多粒度コードクローンの検出と評価

幸 佑亮[1]　肥後 芳樹[1]　楠本 真二[1]

概要：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の 1 つとされており，コード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る．そのため，これまでに多くのコードクローン検出手法が提案され，自動的にコードクローンを検出するツールが開発されている．また近年，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コードクローンを検出する．一般的に，検出の粒度が粗いほど検出時間は短くなるが，検出可能なコードクローンは少なくなる．一方，検出対象の粒度が細かいほど検出可能なコードクローンは多くなるが，検出時間は長くなる．そこで本論文では，より多くのコードクローンをより短時間で検出することを目的として，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することで，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コードクローンを検出できる．提案手法をコードクローン検出ツール **Decrescendo** として実装し，複数のオープンソースソフトウェアに適用した．そして，提案手法を粗粒度な検出手法および細粒度な検出手法と比較して評価を行った．実験の結果より，細粒度な（コード片単位の）検出手法と比較して，多粒度な検出手法が約 10〜20 倍高速にコードクローンを検出できることを示した．また，粗粒度な（メソッド単位の）検出手法と比較して，多粒度な検出手法が約 10〜30 倍のコードクローンを検出した．この検出数は細粒度な（コード片単位の）検出手法とほぼ同数であった．

1. はじめに

コードクローン（以下，クローン）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 [8], [9], [13], [16]．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．

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 [14]．複数のソフトウェアに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できる [17]．

粗粒度な検出手法　類似したクラス・メソッド・ブロックをクローンとして検出する手法．

細粒度な検出手法　任意のコード片をクローンとして検出する手法．細粒度な検出手法はクラス・メソッド・ブロックの一部が類似するコード片をクローンとして検出できる．

これらの手法は以下の一長一短な特徴を持つ．ファイル単位，メソッド単位，ブロック単位，およびコード片単位の検出手法の特徴を図 1 に挙げる．

検出に要する時間　検出の粒度が粗いほど検出時間が短く，検出の粒度が細かいほど検出時間が長い．ソースコードの行数，ソースコード中の字句数，ソースコードを抽象構文木で表現した場合の頂点数はソースコード中のクラス数，メソッド数，ブロック数と比較して多い．

[1] 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
〒 565-0871 大阪府吹田市山田丘 1-5

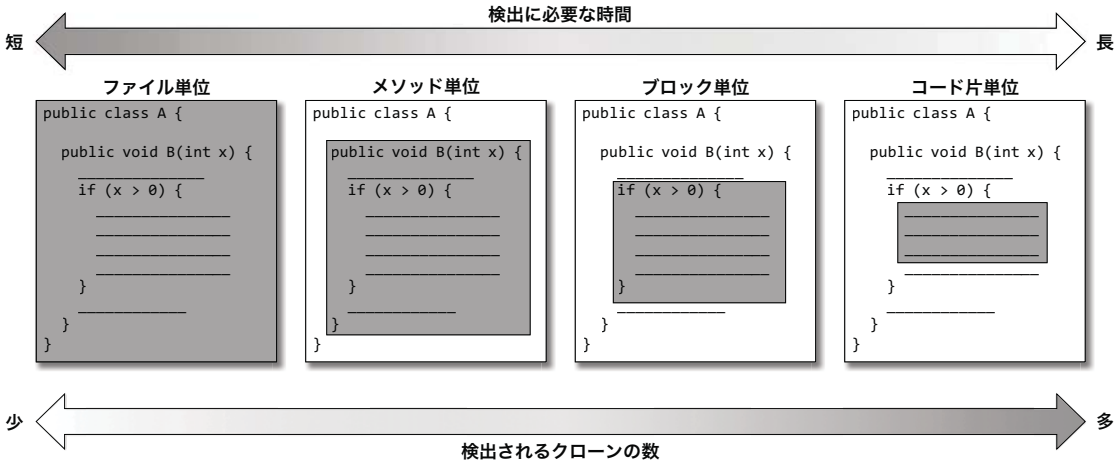


図 1　各検出手法の特徴

そのため，粗粒度な検出手法と比較して細粒度な検出手法は検出時間が長い．

検出可能なクローンの数　検出の粒度が粗いほど，検出可能なクローンの数が少なく，検出の粒度が細かいほど，検出可能なクローンの数が多い．粗粒度な検出手法はファイル単位，メソッド単位，もしくはブロック単位でクローンを検出するが，それらの内部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．そのため，細粒度な検出手法と比較して粗粒度な検出手法は検出可能なクローンの数が少ない．

本論文では，粗粒度な検出手法と細粒度な検出手法のデメリットを低減するために，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，粗粒度で検出できるクローンは粗粒度で検出することによって，より有益なクローンの検出結果となる．例えば，複数のメソッド単位のクローンを 1 つのファイル単位のクローンとして検出できる．これによって，検出されるクローンの数が少なくなり，クローンの分析に要する時間を短縮できる．特にクローンの情報をリファクタリングにおいて利用する場合に有効であると著者らは考えている．また，提案手法は細粒度でも検出を行うためクローンの見逃しが粗粒度の検出に比べて少ない．このように，提案手法は粗粒度な検出手法や細粒度な検出手法と比較してより分析しやすいクローンの検出結果を生成できる．

