseInt(splitLine[2]);
10:
11:     employees.add(new Employee(id, name, age));
12: }
```

(a) コード片 1

```
 1: List<Employee> employees =
 2:                     new ArrayList<Employee>();
 3:
 4: String line;
 5: while ((line = br.readLine()) != null) {
 6:     String[] splitLine = line.split(",");
 7:     int id = Integer.parseInt(splitLine[0]);
 8:     String name = splitLine[1];
 9:     int age = Integer.parseInt(splitLine[2]);
10:     String department = splitLine[3];
11:
12:     employees.add(new Employee(id, name,
13:                             age, department));
14: }
```

(b) コード片 2

図 2 粗粒度な検出手法では検出できないコードクローンの例
Fig. 2 Example of code clones that coarse-grained detector cannot detect.

ドクローンを検出する手法のことを指す．本研究では，粗粒度な検出手法を，“ブロック単位でコードクローン検出を行う手法” と定義する．ここでいう “ブロック” とは，1つ以上の文の集合のことを指し，クラスやファイル，メソッド，関数，および `if` 文などのブロック文が該当する．たとえば**図 1** の例の場合，図 1 (a) のソースコードには，図 1 (b) に示すように，クラス `C1`（ブロック B1），メソッド `method(int x)`（ブロック B2），ならびに `if` 文（ブロック B3）の 3 つのブロックが存在する．

本研究における粗粒度な検出手法では，ソースコード中に存在するブロックを抽出し，それらを互いに比較することで，類似した文字列を持つブロックどうしをコードクローンとして検出する．

3.2 細粒度な検出手法との比較

細粒度な検出手法は，ソースコードをテキストとして比較，あるいはトークン列や木構造に変換した後にそれらを用いて比較することで，コードクローンを検出する手法である．これに対し，粗粒度な検出手法はソースコード中のブロック単位で比較を行う手法である．ソースコードをテキストで比較する場合は文字単位での比較が必要であり，トークン列や木構造での比較であればトークン単位，あるいは頂点単位での比較が必要である．一般的にブロックの数は，ソースコードの文字数やトークン数，文の数，木構造の頂点数などと比較して小さい．したがって，粗粒度な検出手法は細粒度なものと比較して高速にコードクローンの検出を行うことができると期待できる．

一方，粗粒度な検出手法はブロック単位でソースコードを比較するため，2 つのブロックの一部のみが類似している場合に，それらをコードクローンとして検出することができないという欠点がある．**図 2** に，粗粒度な検出手法では検出できないコードクローンの例を示す．この例の場合，図中の 2 つのコード片には類似した `while` 文が存在しているが，図 2 (b) のコード片の 10 行目に図 2 (a) のコード片にない処理が追加されている．これにより，粗粒度な検出手法はこれらの `while` 文をコードクローンとして検出することはできない．

3.3 実装

本研究では，粗粒度な検出手法をツールとして実装し，それを用いて精度ならびに検出時間の評価を行った．ここでは，実装した粗粒度な検出ツールについて簡単に説明する．

概要

実装したツールは Java で記述されたソースコードを入力として受け取り，与えられたソースコード中に存在するブロック単位のコードクローンを出力する．このツールは，与えられたソースコード全体に対するコードクローン検出に加え，増分的なコードクローン検出にも対応している．増分的なコードクローン検出とは，以前にコードクローン検出が行われた時点以降に修正が加えられたファイルのみを解析し，修正されていないファイルについては以前に行われたコードクローン検出の結果を再利用することで，2 回目以降のコードクローン検出に要する時間を大幅に低減する手法である [18], [19]．これを用いることで，開発履歴情報を分析する際などに，短時間でコードクローンの検出を行うことができる．

処理の手順

実装したツールが行う処理は以下のとおりである．

STEP1：与えられたソースコードを解析し，ソースコード中に存在するブロックを特定する．

STEP2：STEP1 で特定されたブロックを正規化する．この正規化では，各ブロックをあらかじめ定められたフォーマットに整形するとともに，変数名やリテラルを特殊文字に置き換える．これにより，2 つのブロックのフォーマットのみが異なる場合や，変数名やリテラルのみが異なる場合でも．それらをコードクローンとして検出することが可能になる．

STEP3：各ブロックについて，正規化後の文字列からハッシュ値を算出する．今回の実装では，ハッシュ関数として Java の String クラスに定義されている `hashCode()` メソッドを使用した．

STEP4：STEP3 で得られたハッシュ値を比較し，同じハッシュ値を持つブロックどうしを互いにコードクローン関係にあるブロックとして検出する．ハッシュ値を用いた比較を行うことで，文字列を用いた比較を行う場合と比べて，2 つのブロックの比較に要する計算コストを小さくすることができる．

増分的な検出の手順

増分的なコードクローン検出を行う際の手順は以下のとおりである．

(1) 以前のコードクローン検出結果をもとに，以前のソースコードに含まれていたブロックに関する情報を復元する．以降，復元されたブロックからなる集合を B とする．

