修士学位論文

題目

多粒度コードクローン検出ツール **Decrescendo** の実装と評価

指導教員
楠本　真二　教授

報告者
幸　佑亮

平成 29 年 2 月 7 日

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

平成 28 年度 修士学位論文

多粒度コードクローン検出ツール Decrescendo の実装と評価

幸　佑亮

内容梗概

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，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ための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のみコードクローンを検出している．一般的に，検出対象の粒度が粗いほど，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．

そこで本研究では，これら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．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することができる．

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，多粒度な検出手法を粗粒度な検出手法・細粒度な検出手法と比較して評価を行った．結果として，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，粗粒度な検出手法と比較して，多粒度な検出手法がコードクローンの検出数が多いことを示した．

主な用語

多粒度コードクローン検出手法
ファイルクローン
メソッドクローン
コード片クローン

目 次

図 目 次

表 目 次

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互いに同一，あるいは類似したコード片である．コード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][2][3][4]．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，そのコード片のコード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，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り，これまでに多くのコードクローン検出ツールが提案されている．

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 [5]．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コードクローンの存在が確認されている．

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．既存のコード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 [6]．

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

細粒度な検出手法：任意のコード片をコード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コードクローンとして検出することができる．

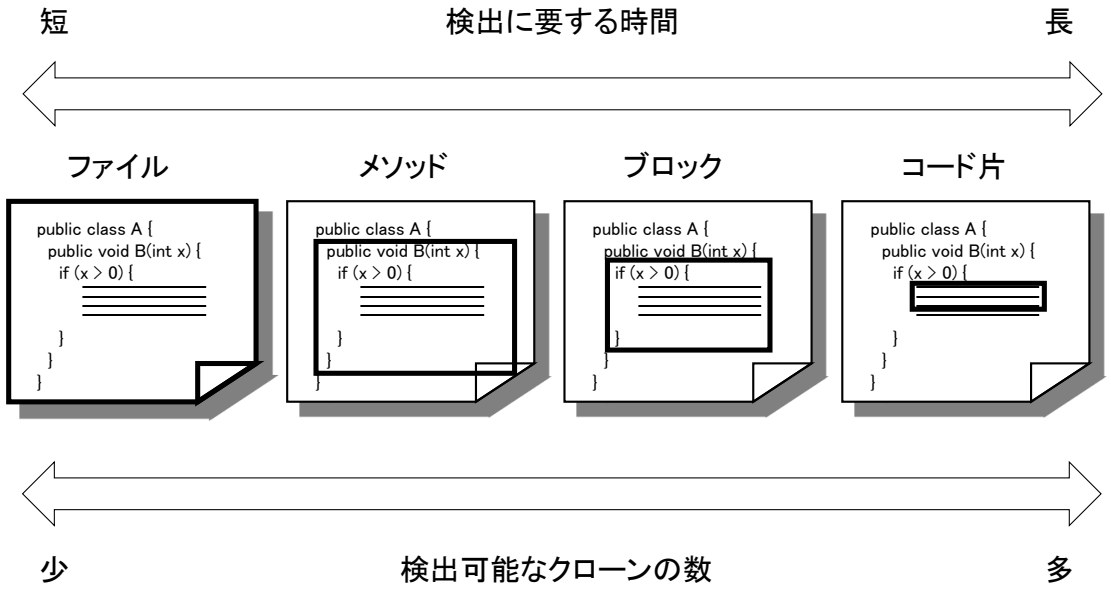

図 1: 各検出手法の特徴

これらの手法は以下の一長一短な特徴を持つ．ファイル単位・メソッド単位・ブロック単位・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：検出対象の粒度が粗いほど，検出時間が短く，検出対象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・ソースコード中のトークン数・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・メソッド・ブロックの数と比較して多い．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．

検出可能なコードクローンの数：検出対象の粒度が粗いほど，検出可能なコードクローンの数が少なく，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の数が多い．粗粒度な検出手法はファイル・メソッド・ブロック単位でコードクローンを検出するが，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コード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コードクローンの数が少ない．

そこで本研究では，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）コード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コードクローンを検出する．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．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．さらに，ソースコード中の連続した複数のコードクローンをより粒度の大きい段階で検出することで，1 つのコードクローンとしてまとめて検出できる．例えば，複数のメソッド単位のコードクローンを 1 つのファイル単位のコードクローンとして検出できる．これによって，検出されるコードクローンの数が少なくなり，コードクローンの分析に要する時間を短縮することができる．つまり，粗粒度な検出手法，細粒度な検出手法と比較してより分析しやすいコードクローンの検出結果を生成できる．

本研究では，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，多粒度な検出手法を粗粒度な検出手法・細粒度な検出手法と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．

- 粗粒度から細粒度へ段階的に（多粒度で）コード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コードクローンを検出できることを示した．
- 粗粒度な検出手法と比較して，多粒度な検出手法がコード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・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コードクローンの検出結果を生成できることを示した．

以降，2 章では準備について述べる．3 章では，提案手法の概要について述べる．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，実験方法や結果について述べる．6 章では実験結果の妥当性に対する評価について述べる．最後に，7 章では，本研究のまとめについて述べる．

2 準備

2.1 コードクローン

コードクローンとは，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．図 2 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにコードクローンであるという．また，ペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにコードクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [7].

ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

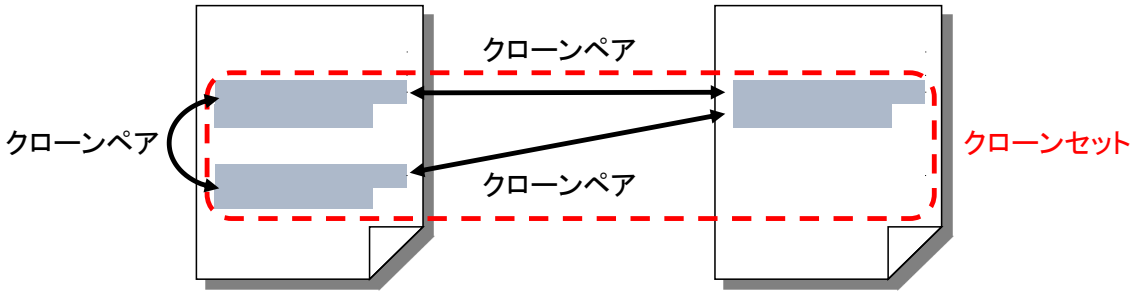

図 2: クローンペアとクローンセット

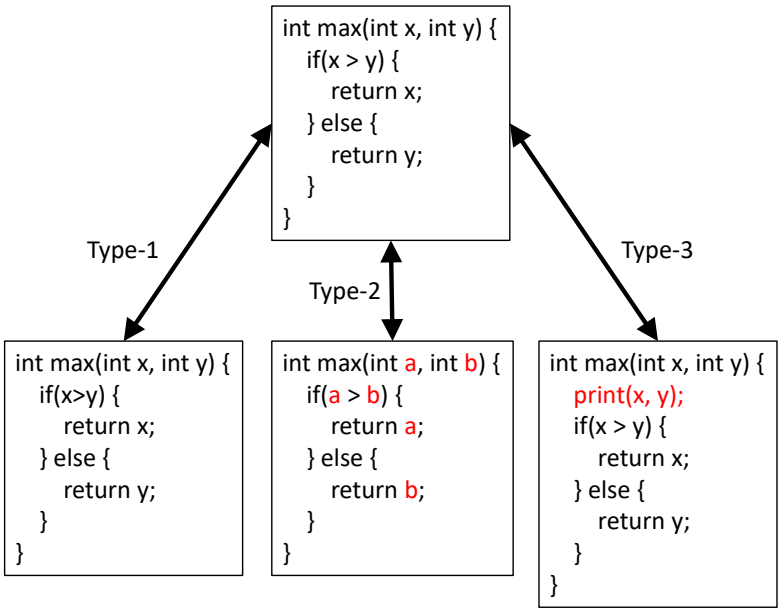

図 3: コードクローンの分類

また，コードクローン間の類似の度合いに基づき，以下の 3 つのタイプに分類できる [8]．図 3 に例を示す．

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，完全に一致するコードクローン．

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コードクローン．

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

2.2 コードクローンの主な発生の原因

コードクローンが発生する原因として次のようなものが挙げられる [9][10][11].

既存コードのコピーアンドペーストによる再利用

近年のソフトウェア設計手法を利用することにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンドペーストによる場当たり的な既存コードの再利用が多く行われるようになった．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである．

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用いた処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは，あらかじめ決められたコードをベースにして自動的に生成される．このため，類似した処理を目的としたコードを生成した場合，識別子名等の違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

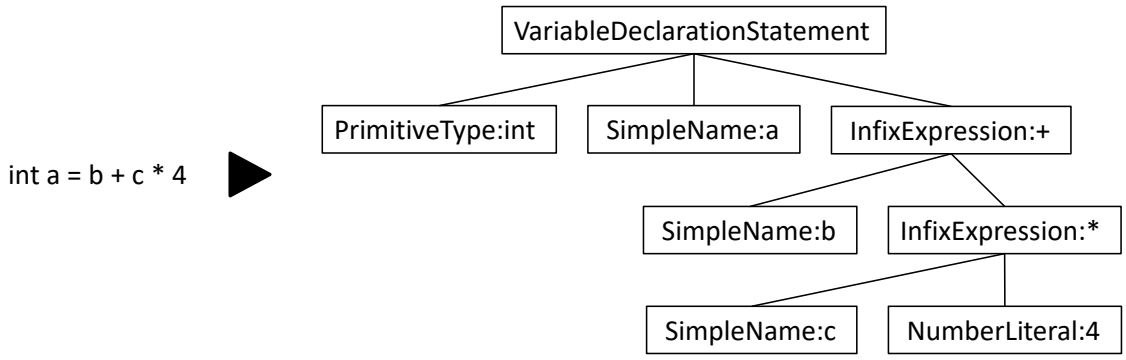

図 4: 抽象構文木の例

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.3 コードクローン検出手法

コードクローンを検出する手法はこれまでに多数提案されている．これらの検出技術はその検出単位によって，大まかに以下の 5 つに分類することができる [1][2].

行単位の検出

行単位の検出 [12][13][14][15][16][17][18] は，ソースコードを行単位で比較してコードクローンを検出する手法であり，閾値以上連続して一致する行をコードクローンとして検出する．しかし，同じ処理を行っているコードであっても，例えば長い行を複数行に分割した場合と分割しなかった場合など，コーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

字句単位の検出

字句単位の検出 [9][19][20][21] は，ソースコードを字句単位に分割し，閾値以上連続して一致する字句の部分列をコードクローンとして検出する手法である．行単位の検出と異なり，コーディングスタイルのみ違う場合などもコードクローンとして検出することが可能である．ソースコードを検出用の中間表現に変換する必要がないため，高速にコードクローン検出を行うことができるという利点もある．