提案手法をクローン検出ツール **Decrescendo** として実装し，複数のオープンソースソフトウェアに適用した．そして，提案手法を粗粒度な検出手法および細粒度な検出手法と比較した．

図 2　ファイルのハッシュ値に基づいたクラスタリング

本論文の貢献は以下の通りである．

- 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．
- 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，および粗粒度な検出手法と比較して多粒度な検出手法がクローンの検出数が多いことを示した．
- 細粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できることを示した．

以降，2 章では本研究の位置づけについて述べる．3 章では提案手法について述べる．4 章では実装したツールの詳細について述べる．5 章では評価実験について述べる．6 章では実験の妥当性について述べる．最後に，7 章で本論文をまとめる．

2. 本研究の位置づけ

コード片単位のクローン検出を行う前にファイルのハッシュ値に基づいたクラスタリングを行う，という Choi らの先行研究がある [3]．このクラスタリングはコード片単

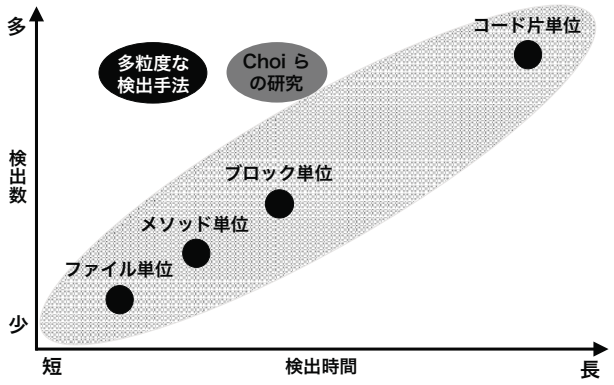


図 3　本研究の位置づけ

位のクローン検出時間を短縮することが目的である．ファイルのハッシュ値に基づいたクラスタリングの例を図 2 に示す．ファイル中の数値はハッシュ値を表しており，ファイル外の枠はハッシュ値が等しいファイルのグループを表す．クラスタリング後，各グループから 1 つのファイルを選択する．そして，選択されたファイルの集合に対してクローン検出を行う．このクラスタリングは，ファイル単位のクローン検出と同一である．

本論文の提案手法は，ファイル単位のクローンに加えて，メソッド単位のクローンも除外する．これによって，既存研究より効果的にコード片単位のクローン検出時間を短縮できる．さらに，本提案手法は検出した各クローンに粒度の情報を付加する．これにより，検出した各クローンの粒度が明らかになり，先行研究よりも分析しやすいクローンの検出結果を生成できる．

既存のクローン検出手法と本論文で提案する多粒度な検出手法の位置づけを図 3 に示す．粗粒度な検出手法（ファイル単位・メソッド単位・ブロック単位の検出手法）は，検出に要する時間が短く，検出されるクローンの数が少ない．一方，細粒度な検出手法（コード片単位の検出手法）は，検出時間が長く，検出数が多い．そこで，本論文で提案する多粒度な検出手法は，検出時間を短く，かつ検出数を多くすることを目的とする．また，本論文の提案手法は，ファイル単位のクローンに加えて，メソッド単位のクローンも検出する．13,000 個の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 [14]．複数のソフトウェア間に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にコード片単位の検出を行うことで，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．

3. 提案手法

提案手法は対象ソースコードに対して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．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．段階的に検出する過程において，ある粒度