(2) 以前にコードクローン検出を行った時点以降に修正された，もしくは削除されたファイルについて，それらのファイルに含まれるブロックを B から削除する．

(3) 以前にコードクローン検出を行った時点以降に追加された，もしくは修正されたファイルについて STEP1 から STEP3 の処理を行い，新たに特定されたブロックを B に追加する．

(4) B に含まれるブロックを用いて STEP4 を実施し，コードクローンを検出する

適用例

図 3 に実装した検出ツールの適用例を示す．図上部のソースコードに対して STEP1 から STEP4 までの処理を行うと，2 つのブロック B3 と B6 の対がコードクローンとして検出される．なお，B3 と B6 は内部で使用されている変数名が異なるが，STEP2 の正規化処理を行ったことで，コードクローンとして検出されている．

4. 調査項目

本研究では，以下の 3 つの調査項目を設定する．

RQ1：粗粒度な検出手法は大規模なソースコード集合に対して高速にコードクローンの検出を行うことができるか？

RQ2：粗粒度な検出手法は細粒度な検出手法と比較してより少ない数のコードクローンを検出するか？

RQ3：粗粒度な検出手法による検出結果は，細粒度な検出手法によるものに劣らない精度であるか？

RQ1 は粗粒度な検出手法の検出速度に主眼を置いた調査項目であり，複数プロジェクトからなるソースコード集合と開発履歴情報を対象として調査する．粗粒度な検出手法は細粒度な検出手法と比較して高速であるという利点があると考えられる．そこで，この調査項目では，粗粒度な検出手法が細粒度なものと比較して高速であることを定量的に確認することを目指す．

RQ2 は粗粒度な検出手法が検出するコードクローンの数に着目した調査項目である．粗粒度な検出手法を用いる利点として，検出されるコードクローンの数が少なく，検出結果の分析に要するコストが小さくなるという点があげられる．したがって，この調査項目はこの点を定量的に確認することを目指す．

RQ3 は粗粒度な検出手法の検出精度を確認するための調査項目である．粗粒度な検出手法は細粒度な検出手法と比較して高速であり，かつ検出されるコードクローンの数が減少することで分析に要するコストが小さくなるという利点が期待される．その一方で，粗粒度な検出手法は細粒度な検出手法と比較して検出精度が劣ることが危惧される．そこでこの調査項目では，粗粒度な検出手法の検出精度を細粒度な検出手法によるものと比較し，粗粒度な検出手法の検出精度がどの程度なのかを定量的に明らかにすることを目指す．

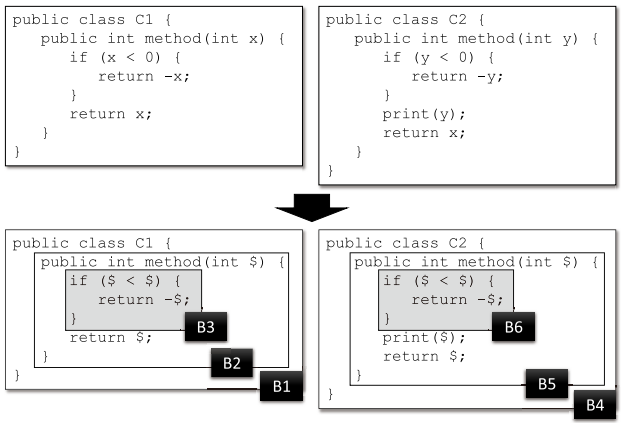


図 3 粗粒度な検出ツールの適用例

Fig. 3 How the coarse-grained detector works.

5. 実験の設定

本章では，本研究で行う実験における検出ツールの精度評価の方法を述べた後，実験環境，対象ソフトウェア，ならびに比較対象となる細粒度な検出ツールについて述べる．

5.1 検出精度の評価方法

用いるベンチマークの概要

本研究では，検出ツールの精度を評価するために，Bellon らのベンチマーク [16] を使用する．Bellon らのベンチマークは，C もしくは Java で記述された 8 つのオープンソースソフトウェアについて，正解となるコードクローンの集合を定義している．Bellon らは上述の 8 つのソフトウェアに複数の検出ツールを適用し，得られたコードクローンのうちランダムに選択された 2%を目視で確認することで，正解となるコードクローンを定義している．

なお，Bellon らのベンチマークは，互いにコードクローン関係にあるコード片の対（以降，クローンペア）を扱っている．したがって，本研究の実験においても，クローンペアを用いた精度評価を行う．

以降本稿では，以下の 2 つの用語を使用する．

正解クローン：Bellon らによって正解と判断されたクローンペア

候補クローン：検出ツールによって検出されたクローンペア

それぞれの検出ツールの精度を評価するために，各検出ツールが報告した候補クローンと正解クローンの対応付けを行い，各検出ツールが正解クローンをどれだけ検出できたかを計測する必要がある．本研究では，この対応付けに OK 値を用いる．OK 値は候補クローンと正解クローンがどの程度一致するかを表すメトリクス値であり，Bellon らの研究において定義されている．この数値が閾値以上であれば，その候補クローンがその正解クローンに対応すると判断される．換言すれば，その検出ツールがその正解クローンを検出することができたと見なされる．

指標の定義

ここでは，正解クローンと候補クローンの対応付けに用いる OK 値の定義，ならびに精度評価に用いる適合率，再現率の定義を述べる．以降，CP_r を正解クローン，CP_c を候補クローンとし，$CP.CF1$，$CP.CF2$ をクローンペア CP を構成するそれぞれのコード片とする．

はじめに，任意の 2 つのコード片の間に順序関係を定義する．$CF.file$ をコード片 CF が存在するファイル名，$CF.start$，$CF.end$ をそれぞれコード片 CF の開始行番号，終了行番号とする．また，2 つの文字列の順序関係は辞書式順序に基づいて決定されるものとする．すなわち，ファイル名 $CF1.file$ とファイル名 $CF2.file$ を辞書式に比較したときに，$CF1.file$ の方が小さいければ，$CF1.file < CF2.file$ が真となる．

このとき，2 つのコード片 $CF1$，$CF2$ の順序関係を以下に定義する．

$$\begin{aligned} CF1 < CF2 \Leftrightarrow ((&CF1.file < CF2.file) \vee \\ &(CF1.file = CF2.file \wedge \\ &CF1.start < CF2.start) \vee \\ &(CF1.file = CF2.file \wedge \\ &CF1.start = CF2.start \wedge \\ &CF1.end < CF2.end)) \end{aligned} \tag{1}$$

