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.

多粒度コードクローン検出手法の提案

幸　佑亮†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1-5
E-mail: †{y-yusuke,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　一般的に，コードクローンはソフトウェアの保守性を低下させる原因とされており，これまでに多くのコードクローン検出手法が提案されている．近年，大規模なソースコードの集合に対してコードクローン検出ツールを適用し，ライブラリの候補や修正漏れを検出する研究が行われている．既存のコードクローン検出手法は，ファイル単位やコード片単位など単一の粒度でのみコードクローンを検出している．検出対象の粒度が大きいほど，検出時間が短いが，検出可能なコードクローンが少なくなる．一方，検出対象の粒度が小さいほど，検出可能なコードクローンは多くなるが，検出時間が長い．そこで本研究では，粗粒度から細粒度へ段階的にコードクローンを検出する手法を提案する．段階的にコードクローンを検出する過程において，ある粒度でコードクローンとして検出されたコードをそれよりも細粒度なコードクローンの検出対象から除外していくことで，細粒度な検出手法と比較してより高速に検出できることを示した．また，粗粒度な検出手法と比較してより多くのコードクローンを検出できることを示した．
キーワード　コードクローン，コードクローン検出ツール，多粒度

1. ま え が き

コードクローン（以降，クローンと表記する）とは，ソースコード中に存在する互いに同一，あるいは類似したコード片である．クローンの主な発生要因はコピーアンドペーストである [1]．一般的に，クローンはソフトウェアの保守性を低下させる原因になるとされている．例えば，あるコード片にバグが存在した場合，そのコード片のクローンに対しても同様のバグが存在する可能性があり，同様の変更を検討する必要がある．そのため，クローンがソフトウェア中にどの程度存在しているか，及びどこに存在しているかを理解することはソフトウェア保守の観点から重要であり，これまでに多くのクローン検出ツールが提案されている [1] [2]．

また近年，単一のソフトウェアのみでなく，複数のソフトウェアからなる大規模なソースコードの集合に対してクローン検出ツールを適用し，ライブラリの候補や修正漏れを検出する研究が行われている [3]．複数のソフトウェアに跨って存在する同一の処理をライブラリ化することは開発効率の向上の観点から有益であり，先行研究においてそのようなクローンの存在が確認されている．

既存のクローン検出手法は，ファイル単位やコード片単位など単一の粒度でのみクローンを検出している．既存のクローン検出手法は，大きく分けて以下の 2 つの検出手法に分類することができる．

粗粒度な検出手法：ソースコードからクラス，メソッド，ブロックなどを抽出し，それらを互いに比較することでクローンを検出する手法．

細粒度な検出手法：ソースコードを行単位で比較，もしくはトークン列や木構造などに変換して比較することでコード片単位のクローンを検出する手法．

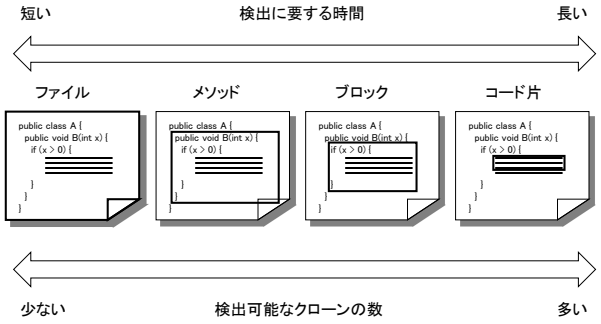


図 1　各粒度における検出手法の特徴

これらの手法は以下の一長一短な特徴を持つ．各粒度における検出手法の特徴を図 1 に挙げる．

検出に要する時間：検出対象の粒度が大きいほど，検出時間が短く，検出対象の粒度が小さいほど，検出時間が長い．ソースコードのクラス・メソッド・ブロックの数は行・トークン・木構造の頂点の数と比較して少ないため，粗粒度な検出手法は高速に検出可能である．

検出可能なクローンの数：検出対象の粒度が大きいほど，検出可能なクローンが少なく，検出対象の粒度が小さいほど，検出可能なクローンが多い．粗粒度な検出手法では，クラス・メソッド・ブロック内の一部のみが類似しているコード片をクロー