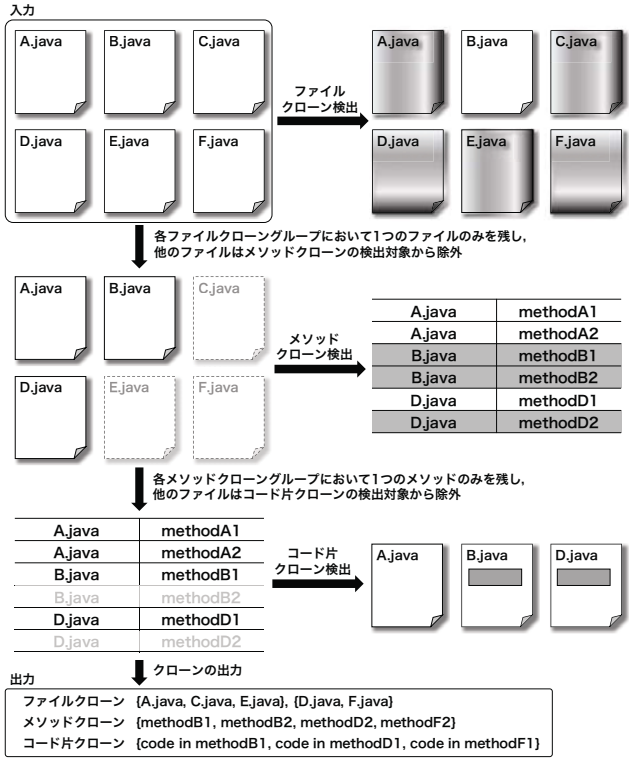


図 4　提案手法の概要

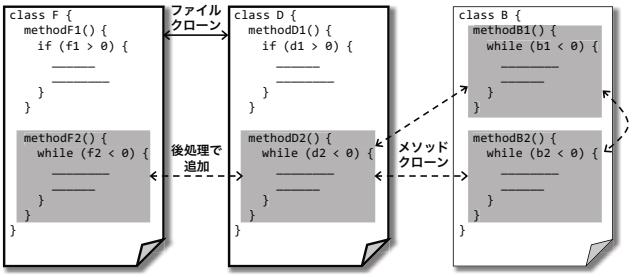


図 5　後処理の例

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．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．

現在のところ，提案手法ではブロック単位でのクローン検出を行わない．その理由は以下の 2 つである．

- ブロックの大きさはメソッドよりも小さく，ブロック単位の検出を導入することによる高速化が限定的と考えたため．
- コードクローンは 1 つのブロック内で閉じているとは限らないため，検出の処理が煩雑になりすぎてしまうと考えたため．

3.1 検出手順

図 4 に提案手法の概要を示す．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．図 4 では，ファイル単位の検出において，以下の 2 つのファイルクローングループが検出されている．

- A.java, C.java, F.java
- D.java, E.java

次のメソッド単位の検出では各ファイルクローングループから 1 つのファイルを選ぶ（各グループ内の他のファイルはメソッドクローンの検出対象から除外する）．

選択されたファイルとファイルクローンになっていないファイル内に存在しているメソッドに対してメソッドクローンの検出処理が適用される．この例では，メソッドクローン検出処理により以下の 3 つのメソッドがクローンとして検出されている．

- methodB1, methodB2, methodD2

このあと除外したファイルに対するメソッドクローンの検出漏れを防ぐための後処理が行われる．具体的には，メソッドクローン methodD2 が存在している D.java は F.java とファイルクローンであるため，F.java に含まれる methodF2 もこのメソッドクローングループに加える（図 5）．結果として，以下の 4 つのメソッドが 1 つのメソッドクローングループとして検出される．

- methodB1, methodB2, methodD2, methodF2

次のコード片単位の検出では，各メソッドクローングループから 1 つのメソッドを選び，メソッドクローンとなっていないメソッドと共にコード片クローン検出処理が適用される．コード片クローン検出後はメソッドクローン検出後の後処理と同様に，除外したメソッド内に含まれるコード片クローンの検出漏れを防ぐための後処理が行われる．

3.2 より理解しやすい検出結果の出力

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より理解しやすいクローンの検出結果を得ることができる．例を図 6 に挙げる．この例ではメソッド単位のみで検出した場合，3 つファイルからは以下の 2 つのメソッドクローングループが検出される．

- methodD1, methodF1
- methodB1, methodB2, methodD2, methodF2

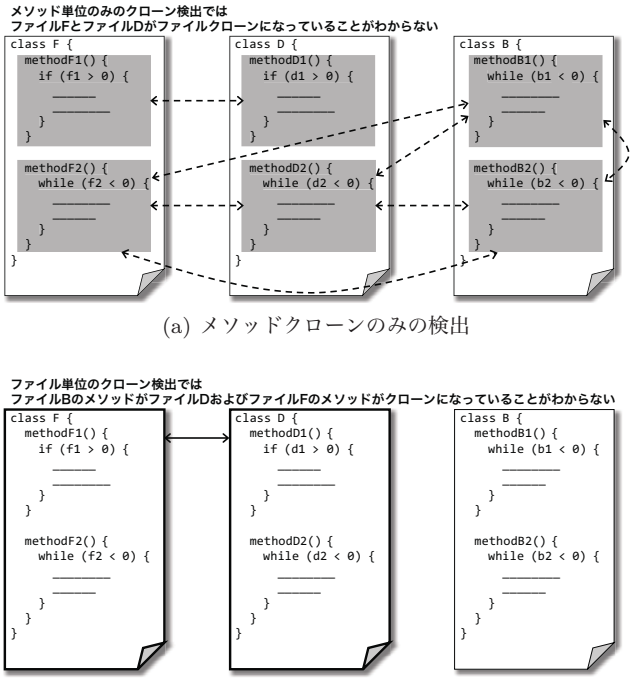
この検出結果からは，D.java と F.java がファイルクローンとなっていることがわかりづらい．