抽象構文木を用いた検出

抽象構文木（図 4）を用いた検出 [10][22][23] は，ソースコードに対して構文解析を行い，抽象構文木を構築した後，その抽象構文木を用いてコードクローンを検出する手法であり，抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を構築するという事前処理を要するため，行単位の検出や字句単位の検出と比べ，時間的，空間的コストが高くなるという欠点がある．ある関数定義の終わりから次の関数定義の先頭までの類似部分など，プログラムの構造を無視した

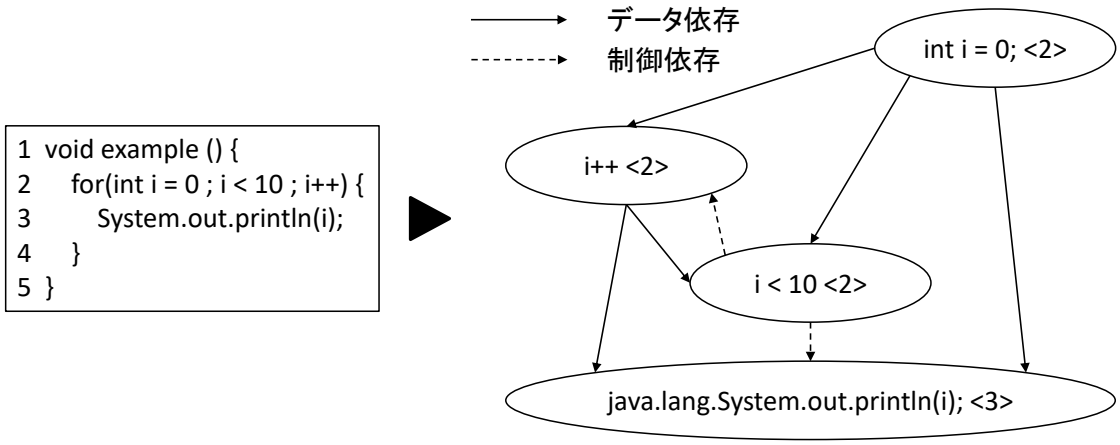

図 5: プログラム依存グラフの例

```
    fp = lookaheadset + tokensetsize;
    for (I = lookaheas(state) ; I < k ; i++) {
%       fp1 = LA + i * tokensetsize;
%       fp2 = lookaheadset;
%       while (fp2 < fp3)
%           *fp2++ |= fp1++;
    }
```

```
    fp3 = base + tokensetsize;
    …
    if (rp) {
        while ((j = *rp++) >= 0) {
            …
#           fp1 = lookaheadset;
#           fp2 = LA + j * tokensetsize;
#           while (fp1 < fp3)
#               *fp1++ |= *fp2++;
        }
    }
```

図 6: 順序入れ替わりコードクローンの例

コードクローンを検出しないという特徴を持つ．

プログラム依存グラフを用いた検出

プログラム依存グラフ（図 5）を用いた検出 [24][25][26] は，ソースコードに対して意味解析を行い，ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後，そのプログラム依存グラフを用いてコードクローン検出を行う手法である．プログラム依存グラフ上の同形部分がコードクローンとして検出される．抽象構文木を用いた検出と同様に事前処理を必要とするため，時間的，空間的コストが高くなるという欠点を持つ．ソースコードの順番が入れ替わっていても意味的に同一であるコードクローン（順序入れ替わりコードクローン）などは意味的な処理を考慮しなければ検出できないが，この手法はこれらのコードクローンを検出することができるという点が特徴として挙げられる．

