粗粒度なコードクローン検出手法の精度評価

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

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のみならず，多数のプロジェクトから成る巨大なデータセットや，過去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高い精度での検出を実現するために，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いる．従って，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．その解決方法の一つとして，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と細粒度なコードクローン検出手法の精度を比較し，粗粒度なコードクローン検出手法の有用性を検討する．

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

Evaluating the Accuracy of Coarse-grained Clone Detection

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

Abstract: There exists many state-of-the-art clone detectors because of much effort of researchers. However, it has been still challenging to detect clones on large corpora of code even with such successful detectors. 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 study evaluates the accuary of it compared to fine-grained detectors and discusses the usefulness of it.

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]．このような理由から，コードクローンに関する研究が盛んに行われている [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の一つがコードクローンの検出技術である．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．従って，コードク

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
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

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検出ツールが提案，開発されている [6], [7].

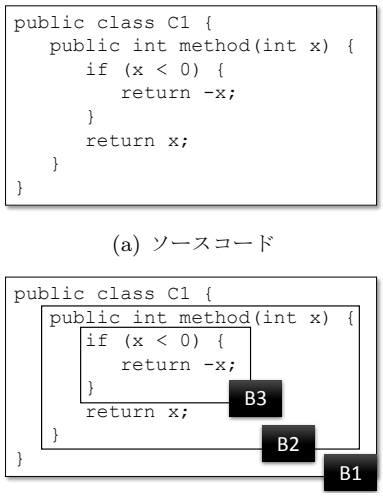
これまでに提案されているコードクローン検出技術は，ソースコードに対して細かな解析処理を行うことで，高い精度でコードクローンを検出することを目指している．しかしながら，これらのコードクローン検出技術を用いた場合，数千，数万のプロジェクトからなるプロジェクト集合などの大規模なソースコード集合に対してコードクローンの検出を行うことは困難である．その理由として，以下の点が挙げられる．

検出に要するコスト： 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．従って，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量： 中規模以上のソースコードに対してコードクローン検出ツールを適用すると，膨大な量のコードクローンが報告される．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，さらに膨大な量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極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挙げられる．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．粗粒度なコードクローン検出手法は，ソースコードをトークンや木構造として表現し比較する細粒度な検出手法と比較して，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量が小さくなる．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，有用な検出手法であるとは言い難い．一方，粗粒度な検出手法が細粒度な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，粗粒度な検出手法は大規模なソースコード集合に対する検出手法として有用であると考えられる．



(a) ソースコード

(b) ブロック

図 1 ブロックの例

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を比較した．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量が小さいことを確認した．さらに，粗粒度な検出手法は高い適合率を有している一方，再現率は細粒度な検出手法と比較して高いとはいえないことが明らかになった．しかし，適合率と再現率を総合して考えると，粗粒度な検出手法の精度は細粒度なものと比較して著しく劣るものでは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分析の初めの一歩として有用であるという結論を得た．

2. 粗粒度な検出手法

本節では，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コードクローン検出ツールについても言及する．

2.1 定義

本研究では，粗粒度なコードクローン検出手法を，“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，及び `if` 文などのブロック文が該当する．例えば図 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)，並びに `if` 文 (ブロック B3) の 3 つのブロックが存在する．

粗粒度なコードクローン検出手法では，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同士をコードクローンとして検出する．

2.2 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，木構造の頂点数と比較して小さくなる．従って，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．

2.3 実装

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，それを用いて精度並びに検出時間の評価を行った．ここでは，実装した粗粒度なコードクローン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コストを大幅に低減する手法である．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通りである．