ファイル単位のみで検出した場合，以下のファイルクローングループが検出される．

- D.java, F.java

この結果からは，B.java が D.java および F.java とメソッドクローンを共有していることがわからない．

多粒度で検出することで以下のクローングループが検出される．



(a) メソッドクローンのみの検出

(b) ファイルクローンのみの検出

図 6　単粒度におけるクローン検出例

- D.java, F.java
- methodB1, methodB2, methodD2, methodF2

このように，多粒度で検出を行うことにより，小さいクローンを見逃すことなく，重複コードはできるだけ大きなクローンとして検出できる．

4. 実装

ここでは，著者らが実装した多粒度クローン検出ツール **Decrescendo** について述べる．なお，このツールにおいて，メソッドクローン検出部分は著者らの既存研究 [14] と同一であり，提案手法のファイルクローン検出部分は既存研究 [14] の検出法をファイル単位に変更したものである．コード片クローン検出については Suffix-Tree アルゴリズムおよび Smith-Waterman アルゴリズムを用いて実装した．どちらのアルゴリズムもクローン検出において有用なことが示されている [5], [15].

Decrescendo の入力は以下の通りである．

- Java ソースファイル群
- 最小クローン長
- 最大ギャップ率

最小クローン長は検出するクローンの最小の大きさを指定するためのものである．最小クローン長に小さな値を指定すればより多くのクローンを検出できるが，検出結果には多くの誤検出が混じる．大きな値を指定すれば誤検出を減らせるが見逃すクローンも増えてしまう．

最大ギャップ率とは，Smith-Waterman アルゴリズムを利用する場合にのみ指定する値であり，クローンとして許

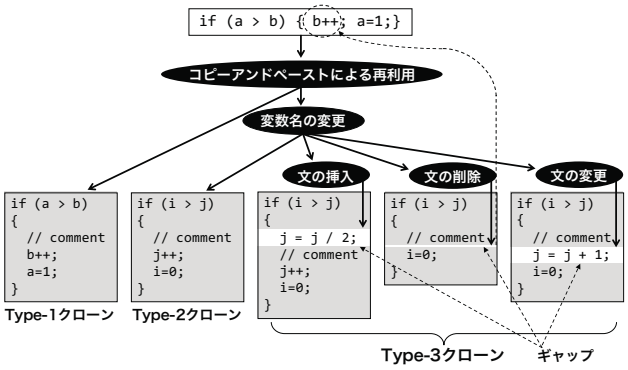


図 7　クローンのタイプ

容するギャップ（不一致部分）の割合を表す．最大ギャップ率を大きくすればより多くのクローンを検出できるが検出結果には多くの誤検出が含まれてしまう．小さい値を設定すれば誤検出を減らせるが見逃してしまうクローンが増えてしまう．

出力はクローンペアの位置情報，粒度（ファイル・メソッド・コード片），およびタイプ（Type-1・Type-2・Type-3）である．クローンのタイプを図 7 を用いて説明する．例えば，コピーアンドペーストしただけのコードであれば，コピー元とコピー先のコード片は完全に同一な Type-1 クローンとなる．変数名を変更した場合には字句単位の差異が生じ，Type-2 クローンとなる．文の追加・削除・変更を行った場合には字句単位よりも大きな差異（ギャップ）が生じ，Type-3 クローンとなる．

検出するクローンのタイプは以下の通りである．

- ファイル単位およびメソッド単位では，Type-1 および Type-2 のクローンが検出される．
- Suffix-Tree アルゴリズムを用いてコード片クローンの検出を行った場合は，Type-1 および Type-2 のクローンが検出される．
- Smith-Waterman アルゴリズムを用いてコード片クローンを検出した場合は，Type-1，Type-2，および Type-3 のクローンが検出される．

コード片単位の検出手法を 2 通りの手法で実装した理由は，細粒度な検出手法のうち，Type-2 までのクローンを検出する手法と Type-3 までのクローンを検出する手法のどちらに対しても，多粒度な検出手法が検出時間と検出数の面で有用かどうかを評価するためである．Type-2 までのクローンを検出する手法の中で Suffix-Tree アルゴリズムを採用した理由は，Suffix-Tree アルゴリズムが用いられているクローン検出ツールが代表的な字句単位の検出ツールであり，様々な企業や研究で採用されているためである [1], [6], [12]．Type-3 までのクローンを検出する手法の中で 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 [15]．

以降，本章では Decrescendo の処理の流れを説明する．必要に応じて図 4 を参照されたい． Decrescendo の最初の処理はファイルクローンの検出である．

STEP-1(ファイルの正規化)　入力として与えられた各 Java ファイルに対して以下に示す正規化を行う．

- インデント，改行，コメント，import 文，修飾子を削除
- 識別子名および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または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．

STEP-2(ファイルフィルタリング)　正規化後のファイルに含まれる字句数が最小クローン長に満たない場合，検出対象から除外する．このフィルタリングを行うことによりクローンが検出されることのないファイルに対するこれ以降の処理を省くことができる．