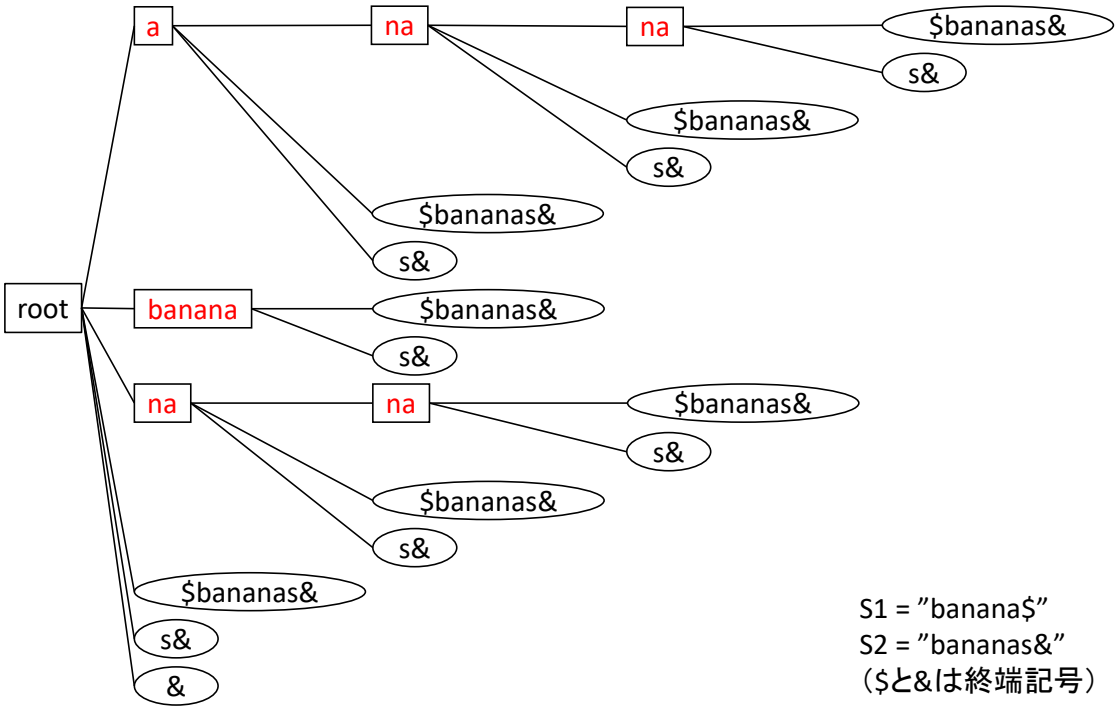

図 7: Suffix Tree アルゴリズムを適用した例

順序入れ替わりコードクローンの例を図 6 に示す．この例の場合，%で表されているコード片と，#で表されているコード片が順序入れ替わりコードクローンとなる．

その他の技術を用いた検出

その他の技術を用いた検出手法 [27][28][29][30] として，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法であるメトリクスを用いた検出や，プログラムの盗用の検出やプログラムの作者を特定することを目的とした，フィンガープリントやバースマークを用いた検出手法などがある．

2.4 コードクローン検出ツール

コードクローンを自動的に検出するツールがこれまでに多数開発されている．以下に，本研究で参考にしたコードクローン検出ツールについて述べる．

CCFinder

CCFinder[9][19] は，Kamiya らによって開発されたコードクローン検出ツールであり，字句単位の検出手法に分類される．ソースコードに対し事前処理を施すことで，ユーザ定義名，定数，名前空間，コンパウンドブロックの中括弧表記などの違いを吸収している．マッチング処理には，Suffix Tree アルゴ

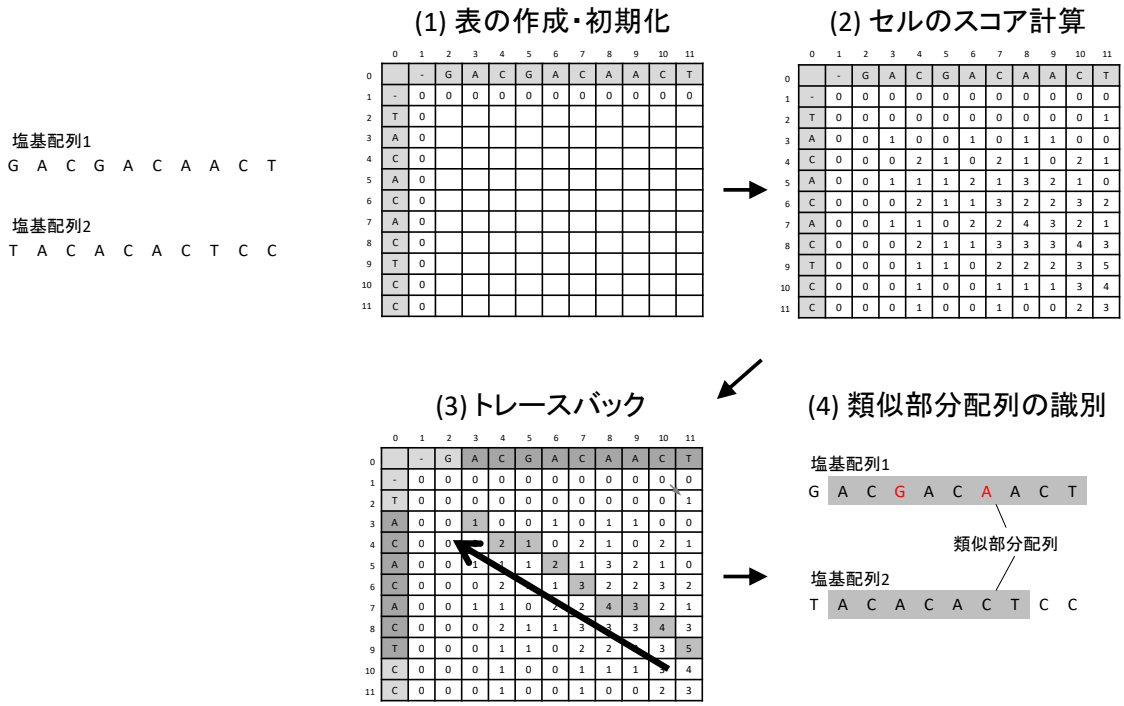

図 8: Smith-Waterman アルゴリズムを適用した例

リズム [31] を用いている．Suffix Tree アルゴリズムとは，与えられた文字列の接尾部を木構造で表すデータ構造である．図 7 は，”banana$” と”bananas&” という 2 つの文字列に対して Suffix Tree アルゴリズムを適用した例である．この Suffix Tree から”anana”，”ana”，”a”，”banana”，”nana”，”na” が共通部分文字列の候補であることが分かり，その内最長の”banana” が共通部分文字列となる．Suffix Tree アルゴリズムを用いることで線形時間での検出が可能になり，大規模ソフトウェアに対して実用的な時間とメモリ消費量で検出できることも特徴である．また，GUI フロントエンドである Gemini[32] により，コードクローンの散布図や実際のソースコードを見ることができる．

CDSW

CDSW[18] は，村上らによって開発されたコードクローン検出ツールであり，行単位の検出手法に分類される．マッチング処理には Smith-Waterman アルゴリズム [33] を用いている．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．図 8 は”GACGACAACT” と”TACACACTCC” という 2 つの塩基配列に対して Smith-Waterman アルゴリズムを適用した例である．Smith-Waterman アルゴリズムの手順を以下に示す．

(1) 表の作成・初期化：入力された 2 つの配列が $\langle a_1, a_2, \cdots, a_N \rangle$ と $\langle b_1, b_2, \cdots, b_M \rangle$ のとき，$(N+2) \times (M+2)$ の表を作成する．そして，表の一番上の行と一番左の列を入力された配列で埋める．さらに，2 番目の行と列を 0 で埋める．