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．

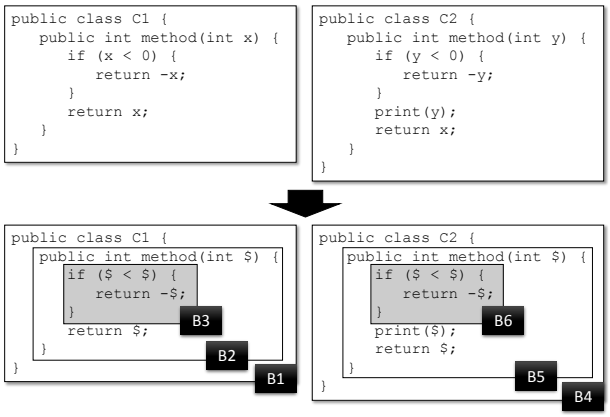


図 2 粗粒度なコードクローン検出ツールの適用例

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同士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．

なお，増分的なコードクローン検出を行う際の手順は以下の通り．

(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 2 に実装したコードクローン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．

3. 実験の設定

本節では，本研究で行う実験におけるコードクローン検出ツールの精度評価の方法を述べた後，実験の調査項目，実験環境，対象ソフトウェア，並びに比較対象となるコードクローン検出ツールについて述べる．

3.1 検出精度の評価方法

本研究では，コードクローン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 [8] 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コードクローン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．従って，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： コードクローン検出ツールによって検出されたクローンペア

それぞれのコードクローン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，そのコードクローン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みなされる．

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．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 ($precision(P,T)$) 並びに再現率 ($recall(P,T)$) 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，$Ref(P)$ 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，$precition(P,T)$，並びに $recall(P,T)$ を以下に定義する．

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

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

3.2 調査項目

本研究では，以下の 4 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

てより少ない量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と比較して高い適合率を有するか？

RQ4： 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と比較して高い再現率を有するか？

3.3 実験環境

本実験はコア数 16 の 64 ビット CPU を 2 つ備え，128GB の RAM を搭載したワークステーション上で行った．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．

3.4 対象ソフトウェア

本研究では，RQ1 に対する実験と RQ2，RQ3，RQ4 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，並びに 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，RQ4 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．本研究で用いる粗粒度なコードクローン検出ツールが 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．実験対象となるソフトウェアの概要を表 3 に示す．

表 1 UCI Dataset の概要

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

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)

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

3.5 比較対象として用いる検出ツール

表に本研究で用いる細粒度なコードクローン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．

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 らの手法は高速なコードクローン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．従って，本研究では粗粒度な検出手法との速度比較を主な目的として，この検出手法を比較対象に加えた．

4. 実験結果

本節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，調査課題に回答する．

なお本節では，本研究で実装した粗粒度なコードクローン検出ツールを Block-based と表記する．また，Dup，CloneDR，CCFinder，並びに Duploc に対する実験結果については，Bellon らの実験 [8] と同じ結果となるため，Bellon らの報告で記載されている数値を表記している．

4.1 RQ1

UCI Dataset 並びに 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．

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わかる．

従って，RQ1 には **Yes** と回答する．

表 4 比較対象となる検出ツール

Tool	Taxonomy
Dup [9]	Token
CloneDR [10]	AST
CCFinder [11]	Token
Duploc [12]	Text
Deckard [13]	AST
CDSW [14]	Other(Statement)
LD (based on [15])	Text