STEP-3(ファイルハッシュ値の計算)　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ファイルがファイルクローンとなる．現在の実装ではハッシュ値の計算に MD5 を利用している．

STEP-4(ファイルグループの生成と出力)　ハッシュ値が同じファイルでグループを作る．2 つ以上のファイルを含むグループがファイルクローンとして検出される．

ファイルクローンの検出の後は，メソッドクローンの検出を行う．

STEP-5(メソッドの切り出し)　STEP-4 において生成された各グループからファイルを 1 つ無作為に選択する．ファイルが 1 つだけ含まれるグループも対象である．選択された各ファイルに含まれるメソッドを抽出する．図 4 では，A.java，B.java，および D.java が選択されている．

STEP-6(メソッドフィルタリング)　メソッドに含まれる字句数が最小クローン長に満たない場合，検出対象から除外する．このフィルタリングを行うことにより，クローンが検出されることのないメソッドに対するこれ以降の処理を省くことができる．

STEP-7(メソッドハッシュ値の計算)　各メソッドに対してハッシュ値を計算する．ハッシュ値が等しいメソッドがメソッドクローンとなる．ファイルハッチュと同じく MD5 を用いて計算する．

STEP-8(メソッドグループの生成)　ハッシュ値が同じメソッドでグループを作る．2 つ以上のメソッドを含むグループがメソッドクローンとして検出される．図 4 では，methodB1，methodB2，および methodD2 がメソッドクローンとして検出されている．

STEP-9(メソッドクローン検出の後処理と出力)
STEP-4 で選択されなかったファイル（図 4 における C.java，E.java，および F.java）に含まれるメソッドクローンを取りこぼさないための処理を行う．1 つのグループを構成する各ファイルはその内部にメソッドクローンを持つ．この性質を利用して，STEP-8 で検出されたメソッドクローンが 2 つ以上のファイルからなるグループから選択されたメソッドである場合には，そのメソッドクローンが同じグループに含まれる他のファイルのメソッドに対しても存在しているとする．図 4 では，この処理により，STEP-8 で検出されたメソッドクローンに，methodF2 が加えられている．

メソッドクローンを検出した後は，コード片クローンの検出を行う．

STEP-10(コード片クローンの検出) STEP-8 において生成された各グループからメソッドを 1 つ無作為に選択する．メソッドが 1 つだけ含まれるグループも対象である．選択された各メソッドに対して，Suffix-Tree もしくは Smith-Waterman アルゴリズムを用いることにより，それらに含まれるコード片クローンを検出する[*1]．

STEP11(コード片クローン検出の後処理と出力)
STEP-4 で選択されなかったファイルや STEP-10 で選択されなかったメソッドに含まれるコード片クローンを取りこぼさないための処理を行う．処理内容は STEP-9 と同様である．

なお，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．

5. 実験

5.1 準備

Decrescendo を複数のオープンソースソフトウェアに対して適用し，多粒度な検出手法と粗粒度な検出手法および細粒度な検出手法と比較した．今回の実験では，最小クローン長を 50 字句，最大ギャップ率を 0.3 としている．これらの値は先行研究 [2][11] を参考にしている．

5.2 調査項目

本実験の調査項目を以下に示す．

表 1 対象ソフトウェア

ソフトウェア数	84
ファイル数	66,724
メソッド数	628,219
LOC	11,545,556

[*1] 本論文では Suffix-Tree アルゴリズムを用いたクローン検出および Smith-Waterman アルゴリズムを用いたクローン検出の手順は記述しない．それらは文献 [5], [15] のクローン検出の手順と同様である．

項目 1 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法は検出時間が短いか．

項目 2 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法は検出されるクローンの数が多いか．

項目 3 粗粒度な検出手法および細粒度な検出手法と比較して，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できるか．

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．コード片単位の検出を行う場合，Suffix-Tree アルゴリズムと Smith-Waterman アルゴリズムを用いた 2 通りにおいて，Decrescendo を実行した．

5.3 実験環境

本実験で用いた計算機の CPU は 2.9GHz Intel Xeon CPU（16 プロセッサ）であり，メモリサイズは 352GB 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．

5.4 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 Software Foundation のリポジトリ[*2]から取得した 2013/10/31 時点の 84 個のソフトウェアである．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．これらのソフトウェアを対象とした理由は，著者らの先行研究 [4] において利用されたデータセットであり，テストコードと自動生成コードが取り除かれているためである．テストコードと自動生成コードは流用の特定や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対しては有用でなく，クローンとして検出する必要性がない．

5.5 項目 1 の調査

表 2 にクローンの検出結果を示す．各粒度の ○ はその粒度の検出が行われたことを表す．検出 ID は，本節の以降でどの検出について言及しているのかを簡潔に表すための記号である．ST および SW はそれぞれ Suffix-Tree アルゴリズム，Smith-Waterman アルゴリズムを表す．なお，Suffix-Tree アルゴリズムを用いた場合と Smith-Waterman を用いた場合で同様の傾向が得られたので，ここでは Suffix-Tree アルゴリズムを用いた場合のみを述べる．