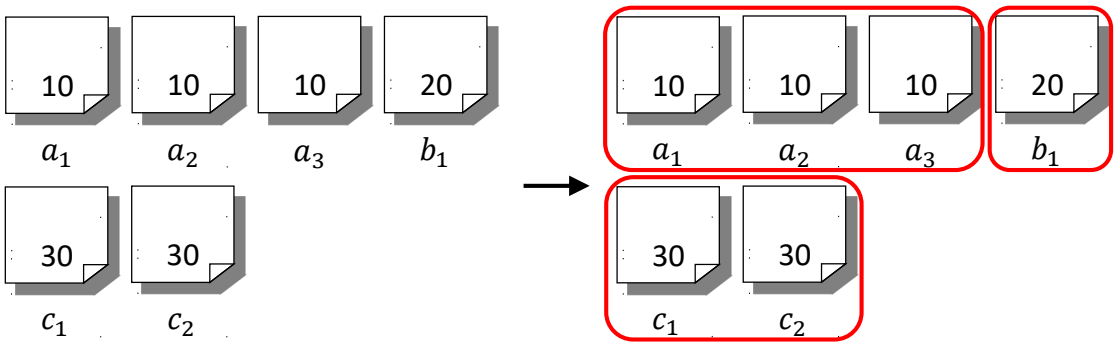

図 9: ファイルのハッシュ値に基づいたクラスタリング

(2) セルのスコア計算：以下の数式にしたがって残りのセルのスコアを計算する．

$$v_{i,j}(2 \leq i, 2 \leq j) = max \begin{cases} v_{i-1,j-1} + s(a_i, b_j), \\ v_{i-1,j} + gap, \\ v_{i,j-1} + gap, \\ 0. \end{cases} \tag{1}$$

$$s(a_i, b_j) = \begin{cases} match & (a_i = b_j), \\ mismatch & (a_i \neq b_j). \end{cases} \tag{2}$$

ここで，$c_{i,j}$ は i 行 j 列に位置するセル，$v_{i,j}$ は $c_{i,j}$ のスコア，a_i は一方の入力配列 $\langle a_1, a_2, \cdots, a_N \rangle$ の i 番目の要素，b_j はもう一方の入力配列 $\langle b_1, b_2, \cdots, b_M \rangle$ の j 番目の要素，$s(a_i, b_j)$ は a_i と b_j の類似度を表す．また，$match$，$mismatch$，gap はスコアパラメータを表している．

スコアパラメータの $match$，$mismatch$，gap には任意の値を設定できる．図 8 においては，説明を簡単にするため，($match$, $mismatch$, gap) にそれぞれ (1, -1, -1) を設定している．

(3) トレースバック：スコアが最大のセルからスコアが 0 のセルまで順に遡る．

(4) 類似部分配列の識別：トレースバックの際に遡ったセルに該当する要素が類似配列として識別される．図 8 において，”ACGACAACT” と”ACACACT” が類似配列として識別され，塩基配列 1 内の赤文字”G” と”A” は不一致部分として識別される．

2.5 先行研究

コード片単位（字句単位）のコード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 [34]．このクラスタリングはコード片単位の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 9 に示す．ファイル中の数値はハッシュ値を表しており，赤枠はハッシュ値が等しいファイルでクラスタリングされたことを表す．図 9 では，同じハッシュ値”10” を持つファイル a_1，a_2，a_3 が 1 つの集