以降，すべてのクローンペア CP について，$CP.CF1 < CP.CF2$ が成立するものとする．

次に，OK 値の定義に必要となる数値の定義を与える．$lines(CF)$ をコード片 CF を構成する行の集合であるとするとき，$contained(CF1, CF2)$ を以下に定義する．

$$contained(CF1, CF2) = \frac{|lines(CF1) \cap lines(CF2)|}{|lines(CF1)|} \tag{2}$$

これらの定義を用い，正解クローン CP_r と候補クローン CP_c との間の OK 値を以下に定義する．

$$\begin{aligned} OK(CP_r,CP_c) = min(& \\ & max(contained(CP_r.CF1, CP_c.CF1), \\ & contained(CP_c.CF1, CP_r.CF1)), \\ & max(contained(CP_r.CF2, CP_c.CF2), \\ & contained(CP_c.CF2, CP_r.CF2))) \end{aligned} \tag{3}$$

$OK(CP_r, CP_c)$ が閾値以上のとき，候補クローン CP_c は正解クローン CP_r に対応するとみなされる．本研究では，閾値として Bellon らの研究で用いられている値と同じ 0.7 を用いる．

次に，対象ソフトウェア P における検出ツール T の検出精度の評価指標である適合率 ($precision(P,T)$) ならびに再現率 ($recall(P,T)$) を定義する．$DetectedRefs(P,T)$ を T が P から検出できた正解クローンの集合，$Cands(P,T)$ を T が P から検出した候補クローンの集合，$Refs(P)$ を P に存在する正解クローンの集合であるとする．このとき，$precision(P,T)$，ならびに $recall(P,T)$ を以下に定義する．

$$precision(P, T) = \frac{|DetectedRefs(P,T)|}{|Cands(P,T)|} \tag{4}$$

$$recall(P, T) = \frac{|DetectedRefs(P,T)|}{|Refs(P)|} \tag{5}$$

$precision(P,T)$ は T が検出した候補クローンのうち正解であったものの割合を示しており，この値が高いほど誤検出が少ないことを意味している．一方，$recall(P,T)$ はすべての正解クローンのうち T が検出できたものの割合を示しており，この値が高いほど検出漏れが少ないことを意味している．

5.2 実験環境

本実験はコア数 16 の 64 ビット CPU を 2 つ備え，128 GB の RAM を搭載した HP 社製のワークステーション Z820

表 1 UCI Dataset の概要
Table 1 Overview of UCI Dataset.

ソースファイル数	2,092,739
プロジェクト数	13,193
ソースコード行数	373,500,402

表 2 DNSJava リポジトリの概要
Table 2 Overview of repository of DNSJava.

最新リビジョンのリビジョン番号	1,670
ソースファイルに変更があったコミットの数	1,432
最新リビジョンのソースファイル数	7,362
最新リビジョンのソースコード行数	1,237,336

表 3 実験対象ソフトウェア（for RQ2, RQ3, and RQ4）
Table 3 Target software systems.

ソフトウェア名	短縮名	LOC	正解クローン数
eclipse-ant	ant	34,744	30
eclipse-jdtcore	jdtcore	147,634	1,345
j2sdk1.4.0-javax-swing	swing	204,037	777
netbeans-javadoc	netbeans	14,360	55

上で行った．また実験に用いたデータセットはすべて SSD 上に配置した．

各検出ツールを用いてコードクローンの検出を行う際，コードクローンとして検出するコード片の長さの下限を 6 行とした．これは Bellon らの実験で用いられている設定と同じである．

5.3 対象ソフトウェア

本研究では，RQ1 に対する実験と RQ2，RQ3 に対する実験に異なる対象ソフトウェアを用いる．RQ1 に対する実験では，粗粒度な検出手法の検出速度を評価するため，大規模なデータセットを実験対象とする必要がある．この実験では，複数プロジェクトからなるデータセットである UCI Dataset[*2]，ならびに DNSJava の開発履歴が蓄積されたソースコードリポジトリ[*3]を実験対象とする．UCI Dataset の概要を**表 1** に，DNSJava リポジトリの概要を**表 2** にそれぞれ示す．

一方，RQ2，RQ3 に対する実験では，Bellon らの実験で用いられているソフトウェアを実験対象とする．実験対象となるソフトウェアの概要を**表 3** に示す．なお，本研究で用いる粗粒度な検出ツールが Java のみを解析対象としているため，この実験では Bellon らの実験で用いられているソフトウェアのうち Java で記述されたもののみを対象とする．

5.4 比較対象として用いる検出ツール

表 4 に本研究で用いる細粒度な検出ツールの一覧を示す．本研究では Bellon らの実験で用いられた検出ツールに加え，Bellon らの実験以後に開発されたいくつかの検出ツールを比較対象とした．

[*2] http://www.ics.uci.edu/~lopes/datasets/
[*3] http://sourceforge.net/projects/dnsjava/

表 4 比較対象となる検出ツール
Table 4 Target clone detectors.