ンとして検出できないため，細粒度な検出手法が検出可能なクローンは多い．しかし，細粒度な検出手法の場合，検出されるクローンの数が多く，検出結果を分析することが難しい．

そこで本研究では，粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する手法を提案する．具体的にはファイル単位・メソッド単位・コード片単位の順にクローンを検出する．段階的にクローンを検出する過程において，ある粒度でクローンとして検出されたコードをそれよりも細粒度なクローンの検出対象から除外していくことで，細粒度な検出手法と比較してより高速に検出できる．また，粗粒度な検出手法と比較してより多くのクローンを検出した上で，複数のクローンペアを 1 つのクローンペアとしてまとめて検出可能になり，細粒度な検出手法と比較して検出されるクローンの数が少なくなる．これによって，より分析しやすいクローンの検出結果を生成することができる．

著者らは提案手法をクローン検出ツールとして実装し，多粒度な検出手法，粗粒度な検出手法，細粒度な検出手法を比較して評価を行った．

本研究の貢献は以下の通りである．

- 細粒度な検出手法と比較して，多粒度な検出手法が高速にクローンを検出できることを示した．
- 粗粒度な検出手法では検出できないクローンを多く検出し，かつ複数のクローンペアを 1 つのクローンペアとしてまとめて検出したケースがあることを示した．

本稿では，以下に示す定義を用いる．

クローンは類似性に基づいて，以下の 3 つのタイプに分類できる [4].

Type-1：空白やタブの有無，括弧の位置などのコーディングスタイルに依存する箇所を除いて，完全に一致するクローン．

Type-2：変数名や関数名などのユーザ定義名，また変数の型など一部の予約語のみが異なるクローン．

Type-3：Type-2 における変更に加えて，文の挿入や削除，変更が行われたクローン．

2. 提 案 手 法

粗粒度な検出手法は検出可能なクローンの数が少ない．細粒度な検出手法は検出に要する時間が長く，検出されるクローンの数が多い．そこで本研究では，それらの各デメリットを解決する多粒度な検出手法を提案する．

2.1 提案手法の概要

本研究の提案手法では，対象となる単一もしくは複数のソフトウェアのソースコードに対してファイル単位・メソッド単位・コード片単位の順に検出する．以降，ファイル単位のクローンをファイルクローン，メソッド単位のクローンをメソッドクローン，コード片単位のクローンをコード片クローンと呼ぶ．

図 2 に提案手法のイメージを挙げる．段階的にクローンを検出する過程において，ファイル単位の検出でクローンとして検出されたファイルを，次のメソッド単位の検出の入力から除外する．同様に，メソッド単位の検出でクローンとして検出されたメソッドを，次のコード片単位の検出の入力から除外する．