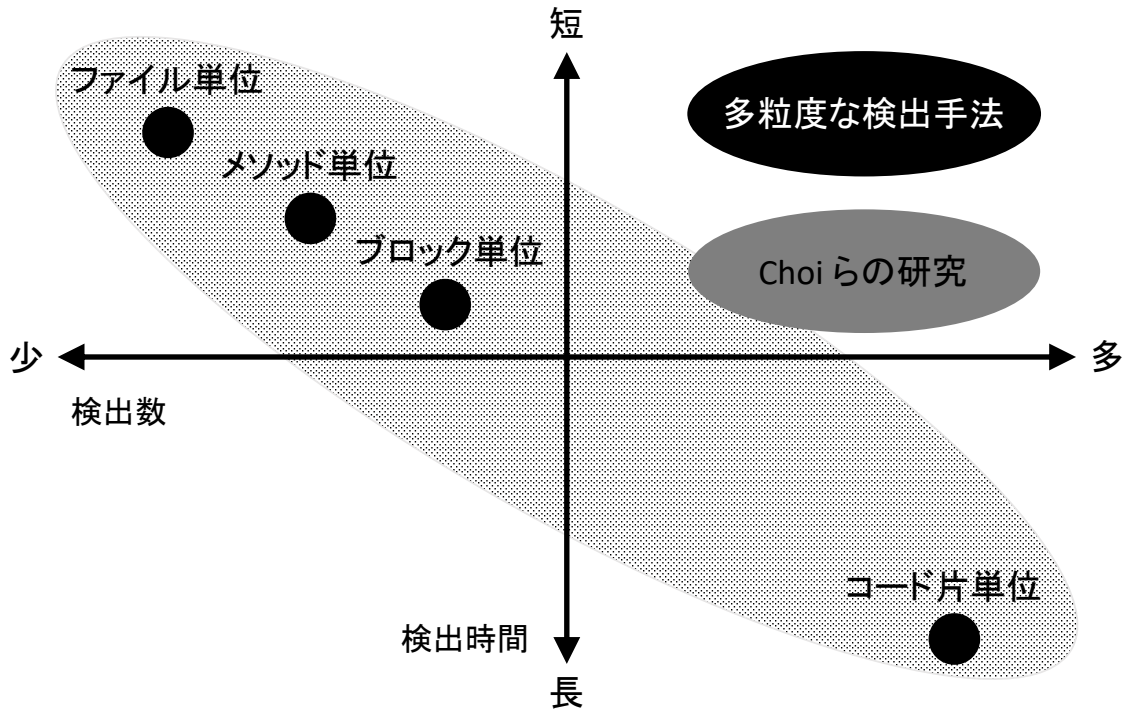

図 10: 本研究の位置づけ

合を形成している．同じハッシュ値”30” を持つファイル c_1，c_2 も同様に 1 つの集合を形成している．ファイル b_1 は単一の要素で集合を形成している．クラスタリング後，各部分集合から 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コード片単位のコードクローン検出を行う．このクラスタリングは，ファイル単位のコードクローン検出と同一である．しかし，メソッド単位のコードクローンと比較すると，ファイル単位のコードクローンの数は少ない．そのため，本研究の提案手法は，ファイル単位のコードクローンに加えて，メソッド単位のコードクローンを除外する．これによって，既存研究より効果的にコード片単位の検出時間を短縮することができる．さらに，本研究の提案手法は，前処理としてハッシュ値に基づいたファイルのクラスタリングを行うのではなく，ファイル単位のコードクローンとして検出している．よって，検出されたコード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コードクローンの検出結果を生成することができる．

2.6 本研究の位置づけ

既存のコードクローン検出手法と本研究で提案する多粒度な検出手法の位置づけを図 10 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，検出されるコード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，本研究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，本研究の提案手法は，ファイル単位のコードクローンに加えて，メソッド単位のコードクローンを除外する．これによって，Choi らの先行研究より効果的にコード片単位の検出時間を短縮する．

3 提案手法

粗粒度な検出手法は検出数が少ない．また，細粒度な検出手法は検出時間が長い．そこで本研究では，それらの各デメリットを低減する多粒度なコードクローン検出手法を提案する．

3.1 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粗粒度から細粒度へ段階的に（多粒度で）コード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コード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．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をメソッドクローン，コード片単位のコードクローンをコード片クローンと呼ぶ．