ツール名	分類
Dup [20]	字句単位の検出
CloneDR [21]	抽象構文木を用いた検出
CCFinder [22]	字句単位の検出
Duploc [23]	行単位の検出
Deckard [24]	抽象構文木を用いた検出
CDSW [25]	文単位の検出
LD (based on [26])	行単位の検出

このうち LD は，Hummel らの論文をもとに著者らが実装したツールである．Hummel らの手法は高速な検出手法として知られており，増分的なコードクローン検出にも対応している．したがって，本研究では粗粒度な検出手法との速度比較を主たる目的として，この検出手法を比較対象に加えた．

6. 実験結果

本章ではそれぞれの調査課題に対する実験結果を述べるとともに，それぞれの調査項目に回答する．

なお本節では，本研究で実装した粗粒度な検出ツールを Block-based と表記する．また，Dup，CloneDR，CCFinder，ならびに Duploc に対する実験結果については，Bellon らの実験 [16] と同じ結果となる．したがって，本実験ではこれらの検出ツールの適用は行わず，Bellon らの報告で記載されている数値を表記している．

6.1 RQ1：粗粒度な検出手法は大規模なソースコード集合に対して高速にコードクローンの検出を行うことができるか？

UCI Dataset ならびに DNSJava リポジトリに対して粗粒度な検出手法と LD を適用し，検出に要する時間を比較した．LD を比較対象とした理由は，本実験で比較対象とした検出ツールの中で最も短時間でコードクローンの検出を行うことができると期待できるためである．その他の細粒度な検出ツールを用いた場合，今回実験対象とした大規模なソースコード集合に対して現実的な時間内で検出を終えることができない可能性が高い．

たとえば，CCFinder はこれらの検出ツールの中で比較的高速であることが知られている [16]．しかし，石原らの試算によると，CCFinder を UCI Dataset に適用した場合，検出に要する時間は約 950 日と推定されている [27]．また，CCFinder を DNSJava の最新リビジョンに対して適用したところ，検出に約 26 分を要した．今回用いた検出ツールのうち，LD と Block-based 以外の 6 つの検出ツールは，増分

的なコードクローン検出に対応していない．したがって，DNSJava リポジトリに対してコードクローン検出を行うためには，リビジョンの数と同じ回数，すなわち 1,432 回検出ツールを適用しなければならない．比較的高速な検出ツールである CCFinder が最新リビジョンからのコードクローン検出に約 26 分を要したことを勘案すれば，LD を除く比較対象を用いた場合に DNSJava リポジトリに対して現実的な時間内で検出を終えることは難しいといえる．

このような理由から，今回の実験では LD のみを比較の対象として用いた．なお，前述のように，粗粒度な検出ツールならびに LD はいずれも増分的なコードクローン検出に対応している．よって DNSJava リポジトリに対する実験では，増分的なコードクローン検出を適用した．

実験結果を**表 5** に示す．なお，UCI Dataset に対して LD を適用したところ，24 時間を超えても検出が完了しなかったため，“N/A” と表記している．表 5 より，どちらの実験対象に対しても，粗粒度な検出手法は短時間で検出を終えていることが分かる．

したがって，RQ1 には **Yes** と回答する．

6.2 RQ2：粗粒度な検出手法は細粒度な検出手法と比較してより少ない数のコードクローンを検出するか？

表 3 に示した 4 つのソフトウェアに対し，各検出ツールを適用し，検出された候補クローンの数を比較した．

実験結果を**表 6** に示す．なお，Duploc は swing に対してコードクローンの検出を完了することができなかったため，該当箇所に “N/A” と記述されている．表 6 より，粗粒度な検出手法が検出した候補クローンの数は，CloneDR と CDSW を除くすべての検出ツールについて，すべての実験対象で小さくなっていることが分かる．また，CDSW と比較すると，粗粒度な検出手法は 2 つの実験対象について少ない数の候補クローンを検出している．

これらの結果より，粗粒度な検出手法は細粒度な検出手法と比較して少ない数の候補クローンを検出する傾向にあるといえる．ゆえに，RQ2 に対する回答は **Yes** となる．

6.3 RQ3：粗粒度な検出手法による検出結果は，細粒度な検出手法によるものに劣らない精度であるか？

各検出ツールの検出結果の適合率を**図 4** に，再現率を**図 5** にそれぞれ示す．なお，いずれも各実験対象における右端の黒い棒グラフが粗粒度な検出ツールの値を示している．

図 4 より，粗粒度な検出手法は jdtcore においてほかのすべての検出ツールよりも高い適合率を有しており，そのほかの実験対象については 2 番目に高い値を有している．したがって，粗粒度な検出手法は高い適合率を有していることが分かる．

一方，図 5 より，粗粒度な検出手法の再現率は最も良い場合（jdctore と swing）で 8 つの検出ツールの中で 5 番目に高い数値となっている．この結果は，粗粒度な検出手法の再現率は必ずしも高くないことを示しているといえる．

一般的に検出ツールの精度評価において，適合率と再現率はトレードオフの関係にあることが知られている [16]．ゆえに，適合率と再現率を総合して検出精度を評価する必要がある．

そこで，適合率と再現率の調和平均である F 値を計測し，比較を行った．ある実験対象ソフトウェア P におけ

表 5 検出速度の比較
Table 5 Comparison of time required for detection.

	LD	Block-based
UCI dataset (multiple projects)	N/A	152 [m]
DNSJava (multiple revisions)	212 [m]	17 [m]

表 6 検出された候補クローンの数
Table 6 Number of clone candidates.

Tool	ant	jdtcore	swing	netbeans
Dup	245	11,589	7,220	344
CloneDR	42	3,593	3,766	33
CCFinder	950	26,049	21,421	5,552
Duploc	162	710	N/A	223
Deckard	319	22,353	25,415	414
CDSW	222	5,247	1,703	1,344
LD	662	44,294	19,253	1,360
Block-based	70	5,398	3,922	57

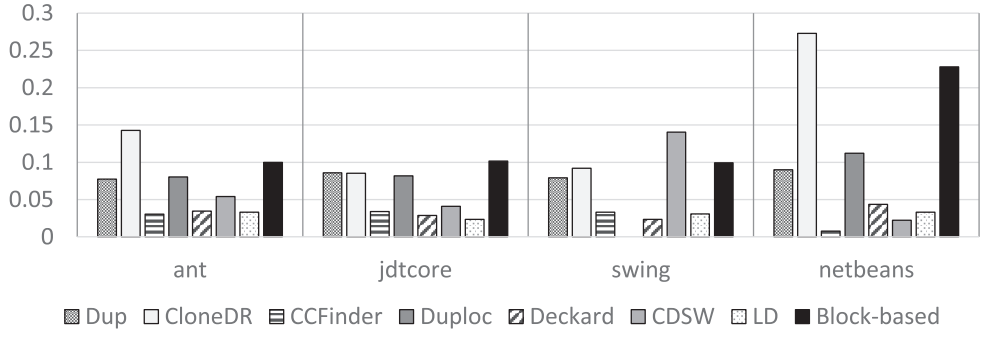


図 4 適合率
Fig. 4 Precision.