ファイル単位，メソッド単位で検出した場合（検出 A，検出 B）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出時間が長かった（7.3 秒，32.8 秒 →3,094.6 秒）．また，コード片単位で検出した場

[*2] http://svn.apache.org/viewvc/

表 2　クローンの検出結果

検出 ID	粒度				検出されたクローンペア数				検出時間 [s]
	ファイル	メソッド	コード片 ST	コード片 SW	ファイル	メソッド	コード片	合計	
A	○	-	-	-	11,029	-	-	11,029	7.3
B	-	○	-	-	-	49,446	-	49,446	32.8
C	-	-	○	-	-	-	1,637,597	1,637,597	62,569.0
D	-	-	-	○	-	-	446,442	446,442	130,367.1
E	○	○	-	-	11,029	44,168	-	55,197	32.3
F	○	-	○	-	11,029	-	1,632,319	1,644,348	46,183.2
G	-	○	○	-	-	49,446	1,588,151	1,637,597	1,885.7
H	○	○	○	-	11,029	44,168	1,588,151	1,643,348	3,094.6
I	○	-	-	○	11,029	-	441,164	452,193	109,717.1
J	-	○	-	○	-	49,446	396,996	446,442	13,013.6
K	○	○	-	○	11,029	44,168	396,996	452,193	13,036.7

合（検出 C）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出速度が大幅に向上した（62,569.0 秒 →3,094.6 秒）．表 3 に各処理に要した時間を示す．全粒度で検出した場合は，STEP-5 から STEP-7（メソッドクローンの検出）に要する時間が若干短くなった（31.7 秒 →24.0 秒）．これはファイルクローンとして検出されたファイルが除外され，メソッド単位の検出時の入力ファイル数が減少したためである．

最も注目すべきは，STEP-10（コード片クローンの検出）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（62,508.9 秒 →2,980.0 秒）．全粒度で検出した場合に実行時間が約 1/20 倍になっている．どちらの場合も検出時間の大半を STEP-10 に要しているため，重複ファイルと重複メソッドを除外することが Suffix-Tree アルゴリズムを用いたマッチング処理に要する時間を短縮し，結果として検出速度の向上に有効であることが分かる．また，STEP-11 に要した時間を時間を比較すると，全粒度で検出した場合は後処理を行うため実行時間が長くなっている．しかし，その差は全体の検出時間と比較すると僅かな時間である．

さらに，ファイルおよびコード片単位で検出した場合（検出 F）と全粒度で検出した場合（検出 H）を比較すると，メソッドクローンを除外してコード片クローンの検出を行うことが有効であることが分かる（46,183.2 秒 →3,094.6 秒）．

項目 1 の結論

これらの結果から多粒度な検出手法は粗粒度な検出手法よりも検出時間が長いという結論を得た．しかし，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 [7], [10], [14]．そのような場合において，多粒度な検出手法はコード片クローンを検出するための時間を短縮することに有効であるといえる．

5.6　項目 2 の調査

項目 2 についても，Suffix-Tree アルゴリズムを用いた場合と Smith-Waterman アルゴリズムを用いた場合で同様の傾向であったため，Suffix-Tree アルゴリズムを用いた場合のみについて述べる．

ファイル単位およびメソッド単位で検出した場合（検出 A，検出 B）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出数が多かった（11,029 個，49,446 個 →1,643,348 個）．また，コード片単位で検出した場合（検出 C）と全粒度で検出した場合（検出 H）を比較すると，検出数にあまり変化はなかった（1,637,597 個 →1,643,348 個）．

項目 2 の結論

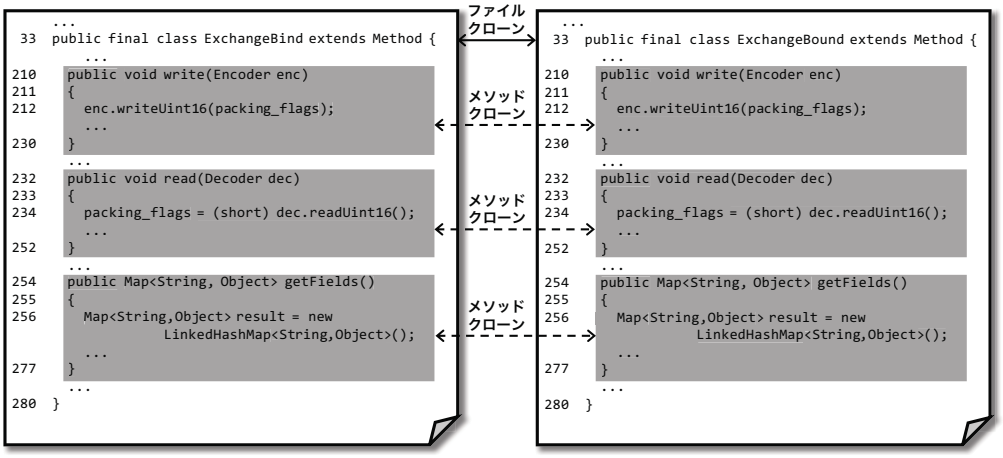
粗粒度な検出手法と比較して多粒度な検出手法は検出数が多いということを示した．また，多粒度な検出手法は細粒度な検出手法と同等数のクローンを検出することを示した．