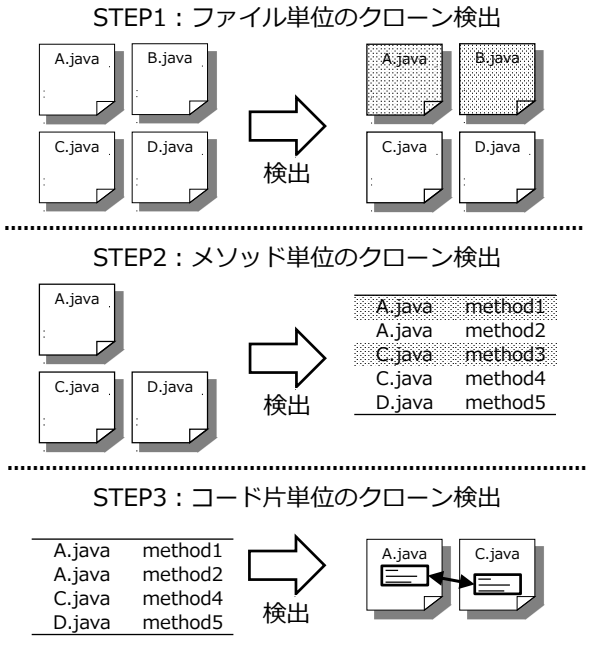


図 2　提案手法の概要

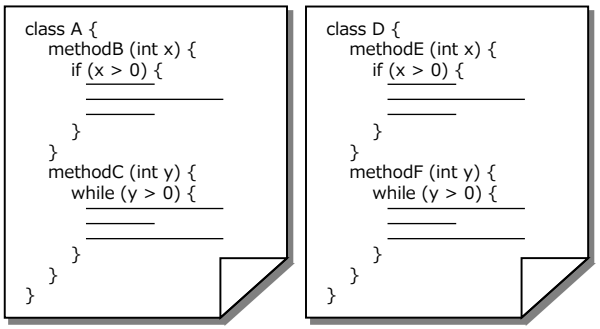
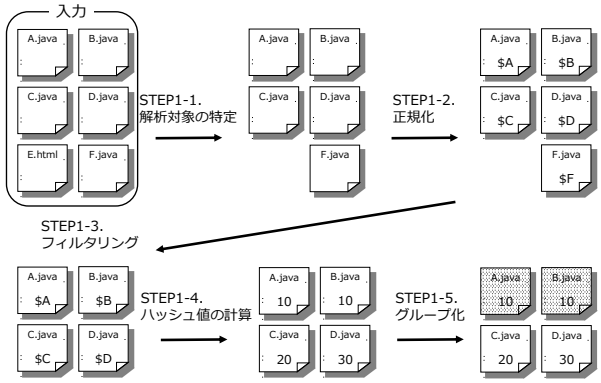


図 3　提案手法の副次的なメリット

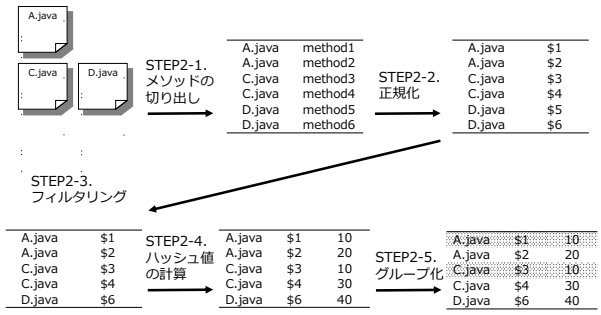
あるソフトウェア群に存在するメソッド集合のうち約 49%がメソッドクローンであったという研究報告がある [3]．ソフトウェアを跨ったメソッドクローンが多数存在するため，メソッド単位の検出の後，コード片単位の検出を行う場合，コード片単位の検出に要する時間的コストが大幅に改善されることが見込める．

2.2 提案手法の副次的なメリット

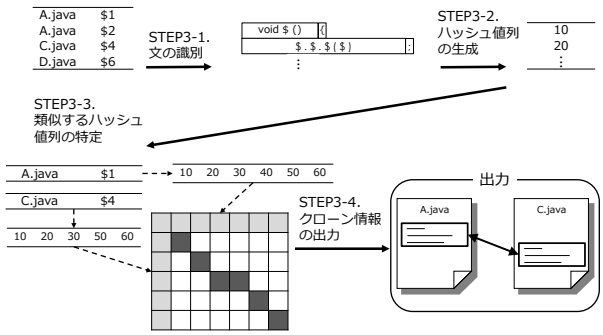
複数のクローンペアを 1 つのクローンペアとしてまとめて検出することで検出されるクローンの数が少なくなる．例を図 3 に挙げる．例えば，メソッド単位のみで検出した場合，メソッド B とメソッド E，メソッド C とメソッド F の 2 つのクローンペアが検出される．しかし，メソッド単位の検出を行う前にファイル単位の検出を行うことで，クラス A とクラス D の 1 つのクローンペアで検出することが可能である．つまり，このような連続したクローンをより粒度の大きい段階で検出することで，細粒度な検出手法と比較して検出されるクローンの数が少なくなる．



(a) ファイルクローン検出の詳細



(b) メソッドクローン検出の詳細



(c) コード片クローン検出の詳細

図 4　各粒度におけるクローン検出の詳細

3. 実　　装

2. 章で述べた提案手法を基に多粒度クローン検出ツール **Decrescendo** として実装した．**Decrescendo** の入力は以下の通りである．

- Java で記述された単一もしくは複数のソフトウェアのソースコード
- 最小クローン長（クローンとみなす最小のトークン数）
- 最大ギャップ率（検出されたトークン数に対する不一致なトークン数の割合）

出力はファイルクローンペア・メソッドクローンペア・コード片クローンペアの情報（ファイルパス・開始行・終了行・不一致行）である．**Decrescendo** が行う大まかな処理の流れは以下の通りである．