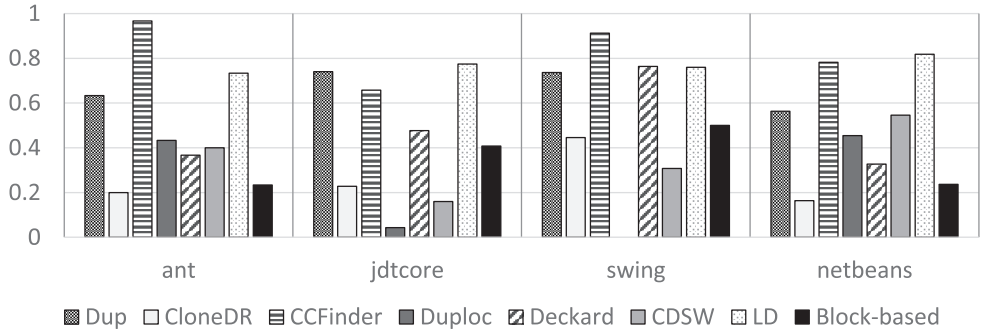


図 5 再現率
Fig. 5 Recall.

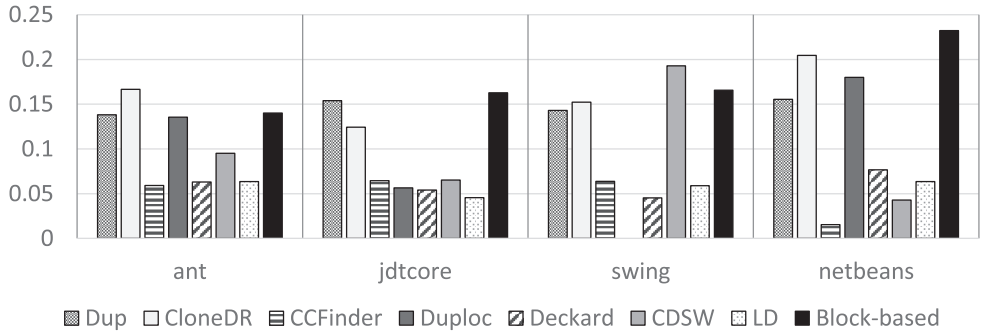


図 6 F 値
Fig. 6 F-measure.

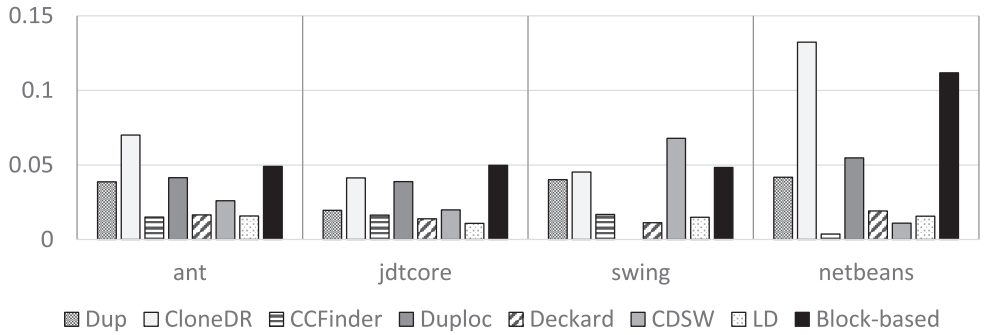


図 7 AUP
Fig. 7 AUP.

る，検出ツール T の F 値 ($F(P,T)$) は以下のように定義される．

$$F(P,T) = \frac{2 * precision(P,T) * recall(P,T)}{precision(P,T) + recall(P,T)} \qquad (6)$$

各検出ツールの F 値を**図 6** に示す．図 6 より，粗粒度な検出手法は 2 つの対象ソフトウェア（jdtcore と netbeans）において他のどの検出ツールよりも高い F 値を記録し，また残りの 2 つの対象ソフトウェアについては 2 番目に高い F 値を有していることが分かる．この結果は，適合率と再現率を総合的に考えたときに，粗粒度な検出手法が高い検出精度を有していることを示している．

これらの結果から，粗粒度な検出手法の検出精度は細粒度な検出手法によるものに劣らないといえる．しかし上述の結果は，OK 値の閾値を 0.7 に設定した場合の結果であり，異なる閾値を用いた場合に同様の結果が得られるとは必ずしもいえない．そこで，OK 値の閾値に依存しない以下の指標を定義し，それを用いた評価を行った．

- AUP（Area Under the Precision curve）：横軸に OK 値，縦軸に適合率を取ったグラフの曲線下部の面積
- AUR（Area Under the Recall curve）：横軸に OK 値，縦軸に再現率を取ったグラフの曲線下部の面積
- AUF（Area Under the F-measure curve）：横軸に OK 値，縦軸に F 値を取ったグラフの曲線下部の面積

本実験では，OK 値を 0.5 から 1.0 までの間で 0.01 刻みで変化させ，上記の値をそれぞれ算出した．OK 値を低く設定しすぎると，検出ツールが報告したクローンペアと正解クローンの位置が大きくずれている場合であっても，わずかでも位置が重なっていればその正解クローンを検出で

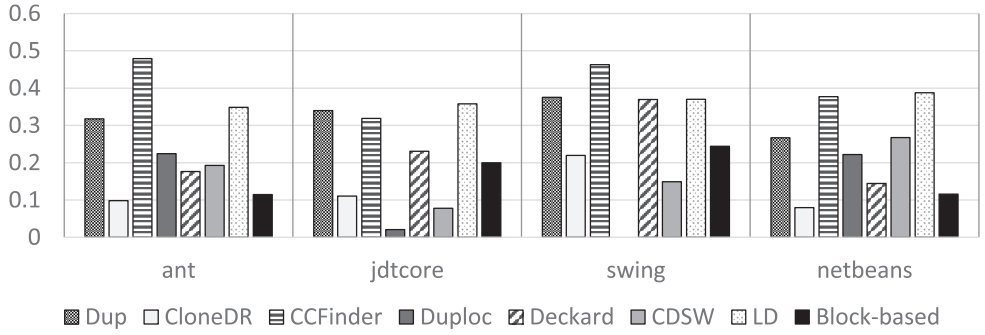


図 8 AUR
Fig. 8 AUR.