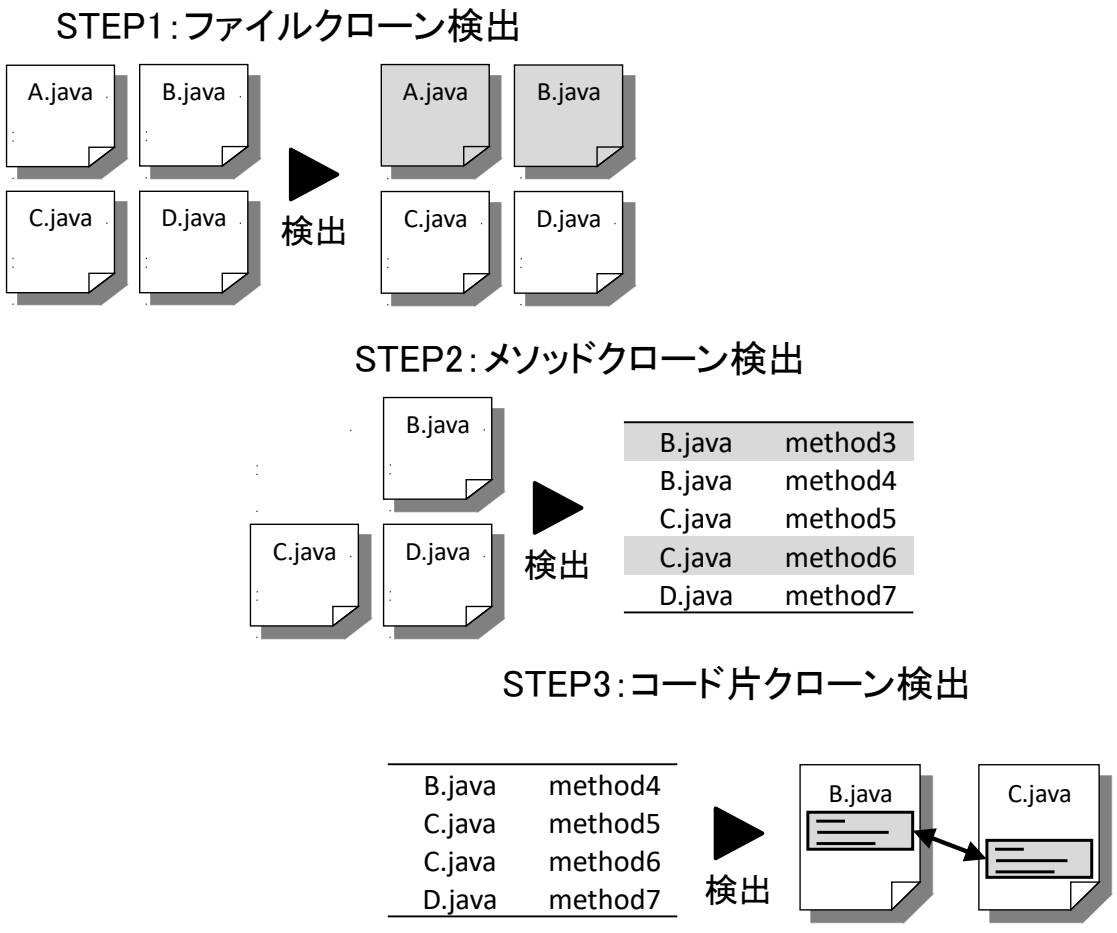

図 11: 提案手法の概要

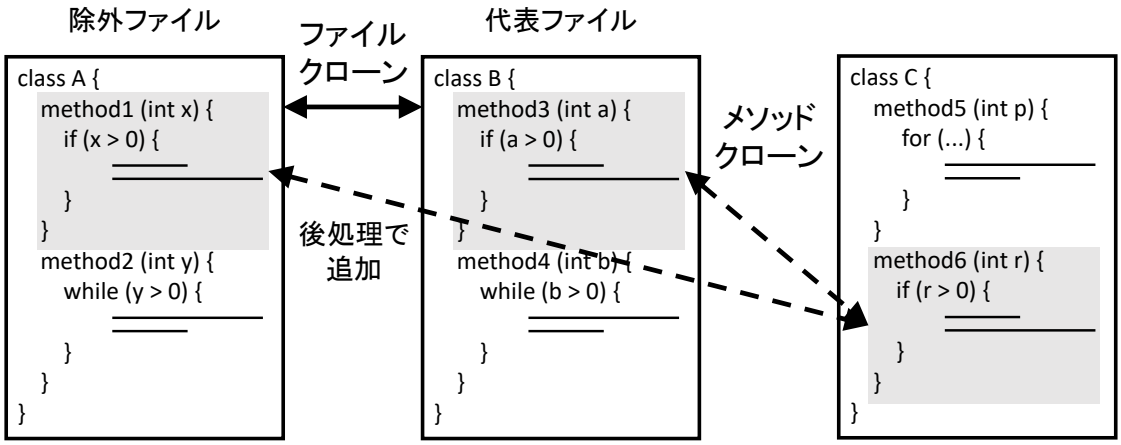

図 12: 後処理の例

図 11 に提案手法の概要を示す．段階的にコード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コード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 11 では，ファイル単位の検出において，A.java と B.java がファイルクローンとして検出されている．そこで，次のメソッド単位の検出では，ファイルクローンとして検出されなかったファイルを入力として与える．さらに，A.java と B.java のうち無作為に 1 つのファイルを選び，代表のファイルとして入力に加える．ここでは，B.java を入力に加えている．同様に，メソッド単位の検出でコード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 11 では，メソッド単位の検出において，B.java のメソッド 3 と C.java のメソッド 6 がメソッドクローンとして検出されている．そこで，次のコード片単位の検出では，メソッドクローンとして検出されなかったメソッドを入力として与える．さらに，メソッド 3 とメソッド 6 のうち無作為に 1 つのメソッドを選び，代表のメソッドとして入力に加える．ここでは，メソッド 6 を入力に加えている．

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 [5]．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．

また，提案手法では，メソッドクローンとコード片クローンの検出後に後処理を行う．例を図 12 に示す．まず，ファイル単位の検出時にクラス A とクラス B がファイルクローンとして検出されたと仮定する．また，クラス A が次のメソッド単位の入力から除外されたと仮定する．そして，次のメソッド単位の検出時にメソッド 3 とメソッド 6 がメソッドクローンとして検出された場合，メソッド 3 は代表ファイルであるクラス B 中に存在しているため，クラス B とコードクローンの関係にあるクラス A のメソッド 1 は，メソッド 3，メソッド 6 とコードクローンの関係にある．そこで，メソッド 1 とメソッド 6 をメソッドクローンとして検出結果に追加する．この後処理を行うことで，検

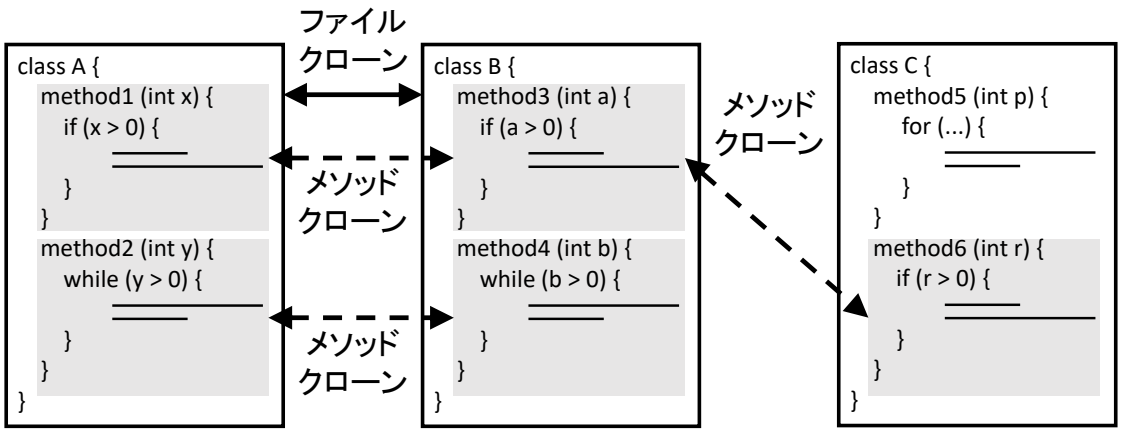

図 13: 提案手法の副次的なメリット

出済みのファイルを除外したことで生じる検出漏れを防ぐことができる．コード片単位の検出後においても同様の後処理を行う．

3.2 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コードクローンの数が少なくなる．例を図 13 に挙げる．例えば，メソッド単位のみで検出した場合，クラス A とクラス B 間では，メソッド 1 とメソッド 3，メソッド 2 とメソッド 4 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 B の 1 つのクローンペアで検出できる．しかし，ファイル単位のみで検出した場合，メソッド単位の検出時には検出できていたメソッド 3 とメソッド 6 をコードクローンとして検出できない．このことから，多粒度な検出手法によって，このような連続したコードクローンをより粒度の大きい段階で検出することで，検出されるコードクローンの数が少なくなり，分析しやすいコードクローンの検出結果を生成できる．また，単一の粒度では検出できないコードクローンを検出することができる．つまり，粗粒度な検出手法・細粒度な検出手法のどちらに対しても多粒度な検出手法は分析のしやすさ，検出数の面で有用である．