- ファイルクローンの検出

STEP1-1：解析対象の特定
STEP1-2：正規化
STEP1-3：ファイルタリング
STEP1-4：ハッシュ値の計算
STEP1-5：グループ化

- メソッドクローンの検出

STEP2-1：メソッドの切り出し
STEP2-2：正規化
STEP2-3：ファイルタリング
STEP2-4：ハッシュ値の計算
STEP2-5：グループ化

- コード片クローンの検出

STEP3-1：文の識別
STEP3-2：ハッシュ値列の生成
STEP3-3：類似するハッシュ値列の特定
STEP3-4：クローン情報の出力

Decrescendo は，設定によって各粒度における検出のオン・オフを切り替えることができる．メソッド単位の検出がオフの場合でも，コード片単位で検出する際の入力はメソッドの集合である．また，ファイル・メソッド単位の検出がオフの場合は，フィルタリング（STEP1-3，STEP2-3）を行っていない．

以降，各 STEP の詳細について述べる．

3.1　ファイルクローンの検出

ファイルクローンの検出手順は以下の通りである．概要を図 4(a) に示す．

STEP1-1：解析対象の特定

入力として与えられた単一もしくは複数のソフトウェアから，解析対象となるファイルを特定する．**Decrescendo** では，Java で記述されたソースコードのみを解析している．

STEP1-2：正規化

STEP1-1 で特定したファイルに対して，以下に示す正規化処理を行う．

- 空白・改行・コメント文・インポート文・修飾子を削除
- 識別子名・リテラルを特殊文字に置換

この処理によって，2 つのファイル間でコーディングスタイルが異なる場合や識別子名・リテラルが異なる場合でも，その 2 つのファイルをクローンとして検出可能になる．

STEP1-3：ファイルタリング

正規化後のファイルに含まれるトークン数が最小クローン長以下の場合，検出対象から除外する．処理が単純で短い構文のファイルは，大量に検出される可能性があり，一般的にそのようなファイルを検出する必要性は低いためである．

STEP1-4：ハッシュ値の計算

検出対象の各ファイルに対して，ハッシュ値を算出する．ハッシュ値が等しいソースコードはファイルクローンとなる．**Decrescendo** では，ハッシュ関数として MD5 を用いた．

STEP1-5：グループ化

ハッシュ値が同じファイルでグループを作る．それらのグループのうち，2 つ以上のファイルを持つグループがファイルクローンとして検出される．図 4(a) では，A.java と B.java がファイルクローンとなる．ここで検出されるクローンのタイプは Type-1，Type-2 である．

3.2 メソッドクローンの検出

メソッドクローンの検出手順は以下の通りである．概要を図 4(b) に示す．

STEP2-1：メソッドの切り出し

入力はファイルクローン検出の STEP1-5 にて同じハッシュ値を持たなかったファイルである．また，同じハッシュ値を持つファイルが 2 つ以上のグループから 1 つのファイルを無作為に選択し，そのグループを代表したファイルとして入力に加える．図 4(a) では，A.java と B.java がファイルクローンである．図 4(b) において，A.java を代表したファイルとして入力に加えている．これは，メソッドクローンの検出漏れを失くすための処理である．そして，入力されたファイルに対してメソッド毎に分割する．

STEP2-2：正規化

STEP2-1 で特定したメソッドに対して，STEP1-2 と同様の正規化処理を行う．

STEP2-3：ファイルタリング

正規化後のメソッドに含まれるトークン数が最小クローン長以下の場合，検出対象から除外する．

STEP2-4：ハッシュ値の計算

検出対象の各メソッドに対して，ハッシュ値を算出する．ハッシュ値が等しいメソッドはメソッドクローンとなる．**Decrescendo** では，ハッシュ関数として MD5 を用いた．

STEP2-5：グループ化

ハッシュ値が同じメソッドでグループを作る．それらのグループのうち，2 つ以上のメソッドを持つグループがメソッドクローンとして検出される．図 4(b) では，A.java のメソッド 1 と C.java のメソッド 3 がメソッドクローンとなる．ここで検出されるクローンのタイプは Type-1，Type-2 である．