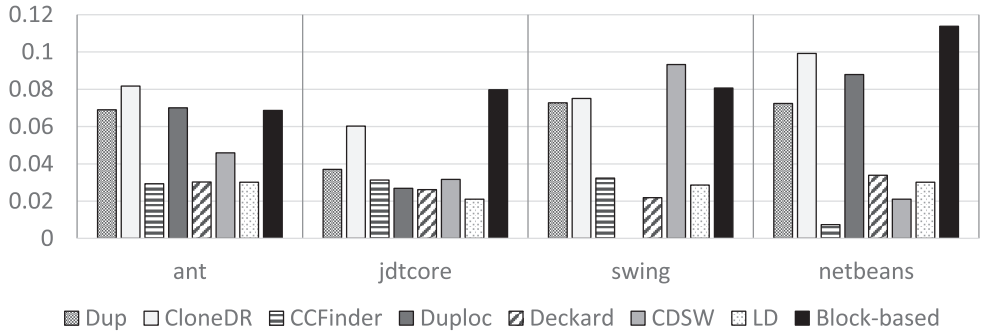


図 9 AUF
Fig. 9 AUF.

きたと判定される．したがって，*OK* 値の閾値に下限を設定した上で，それぞれの指標を算出した．

評価結果を**図 7**，**図 8**，ならびに**図 9** に示す．これらと図 4，図 5，図 6 を比較すると，*F* 値に関する指標の ant に対する結果において順位の変動がみられるが，それ以外の箇所については *OK* 値の閾値を 0.7 に固定した場合と同様の結果が得られた．したがって，粗粒度な検出手法は *OK* 値の決め方によらず，細粒度な検出手法に劣らない検出精度を有しているといえる．

以上の結果から，RQ3 には **Yes** と回答する．

7. 考察

7.1 検出できた例と検出できなかった例

図 10 に粗粒度な検出手法を用いて検出することができた正解クローンの例を示す．図中で薄くハイライトされている箇所は正解クローンを表している．また，その上で濃くハイライトされている箇所は，粗粒度な検出手法がクローンペアとして検出したブロックを示している．この例では，粗粒度な検出手法は正解クローンの先頭の 1 行と末尾の 1 行をコードクローンとして検出できていないが，*OK* 値が閾値を超えているため，粗粒度な検出手法はこの正解クローンを検出できたと判定された．

一方，**図 11** に粗粒度な検出手法が検出できなかった正解クローンの例を示す．この例では，2 行からなる小さな `if` 文が 3 つ連続して出現している．粗粒度な検出手法を用いた場合，それぞれの `if` 文の大きさが本実験で用いた閾値である 6 行を下回っているため，それぞれの `if` 文はコードクローンとして検出されない．したがって，粗粒度

```
public void execute() throws BuildException {
  ...
  do {
    ...
    if (src.lastModified() > now) {
      log("Warning: "+sourceFiles[i]+" modified in the future.",
          Project.MSG_WARN); }
    Enumeration enumTargets = allTargets.elements();
    while (upToDate && enumTargets.hasMoreElements()) {
      File dest = (File)enumTargets.nextElement();
      if (src.lastModified() > dest.lastModified()) {
        log(dest.getPath() + "is out of date with respect to " +
            sourceFiles[i], Project.MSG_VERBOSE);
        upToDate = false; } }
  } while (upToDate && (++i < sourceFiles.length)); } }
  ...
```

(a) コード片 1

```
public void execute() throws BuildException {
  ...
  do {
    ...
    if (!src.exists()) {
      log(sourceFiles[i]+ " does not exist.", Project.MSG_VERBOSE);
      upToDate = false;
      break; }
    Enumeration enumTargets = allTargets.elements();
    while (upToDate && enumTargets.hasMoreElements()) {
      File dest = (File)enumTargets.nextElement();
      if (src.lastModified() > dest.lastModified()) {
        log(dest.getPath() + "is out of date with respect to " +
            sourceFiles[i], Project.MSG_VERBOSE);
        upToDate = false; } }
  } while (upToDate && (++i < sourceFiles.length)); } }
  ...
```

(b) コード片 2

図 10 粗粒度な検出手法が検出できた正解クローンの例
Fig. 10 Clone reference detected by coarse-grained detector.

```
public boolean execute() throws BuildException {
   attributes.log("Using jikes compiler", Project.MSG_VERBOSE);
   ...
   if (debug) {
      cmd.createArgument().setValue("-g"); }
   if (optimize) {
      cmd.createArgument().setValue("-O"); }
   if (verbose) {
      cmd.createArgument().setValue("-verbose") }
   ...
```

(a) コード片 1

```
public boolean execute() throws BuildException {
   attributes.log("Using jvc compiler", Project.MSG_VERBOSE);
   ...
   if (debug) {
      cmd.createArgument().setValue("-g"); }
   if (optimize) {
      cmd.createArgument().setValue("-O"); }
   if (verbose) {
      cmd.createArgument().setValue("-verbose") }
   ...