表 5 検出速度の比較

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

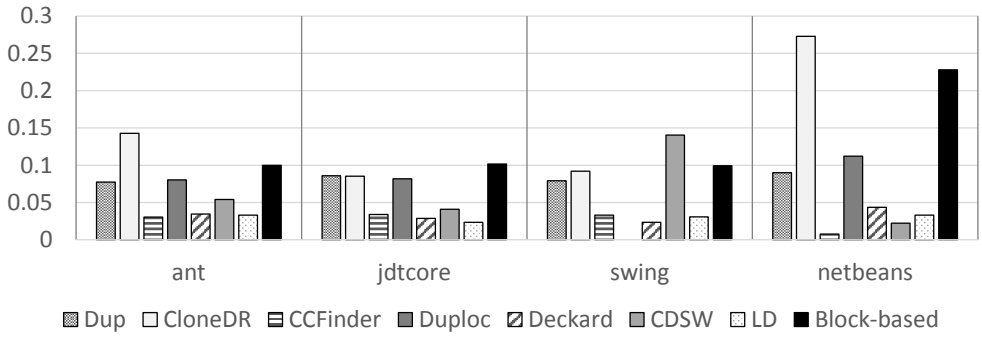


図 3 適合率

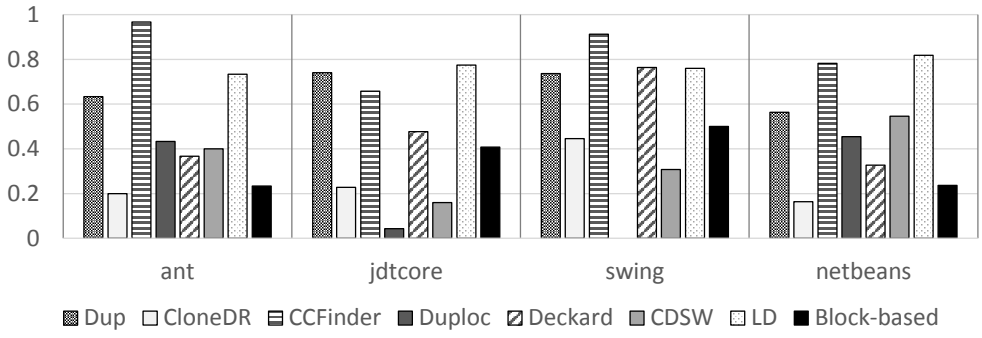


図 4 再現率

4.2 RQ2

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わかる．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．

4.3 RQ3

各検出ツールの検出結果の適合率を図 3 に示す．図中の黒の棒グラフが粗粒度な検出ツールの値を示している．

表 6 検出された候補クローンの数

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 3 より，粗粒度な検出手法は jdtcore において他のすべての検出ツールよりも高い適合率を有しており，その他の実験対象については 2 番目に高い値を有している．従って，RQ3 に対する回答は **Yes** となる．

4.4 RQ4

各検出ツールの検出結果の再現率を図 4 に示す．図 3 と同じく，図中の黒の棒グラフが粗粒度な検出ツールの値を表している．

図 4 より，粗粒度な検出手法の再現率は最もよい場合 (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．従って，RQ4 に対する回答は **No** となる．

5. 考察

本節では，実験結果に対する考察を述べる．

5.1 F 値の比較

実験結果より，粗粒度な検出手法は細粒度な検出手法と比較して高い適合率を有するが，再現率は高くないことが明らかになった．しかし，一般的にコードクローン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 [8]．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 における，検出ツール T の F 値 ($F(P,T)$) は以下のように定義さ

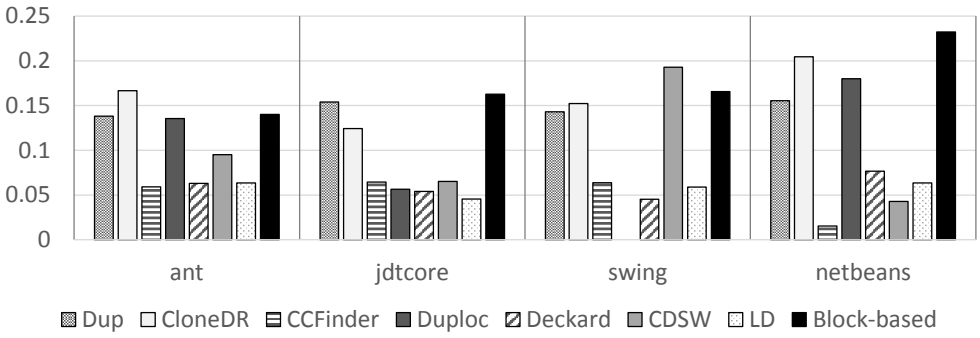


図 5 F 値

れる．

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

各検出ツールの F 値を図 5 に示す．図 5 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わかる．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．