3.3 コード片クローンの検出

コード片クローンの検出手順は以下の通りである．概要を図 4(c) に示す．

STEP3-1：文の識別

入力はメソッドクローン検出の STEP2-5 にて同じハッシュ値を持たなかったメソッドである．また，同じハッシュ値を持つメソッドが 2 つ以上のグループから 1 つのメソッドを無作為に選択し，そのグループを代表したメソッドとして入力に加える．図 4(b) では，A.java のメソッド 1 と C.java のメソッド 3 がファイルクローンである．図 4(c) において，そのうち A.java のメソッド 1 を代表したメソッドとして入力に加えている．そして，正規化後の各メソッドに対して文を識別する．文はセミコロンや中括弧で区切られた字句列とする．また，各文に含まれるトークン数を保存しておく．

STEP3-2：ハッシュ値列の生成

STEP3-1 で識別した各文に対してハッシュ値を算出する．**Decrescendo** では，ハッシュ関数として MD5 を用いた．

STEP3-3：類似するハッシュ値列の特定

生物学の分野で使われている Smith-Waterman アルゴリズム [5] を用いて類似するハッシュ値列を特定する．Smith-Waterman アルゴリズムとは，2 つの配列から類似する部分配列のペアを検出するためのアルゴリズムであり，部分配列の中にいくつかの不一致部分が存在していても検出できる．Smith-Waterman アルゴリズムを採用した理由は，Smith-Waterman アルゴリズムが応用されているクローン検出ツールがその他の Type-3 クローン検出ツールと比較して，高速に検出可能なためである [6]．全てのメソッドの組み合わせに対して，類似するハッシュ値列を特定する．

STEP3-4：クローン情報の出力

STEP3-3 で特定した類似する文全体に含まれるトークン数（$match$）が最小クローン長 θ 以上の場合（式 1），かつ不一致文に含まれるトークン数（gap）の割合が最大ギャップ率 ϕ 以下の場合（式 2），クローンペアとしてファイルパス・開始行・終了行・不一致行を出力する．

- $match \geqq \theta$ (1)
- $gap/match \leqq \phi$ (2)

4. 実　　験

4.1 準　　備

Decrescendo を用いて，多粒度な検出手法，粗粒度な検出手法，細粒度な検出手法を比較して，評価を行った．今回の実験では，最小クローン長を 50 トークン，最大ギャップ率を 0.3 としている．これは先行研究 [4] [7] を参考にしている．

4.2 調 査 項 目

調査項目 1：細粒度な検出手法と比較して，多粒度な検出手法が検出に要する時間が短いか．

調査項目 2：粗粒度な検出手法では検出できないクローンを多く検出し，かつ細粒度な検出手法と比較して検出されるクローンの数が少ないか．

これらの項目を調査するために，各粒度における検出のオン・オフを切り替えた全ての場合において，**Decrescendo** を実行した．

4.3 実 験 環 境

本実験で用いた計算機の CPU は 2.40GHz Intel Xeon CPU（2 プロセッサ）であり，メモリサイズは 32.0GB である．また，実験対象のソフトウェアや検出結果を出力するためのデータベースはすべて SSD 上に配置した．

表 1　対象ソフトウェア

ソフトウェア名	Java ファイル数	LOC
Forrest	252	32,491
OODT	1,628	223,846
Roller	612	96,617
計	2,492	352,954

表 2　実 験 結 果

	ファイルクローン	メソッドクローン	コード片クローン	計	実行時間
ファイル	70	-	-	70	1.8[s]
メソッド	-	1,006	-	1,006	4.7[s]
コード片	-	-	9,128	9,128	606.7[s]
ファイル・メソッド	70	935	-	1,005	5.0[s]
ファイル・コード片	70	-	9,057	9,127	580.9[s]
メソッド・コード片	-	1,006	8,122	9,128	55.8[s]
全粒度	70	935	8,122	9,127	55.2[s]

4.4 実 験 対 象

表 1 に対象ソフトウェアの概要を示す．対象ソフトウェアは Apache のリポジトリ (注1) から 2016/9/13 時点で最新のソフトウェアを 79 個取得し，そのうち無作為に 3 個のソフトウェアを選択した．バージョンが異なる同一ソフトウェア間からのクローン検出を避けるために trunk 以下のファイルのみを検出対象とする．