表 3　各処理に要した時間（Suffix-Tree アルゴリズム）

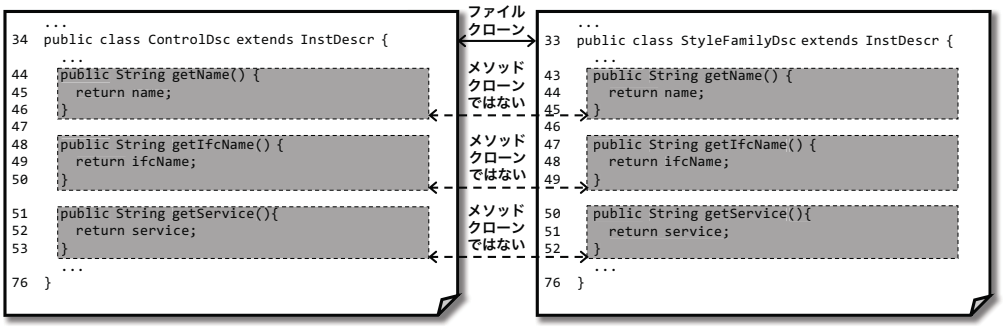
処理	全粒度 [s]	コード片 [s]
STEP-1 から STEP-3	6.7	6.4
STEP-4	4.0	-
STEP-5 から STEP-7	24.0	31.7
STEP-8 および STEP-9	1.2	-
STEP-10	2,980.0	62,508.9
STEP-11	68.5	16.6

表 4　各処理に要した時間（Smith-Waterman アルゴリズム）

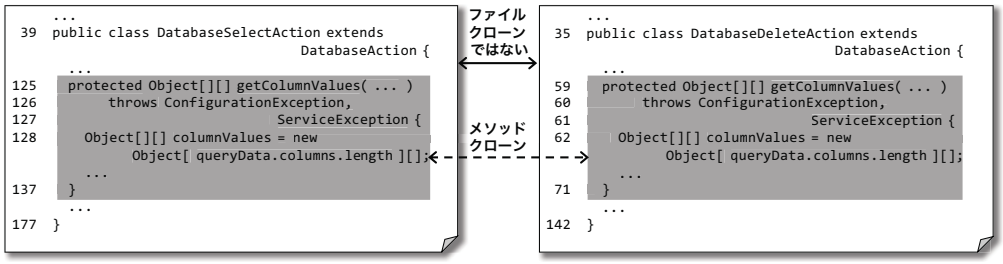
処理	全粒度 [s]	コード片 [s]
STEP-1 から STEP-3	6.7	6.6
STEP-4	4.3	-
STEP-5 から STEP-7	20.0	24.4
STEP-8 および STEP-9	2.3	-
STEP-10	12,988.8	130,328.5
STEP-11	12.8	5.5



(a) 複数のメソッドクローンを 1 つのファイルクローンとして検出した例



(b) メソッドクローンが検出されないファイルをファイルクローンとして検出した例



(c) ファイル内の一部のメソッドのみをメソッドクローンとして検出した例

図 **8**　分析しやすい検出結果の例

5.7　項目 3 の調査

粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成した例を以下に 3 つ示す．

1 つ目の例を図 8(a) に示す．メソッド単位のみで検出した場合，write，read，getFields 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．メソッドクローンペアを 1 つ 1 つ確認することで，これらのクラス内の全てのメソッドがクローンになっており，2 つのクラスを 1 つのクラスに集約するリファクタリングが可能であると判断できる．しかし，複数のメソッドクローンペアとしてこれら 2 つのクラスの重複が検出されるため，リファクタリングが可能かどうかの判断に時間を要する．多粒度な検出手法では，メソッド単位の検出前にファイル単位の検出を行うため，これら 2 つのクラスはファイルクローンであることが明らかになり，より容易にリファクタリングが可能かどうかの判断できる．多粒度な検出手法によって，5,278 個のメソッドクローンペアが 2,345 個のファイルクローンペアとして検出されたことを確認した．

2 つ目の例を図 8(b) に示す．メソッド単位のみやコード片単位のみで検出した場合，getName，getIfcName，getService メソッドはクローンとして検出されなかった．この理由は，2 つのクラス内の全メソッドが最小クローン長未満のメソッドであったためである．しかし，メソッド単位の検出に加えてファイル単位でも検出を行うことで，これら 2 つのクラスがクローンとして検出された．これらのクラスがファイルクローンとして検出されたことで，2 つのクラスを 1 つのクラスに集約するリファクタリングがで