5.2 2 段階のコードクローン分析

実験結果より，粗粒度な検出手法は高い適合率と F 値を有しており，高い検出精度を有する検出手法であることが明らかになった．さらに，粗粒度な検出手法が細粒度なものと比較して，大規模ソースコード集合に対しても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明らかになった．一方，粗粒度な検出手法の再現率は必ずしも高いとはいえないことも確認された．この結果は，細粒度な検出手法が検出できるコードクローンの一部を粗粒度な検出手法が特定できていないことを示している．

これらの結果から，粗粒度な検出手法と細粒度な検出手法のそれぞれの利点を活かし，欠点を補うために，双方の検出手法を用いた 2 段階のコードクローン分析を提案する．この方法では，大規模なデータセットに対し，まず粗粒度な検出手法を適用する．粗粒度な検出手法は細粒度な検出手法では扱うことの難しい大規模なデータセットを扱うことができ，検出されるコードクローンの数も細粒度なものと比較して小さくなる．かつ粗粒度な検出手法は高い適合率，F 値を有している．ゆえに，大規模なデータセットに存在するコードクローンのおおまかな情報をつかむために粗粒度な検出手法は有効であるといえる．

その一方で，粗粒度な検出手法は細粒度な検出手法が見つけることのできるコードクローンの一部を発見できない可能性がある．従って，2 段階のコードクローン分析では，粗粒度な検出手法を用いて得られた結果をもとに分析対象の絞り込みを行い，その後に細粒度な検出手法を適用する．これにより，粗粒度な検出手法では見つけることのできないコードクローンを考慮に入れたコードクローン分析が可能になる．

5.3 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，コードクローン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．従って，Bellon らのベンチマークでは，誤検出とみ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．従って，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．従って，他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コードクローン検出ツールは，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．従って，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 [16]．ゆえに，本実験におけるハッシュ値の衝突

の確率は十分に小さいものと考えている．

6. 関連研究

Bellon らは 8 つのソフトウェアを対象とし，6 つのツールを用いたコードクローン検出手法の精度評価を行った [8]．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 と Cordy の実験がある [17]．Bellon らの実験ではクローンペアを目視で確認して正解か否かを判別しているが，Roy らの手法はクローンペア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，コードクローン検出手法の評価を行っている．

7. 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分析の第一歩として有用な手法であるという結論を得た．

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)，並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) の助成並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, *Proceedings of the 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, *Proceedings of the 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, *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 18–33 (2014).

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

[9] Baker, B.: On Finding Duplication and Near-Duplication in Large Software Systems, *Proceedings of the 2nd Working Conference on Reverser Engineering*, pp. 86–95 (1995).

[10] Baxter, I., Yahin, A., L. Moura, M. A. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proceedings of the 14th International Conference on Software Maintenance*, pp. 368–377 (1998).

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

[12] Ducasse, S., Rieger, M. and Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code, *Proceedings of the 15th International Conference on Software Maintenance*, pp. 109–118 (1999).

[13] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD : Scalable and Accurate Tree-based Detection of Code Clones, *Proceedings of the 29th International Conference on Software Engineering* (2007).

[14] Murakami, H., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S.: Gapped Code Clone Detection with Lightweight Source Code Analysis, *Proceedings of the 21st International Conference on Program Comprehension*, pp. 93–102 (2013).

[15] Hummel, B., Juergens, E., Heinemann, L. and Conradt, M.: Index-Based Code Clone Detection: Incremental, Distributed, Scalable, *Proceedings of the 26th International Conference on Software Maintenance*, pp. 1–9 (2010).

[16] Keivanloo, I., Rilling, J. and Charland, P.: Internet-scale Real-time Code Clone Search via Milti-level Indexing, *Proceedings of the 18th Working Conference on Reverse Engineering*, pp. 23–27 (2011).

[17] Roy, C. K. and Cordy, J. R.: A Mutation/Injection-based Automatic Framework for Evaluating Clone Detection Tools, *Proceedings of the 4th Workshop on Mutation Analysis*, pp. 157–166 (2009).