4.5 実 験 結 果

表 2 に実験結果を示す．ファイル・メソッド・コード片クローンの項目の数字は検出されたクローンペア数を指す．各調査項目ごとに結果を述べる．

4.5.1 調査項目 1

全粒度でクローンを検出した場合とコード片単位で検出した場合を比較すると，検出速度が大幅に向上していることが分かる．表 3 に各処理に要した時間を示す．

全粒度でクローンを検出した場合とコード片単位で検出した場合を比較すると，メソッド単位の検出における STEP2-1 から STEP2-4 に要する時間が若干ではあるが，短くなっていることが分かる．これはファイルクローンが除外され，入力されたファイル数が減少したためである．

さらに，最も注目すべきは，コード片単位の検出における STEP3-3 に要した時間である．この処理が最も実行時間が減少している．全粒度で検出した場合，実行時間が 1/12 倍以下になっている．どちらのパターンも実行時間の大半をコード片単位における STEP3-3 に要しているため，検出済みのファイルとメソッドを除外することが検出速度の向上に有効であることが分かる．また，ファイル・コード片単位で検出した場合と，メソッド・コード片単位で検出した場合を比較すると，検出済みのメソッドを除外することが特に有効であることが分かる．

```
public class NewXDoc extends Wizard implements INewWizard {
 (中略)
   public boolean performFinish() {
            (中略)
   }

   private void doFinish(String containerName, String fileName,
            IProgressMonitor monitor) throws CoreException {
            (中略)
   }

   protected void createFile(IResource resource, String fileName,
            IProgressMonitor monitor) throws CoreException {
            (中略)
   }
 (中略)
}
```

(a) NewXDoc.java

```
public class NewViewDoc extends Wizard implements INewWizard {
 (中略)
   public boolean performFinish() {
            (中略)
   }

   private void doFinish(String containerName, String fileName,
            IProgressMonitor monitor) throws CoreException {
            (中略)
   }

   protected void createFile(IResource resource, String fileName,
            IProgressMonitor monitor) throws CoreException {
            (中略)
   }
 (中略)
}
```

(b) NewViewDoc.java

図 5　複数のメソッドクローンが 1 つのファイルクローンとして検出された例

表 3　各処理に要した時間

処理	全粒度	コード片
STEP1-1 から STEP1-4	1.3[s]	1.3[s]
STEP1-5	0.0[s]	-
結果出力（ファイル）	0.9[s]	-
STEP2-1 から STEP2-4	2.5[s]	3.4[s]
STEP2-5	0.0[s]	-
結果出力（メソッド）	0.2[s]	-
STEP3-1 から STEP3-2	0.4[s]	0.5[s]
STEP3-3	48.2[s]	600.6[s]
結果出力（コード片）	1.0[s]	0.3[s]

結論として，細粒度な検出手法と比較して，多粒度な検出手法が高速にクローンを検出できることを示した．大規模なソースコードの集合に対してクローン検出を行う場合，多量のファイル・メソッドクローンが検出されることが予測されるため，特に有効であると想定される．

4.5.2 調査項目 2

全粒度でクローンを検出した場合とファイル・メソッド単位で検出した場合を比較すると，検出されたクローンペア数が 1,005 個から 9,127 個に増加した．全粒度でクローンを検出した場合とコード片単位で検出した場合を比較すると，検出され

(注1)：http://svn.apache.org/repos/asf/

たクローンペア数が 9,128 個から 9,127 個に減少した．

クローンペア数が減少した理由は，複数のメソッドクローンが 1 つのファイルクローンとして検出されたためである．実際の例を図 5 に示す．ファイル単位の検出がオフの場合は，メソッド単位の検出において performFinish・doFinish・createFile メソッドがそれぞれ異なるクローンペアとして検出されていた．しかし，ファイル単位で検出したことで，NewXDoc・NewViewDoc クラスの 1 つのクローンペアとして検出された．検出結果を確認したところ，ファイル単位で検出したことで，メソッド単位で検出された 71 個のメソッドクローンペアが 47 個のファイルクローンペアへ減少していた．