きる可能性がある．そのため，メソッド単位のみのクローン検出では気づくことができないリファクタリングが可能となる．多粒度な検出手法によって，8,684 個のファイルクローンがこのようなケースで検出されたことを確認した．

3つ目の例を図8(c) に示す．ファイル単位のみで検出した場合，2 つのクラスはクローンとして検出されなかった．しかし，メソッド単位のクローン検出を行うことで，getColumnValues メソッドが検出された．DatabaseSelectAction と DatabaseDeleteAction クラスは DatabaseAction クラスを継承しており，getColumnValues メソッドに対してメソッド引き上げリファクタリングができる可能性がある．

以上のことから，多粒度な検出手法は，単一の粒度では気づくことができないリファクタリングが可能となる．ファイル，メソッド，コード片クローンに対して適用できるリファクタリングは異なるため，検出されたクローンの粒度が明確になることで，どのリファクタリングを適用できるかどうかをより容易に判断できる．

多数のクローンが検出されたため，すべてのクローンを確認はできてはいないが，目視で確認した範囲では，多粒度による検出が不利になると思われるクローンはなかった．

項目 **3** の結論

粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できることを示した．

6. 実験結果の妥当性について

対象ソフトウェア　本論文では，Java で記述された 84 個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，本論文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　本論文では，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[*3]．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，本論文では 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．

正規化の方法　本論文では，3 章で述べた正規化を行っている．正規化により，Type-2 クローンの検出が可能になるが，誤検出も増える．現時点では，正規化によりどの程度の誤検出が増えるのかは確認できていない．

7. おわりに

本論文では，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．また，提案手法をクローン検出ツール Decrescendo として実装し，複数のオープンソースソフトウェアに適用した．そして，多粒度な検出手法を粗粒度な検出手法および細粒度な検出手法と比較した．

比較の結果，以下の項目を示した．

- 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．
- 粗粒度な検出手法と比較して，多粒度な検出手法がクローンの検出数が多い．
- 細粒度な検出手法・細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できる．

今後はまず，被験者実験を行い多粒度な検出手法が生成した検出結果の分析のしやすさをさらに評価する予定である．検出において内部クラスを考慮する等のさらによりよい検出が行えるよう拡張することも考えられる．また，Java 以外の言語への拡張やその他のクローン検出ツールとの比較を行う予定である．さらには，より大規模なソースコードの集合に対して適用し，ライブラリの候補となるクローンや修正漏れの発見も試みる．

謝辞　本研究は，科学研究費補助金基盤研究 (S)(課題番号：25220003) および基盤研究 (B)(課題番号：17H01725) の助成を得て行われた．

[*3] 文を構成する文字列から算出したハッシュ値は，Smith-Waterman アルゴリズムで利用している．

参考文献

[1] Barbour, L., Khomh, F. and Zou, Y.: An empirical study of faults in late propagation clone genealogies, *Journal of Software: Evolution and Process*, Vol. 25, No. 11, pp. 1139–1165 (2013).

[2] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and evaluation of clone detection tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).

[3] Choi, E., Yoshida, N., Higo, Y. and Inoue, K.: Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files, *IEICE Transactions on Information and Systems*, Vol. 98, No. 2, pp. 325–333 (2015).

[4] Higo, Y. and Kusumoto, S.: How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods, *Proc. of the 22nd International Symposium on the Foundations of Software Engineering*, pp. 294–305 (2014).

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

[6] Kim, M., Sazawal, V., Notkin, D. and Murphy, G.: An empirical study of code clone genealogies, *Proc. of ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 5, pp. 187–196 (2005).

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

[8] Rattan, D., Bhatia, R. and Singh, M.: Software clone detection: A systematic review, *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199 (2013).

[9] Roy, C. K., Cordy, J. R. and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495 (2009).
[10] Sasaki, Y., Yamamoto, T., Hayase, Y. and Inoue, K.: Finding file clones in FreeBSD ports collection, *Proc. of the 7th Working Conference on Mining Software Repositories*, pp. 102–105 (2010).
[11] Svajlenko, J. and Roy, C. K.: Evaluating clone detection tools with BigCloneBench, *Proc. of the 31st International Conference on Software Maintenance and Evolution*, pp. 131–140 (2015).
[12] Weissgerber, P. and Diehl, S.: Identifying refactorings from source-code changes, *Proc. of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 231–240 (2006).
[13] 神谷，肥後，吉田：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).
[14] 石原，堀田，肥後，井垣，楠本：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).
[15] 村上，堀田，肥後，井垣，楠本：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).
[16] 肥後，楠本，井上：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).
[17] 堀田，楊，肥後，楠本：粗粒度なコードクローン検出手法の精度に関する調査，情報処理学会論文誌，Vol. 56, No. 2, pp. 580–592 (2015).