```

(b) コード片 2

図 11 粗粒度な検出手法が検出できなかった正解クローンの例
Fig. 11 Clone reference not detected by coarse-grained detector.

な検出手法はこの正解クローンを検出することができなかった．このような連続した小さなブロックで構成される正解クローンが多数存在した．

このような正解クローンを検出する方策として，コードクローンとして検出するブロックの最小行数を小さく設定してコードクローン検出を行った後，ソースコード中で連続しているブロックがともにコードクローンとして検出されている場合，それらを連結してコードクローンと見なす，という改良を加えることがあげられる．本研究では単純なアルゴリズムを用いてコードクローン検出を行う手法を用いて実験を行ったが，このように粗粒度な検出手法に簡単な改良を加えることで検出可能となる正解クローンが多数存在することが確認された．したがって，粗粒度な検出手法に様々な改良を加えることで，高速に検出を終えることができるという利点を活かしつつ，その検出精度を向上させられると期待できる．

また，今回の実験の結果，粗粒度な検出手法は適合率が高く，再現率がやや低い傾向にあることが分かった．今回用いた粗粒度な検出手法で検出されるコードクローンはブロック単位である．すなわち，一定以上の大きさを持つブロックが類似している場合に，コードクローンとして検出される．ブロックというまとまった箇所が類似しているため，人が目で見たときにコードクローンであると判定される可能性が高くなっていると考えられる．また，本実験で調査したように，粗粒度な検出手法を用いると検出されるコードクローンの数が少なくなる．ゆえに，式 (4) の分母の値が小さくなる．このような理由から，粗粒度な検出手法の適合率が高くなったと考えられる．一方，再現率がやや低くなった理由は，細粒度な検出手法を用いた場合に検出できるコードクローンを，粗粒度な検出手法が検出できなかったことに起因する．ただし，上述のような改良を加えることで，今回検出できなかったいくつかのコードクローンを検出できるようになると期待できる．

表 7 ブロックの種類
Table 7 Types of blocks.

	ant	jdtcore	swing	netbeans
class	2	10	89	14
method	23	2,421	3,686	19
catch	1	43	1	0
do	0	3	0	0
else	2	202	16	1
finally	11	3	0	0
for	2	77	5	0
if	24	2,475	117	22
switch	0	138	7	0
synchronized	0	0	0	0
try	0	8	0	0
while	5	18	1	1
計	70	5,398	3,922	57

7.2 コードクローンとして検出されたブロックの種類

表 7 に粗粒度な検出手法で検出されたクローンペアのブロックの種類ごとの内訳を示す．表より，いずれのソフトウェアについてもメソッドと `if` 文が多数クローンペアとして検出されていることが分かる．特に swing についてはメソッドが大多数を占めていることが分かる．

7.3 結果の妥当性について

正解クローン集合の構築

本研究では，検出ツールの精度を評価するために Bellon らのベンチマークを用いた．しかし，Bellon らは正解クローン集合を構築する際，実験対象ソフトウェアから得られたクローンペアのうち 2%をランダムに抽出し，それらに対して正解か否かの判別を行っている．したがって，Bellon らのベンチマークでは，誤検出と見なされたクローンペアが実際には誤検出ではない可能性がある．ゆえに，適合率の値は本来の値よりも低い値となっている可能性がある．

しかし，Bellon らは判別対象のクローンペアをランダムに抽出しているため，抽出の際の偏りはないものと考えられる．したがって，今回の実験で行った比較は公平なものであるといえる．

実験対象ソフトウェア

本研究では，Java で記述されたオープンソースソフトウェアのみを実験対象としている．したがって，ほかの言語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場合，本研究で得られた結果と異なる結果が導かれる可能性がある．

ハッシュ値の衝突

今回の実験で用いた粗粒度な検出ツールは，2 つのブロッ

クを比較する際に，それぞれのブロックを構成する文字列から算出されたハッシュ値を用いている．したがって，ハッシュ値の衝突が起きた場合，本来はコードクローン関係にはないブロックをコードクローン関係にあると誤認識するおそれがある．

ただし，Keivanloo らは，本実験で用いた UCI Dataset に対して本研究で用いたものと同じハッシュ関数を用いて行った実験の結果，このハッシュ関数がこのような大規模なデータセットに対して十分な安全性を持っていると報告している [28]．

粗粒度な検出手法の実装

本研究では，粗粒度な検出手法を著者らが実装し，それを用いて実験を行った．検出の際に用いたアルゴリズムは，ブロックの文字列からハッシュ値を算出して比較するという単純なものである．このため，検出精度を高めるための工夫を加えるなどして実装された，異なる粗粒度な検出ツールを適用した場合，本研究で得られた結果とは異なる結果が得られる可能性がある．ただし，本研究で用いたアルゴリズムは最も基本的なものであるため，他の粗粒度な検出ツールを適用した場合に，粗粒度な検出手法の精度が著しく低くなる可能性は小さい．

8. あとがき

本研究では，ソースコードに対する簡易的な解析のみを用いてコードクローンを検出する粗粒度なコードクローン検出手法の精度や検出に要する時間を評価した．ソースコードのブロック単位でコードクローンを検出するツールを実装し，7 つの細粒度なコードクローン検出手法と比較することで評価を行った．実験の結果，粗粒度なコードクローン検出手法は大規模なソースコード集合から短時間でコードクローンを検出することができることが確認されたとともに，粗粒度な手法が高い検出精度を有していることが明らかになった．

この結果から，粗粒度なコードクローン検出手法は大規模なソースコード集合に対するコードクローン検出手法として有用であるという結論を得た．

本研究の今後の課題として，粗粒度なコードクローン検出手法にどのような改良を加えることで検出精度を向上させられるのかを検討することがあげられる．それらを実装することで，検出速度が高速であるという粗粒度な検出手法の利点を活かしつつ，検出精度のさらなる向上が期待できる．

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（S）（課題番号：25220003），挑戦的萌芽研究（課題番号：24650011），特別研究員奨励費（課題番号：25・1382），ならびに文部科学省科学研究費補助金若手研究（A）（課題番号：24680002）の助成ならびに支援を受けて行われた．

参考文献

[1] Zhang, G., Peng, X. and Zhao, Z.X.W.: Cloning Practices: Why Developers Clone and What can be Changed, *Proc. 28th International Conference on Software Maintenance*, pp.285–294 (2012).