また，ファイル単位の検出ではファイルクローンとして検出されていたにも関わらず，メソッド単位の検出において，そのファイル内のメソッドがクローンとして検出されないケースが 23 個のファイルクローンに存在した．これはそのファイルクローン内の全メソッドが最小クローン長以下のメソッドであったためである．しかし，個々のメソッドはクローンとして検出する必要がないが，クラス全体はファイルクローンとして検出する必要があるケースが存在する．例えば，クラス全体をファイルクローンとして検出することで，複数のクラスを 1 つのクラスにまとめるリファクタリングができる可能性がある．このような場合においても，多粒度な検出手法は分析しやすいクローンの検出結果を生成することができる．

結論として，粗粒度な検出手法では検出できないクローンを多く検出した．また，複数のクローンペアを 1 つのクローンペアとしてまとめて検出したケースがあることを示した．大規模なソースコードの集合に対して多粒度で検出した場合，より多くのファイルクローンが検出されるため，クローンの数を少なくすることができる可能性がある．

4.5.3 議　　論

今回の実験では，検出に要する時間を改善することを示した．しかし，依然として，粗粒度な検出手法に比べると，細粒度な検出手法の実行速度は遅く，新たな工夫が必要である．コード片単位の検出結果を目視で確認したところ，識別子名が類似しているクラス・メソッド内の一部のコード片同士がクローンとして検出された場合が多く存在した．そこで，現時点で考えられるアイデアとして，Smith-Waterman アルゴリズムで類似するハッシュ値列を特定する際に，識別子名が類似していなければ，そのメソッドペアについては検出の対象としないことが考えられる [8]．これによって，検出されるクローンの数が多少減少するが，検出速度が大幅に向上する可能性がある．

5. 妥当性の脅威

対象ソフトウェア：本研究では，Java で記述された 3 つのソフトウェアを対象にしている．しかし，他のソフトウェアに対して実装したツールを実行した場合，本研究で得られた結果と異なる可能性がある．

ハッシュ値の衝突：本研究では，ファイル・メソッド・文を構成する文字列からハッシュ値を算出している．ハッシュ値の衝突が発生した場合，誤ったクローンが検出される可能性がある．しかし，本研究では 128 ビットのハッシュ値を出力する MD5 を用いており，ハッシュ値の衝突の可能性は十分に低いものと考えられる．

正規化の方法：本研究では，3. 章で述べた正規化を行っている．しかし，異なる正規化を行うことで，本研究で得られた結果と異なる可能性がある．

6. あ と が き

本研究では，粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する手法を提案した．著者らは提案手法をクローン検出ツールとして実装し，多粒度な検出手法，粗粒度な検出手法，細粒度な検出手法を比較して，評価を行った．

本研究の貢献は以下の通りである．

- 細粒度な検出手法と比較して，多粒度な検出手法が高速にクローンを検出できることを示した．
- 粗粒度な検出手法では検出できないクローンを多く検出し，かつ複数のクローンペアを 1 つのクローンペアとしてまとめて検出したケースがあることを示した．

今後の課題は以下の通りである．

- Java 以外の言語に対応．
- 識別子名の類似度を用いた細粒度な検出手法の速度向上．
- メモリ使用量の改善．
- その他のクローン検出ツールとの比較．
- 大規模なソースコードの集合に対してクローン検出ツールを適用し，ライブラリの候補や修正漏れを検出．

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：JP25220003）の助成を得て行われた．

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 D，vol.91，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.29–42，2011.

[3] 石原知也，堀田圭佑，肥後芳樹，井垣宏，楠本真二他，“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.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” Software Engineering, IEEE Transactions on, vol.33, no.9, pp.577–591, 2007.

[5] T.F. Smith and M.S. Waterman, “Identification of common molecular subsequences,” Journal of molecular biology, vol.147, no.1, pp.195–197, 1981.

[6] 村上寛明，堀田圭佑，肥後芳樹，井垣宏，楠本真二，“Smith-waterman アルゴリズムを利用したギャップを含むコードクローン検出，” 情報処理学会論文誌，vol.55，no.2，pp.981–993，2014.

[7] J. Svajlenko and C.K. Roy, “Evaluating clone detection tools with bigclonebench,” 2015 IEEE International Conference on Software Maintenance and Evolution, pp.131–140, 2015.

[8] Y. Higo and S. Kusumoto, “How should we measure functional sameness from program source code? an exploratory study on java methods,” In Proc. of the 22nd International Symposium on the Foundations of Software Engineering (FSE2014), pp.294–305, Nov. 2014.