[2] Yamashita, A. and Moonen, L.: Do Developers Care about Code Smells? An Exploratory Survey, *Proc. 20th Working Conference on Reverse Engineering*, pp.242–251 (2013).

[3] Roy, C.K., Zibran, M.F. and Koschke, R.: The Vision of Software Clone Management: Past, Present and Future, *Proc. Conference on Software Maintenance, Reengineering, and Reverse Engineering* (*CSMR-WCRE*), pp.18–33 (2014).

[4] Koschke, R.: Frontiers on Software Clone Management, *Proc. Frontiers of Software Maintenance in the 24th International Conference on Software Maintenance*, pp.119–128 (2008).

[5] 堀田圭佑，肥後芳樹，楠本真二：生成抑止，分析効率化，不具合検出を中心としたコードクローン管理支援技術に関する研究動向，コンピュータソフトウェア，Vol.31, No.1, pp.14–29 (2014).

[6] 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.91-D, No.6, pp.1465–1481 (2008).

[7] 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).

[8] Koschke, R.: Large-Scale Inter-System Clone Detection Using Suffix Trees and Hashing, *Journal of Software: Evolution and Process* (2013).

[9] Ishihara, T., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S.: Inter-Project Functional Clone Detection toward Building Libraries - An Empirical Study on 13,000 Projects, *Proc. 19th Working Conference on Reverse Engineering*, pp.387–391 (2012).

[10] Juergens, E., Deissenboeck, F., Hummel, B. and Wagner, S.: Do Code Clones Matter?, *Proc. 31st International Conference on Software Engineering*, pp.485–495 (2009).

[11] Göde, N. and Koschke, R.: Frequency and Risks of Changes to Clones, *Proc. 33rd International Conference on Software Engineering*, pp.311–320 (2011).

[12] Ishihara, T., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S.: Reusing Reused Code, *Proc. 20th Working Conference on Reverse Engineering*, pp.457–461 (2013).

[13] Kim, M., Sazawal, V., Notokin, D. and Murphy, G.C.: An Empirical Study of Code Clone Genealogies, *Proc. 13th International Symposium on Foundations of Software Engineering*, pp.187–196 (2005).

[14] Duala-Ekoko, E. and Robillard, M.P.: Clone Region Descriptors: Representing and Tracking Duplication in Source Code, *ACM Trans. Software Engineering and Methodology*, Vol.20, No.1, pp.3:1–3:31 (2010).

[15] Higo, Y., Hotta, K. and Kusumoto, S.: Enhancement of CRD-based Clone Tracking, *Proc. 13th International Workshop on Principles of Software Evolution*, pp.28–37 (2013).

[16] Bellon, S., Koschke, R., Antniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Software Engineering*, Vol.31, No.10, pp.804–818 (2007).

[17] Roy, C.K. and Cordy, J.R.: A Mutation/Injection-based Automatic Framework for Evaluating Clone De-

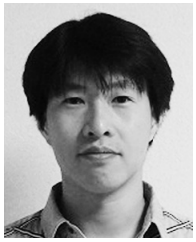
tection Tools, *Proc. 4th Workshop on Mutation Analysis*, pp.157–166 (2009).
[18] Göde, N. and Koschke, R.: Incremental Clone Detection, *Proc. 13th European Conference on Software Maintenance and Reengineering*, pp.219–228 (2009).
[19] 肥後芳樹，植田泰士，西野 稔，楠本真二：プログラム依存グラフを用いた増分的なコードクローン検出，情報処理学会論文誌，Vol.53, No.2, pp.601–611 (2012).
[20] Baker, B.: On Finding Duplication and Near-Duplication in Large Software Systems, *Proc. 2nd Working Conference on Reverser Engineering*, pp.86–95 (1995).
[21] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. 14th International Conference on Software Maintenance*, pp.368–377 (1998).
[22] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Software Engineering*, Vol.28, No.7, pp.654–670 (2002).
[23] Ducasse, S., Rieger, M. and Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code, *Proc. 15th International Conference on Software Maintenance*, pp.109–118 (1999).
[24] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD : Scalable and Accurate Tree-based Detection of Code Clones, *Proc. 29th International Conference on Software Engineering* (2007).
[25] Murakami, H., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S.: Gapped Code Clone Detection with Lightweight Source Code Analysis, *Proc. 21st International Conference on Program Comprehension*, pp.93–102 (2013).
[26] Hummel, B., Juergens, 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).
[27] 石原知也，堀田圭佑，肥後芳樹，井垣 宏，楠本真二：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，情報処理学会論文誌，Vol.54, No.2, pp.835–844 (2013).
[28] Keivanloo, I., Rilling, J. and Charland, P.: Internet-scale Real-time Code Clone Search via Milti-level Indexing, *Proc. 18th Working Conference on Reverse Engineering*, pp.23–27 (2011).

堀田 圭佑 （正会員）

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．平成 25 年同大学院研究科博士後期課程修了．現在，日本学術振興会特別研究員 PD．博士（情報科学）．コードクローン分析に関する研究に従事．電子情報通信学会，IEEE 各会員．

楊 嘉晨

平成 23 年中国上海交通大学軟件学院軟件工程専業卒業．平成 25 年大阪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．



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．電子情報通信学会，IEEE 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．電子情報通信学会，IEEE，JFPUG 各会員．