4 実装

3 章で述べた提案手法を基に多粒度コード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．

- 単一もしくは複数のソフトウェアのソースコード（Java で記述されたソースコードのみを対象）
- 最小クローン長（コード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位置情報とコードクローンの粒度およびタイプである．コードクローンの検出結果はデータベースに出力する．

ファイル単位・メソッド単位の検出手法として，ハッシュ値による比較を用いている [5][6]．コード片単位の検出手法として，Suffix Tree アルゴリズムと Smith-Waterman アルゴリズムを用いている．コード片単位の検出手法は，Decrescendo の設定によって切り替えることができる．

ファイル単位・メソッド単位の検出手法では，Type-1，Type-2 のコードクローンを検出する．コード片単位の検出手法のうち，Suffix Tree アルゴリズムを用いた検出の場合は，Type-1，Type-2 のコードクローンを検出する．Smith-Waterman アルゴリズムを用いた場合は，Type-1，Type-2，Type-3 のコードクローンを検出する．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 までのコードクローンを検出する手法と Type-3 までのコード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 までのコード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コード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 [35][36][37].

Type-3 までのコード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コードクローン検出ツールがその他の Type-3 コード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 [18].

Decrescendo が行う大まかな処理の流れは以下の通りである．

STEP1：ファイルクローン検出

STEP1-1：解析対象の特定
STEP1-2：正規化
STEP1-3：フィルタリング
STEP1-4：ハッシュ値の生成
STEP1-5：グループ化

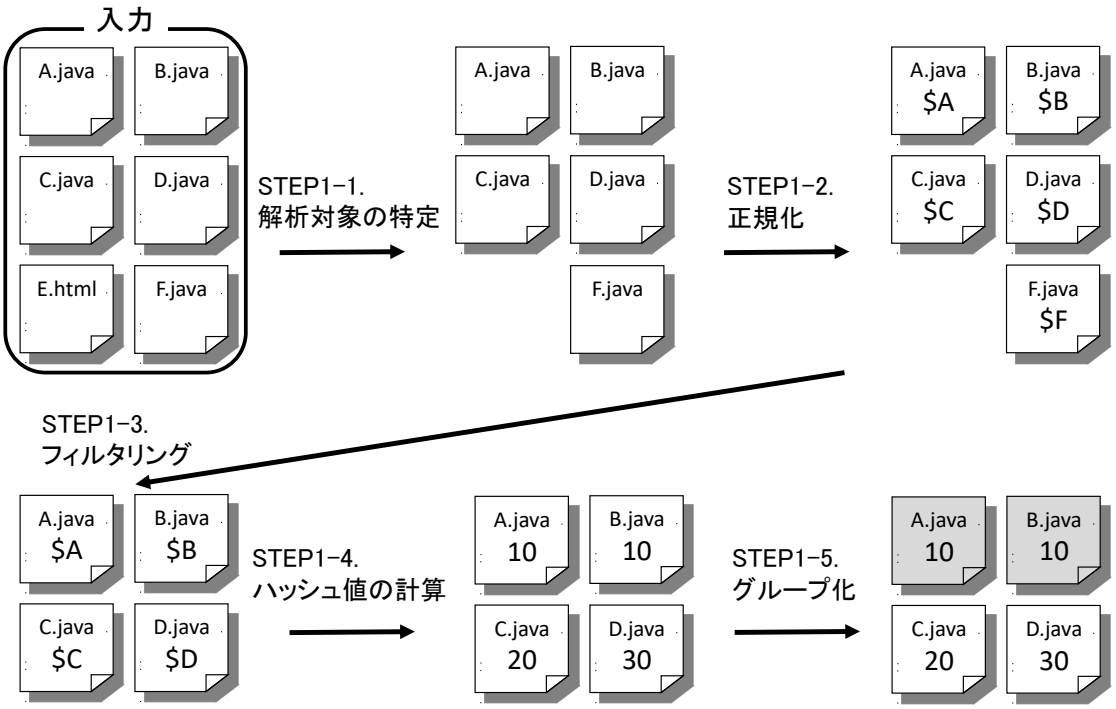

図 14: ファイルクローン検出の概要

STEP2：メソッドクローン検出

STEP2-1：メソッドの切り出し

STEP2-2：正規化

STEP2-3：フィルタリング

STEP2-4：ハッシュ値の生成

STEP2-5：グループ化・後処理

STEP3：コード片クローン検出

- Suffix Tree アルゴリズムを用いた検出の場合

STEP3-1：Suffix Tree の構築

STEP3-2：共通部分文字列の探索

STEP3-3：クローン情報の出力・後処理

- Smith-Waterman アルゴリズムを用いた検出の場合

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・後処理

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 の概要について述べる．

4.1 ファイルクローン検出

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 14 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 つのファイルをコード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．フィルタリングを行う理由は以下の通りである．

- STEP1-5 におけるマッチング処理に要する時間を短縮するため．
- 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．

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 14 では，A.java と B.java がファイルクローンとなる．ここで検出されるコードクローンのタイプは Type-1，Type-2 である．

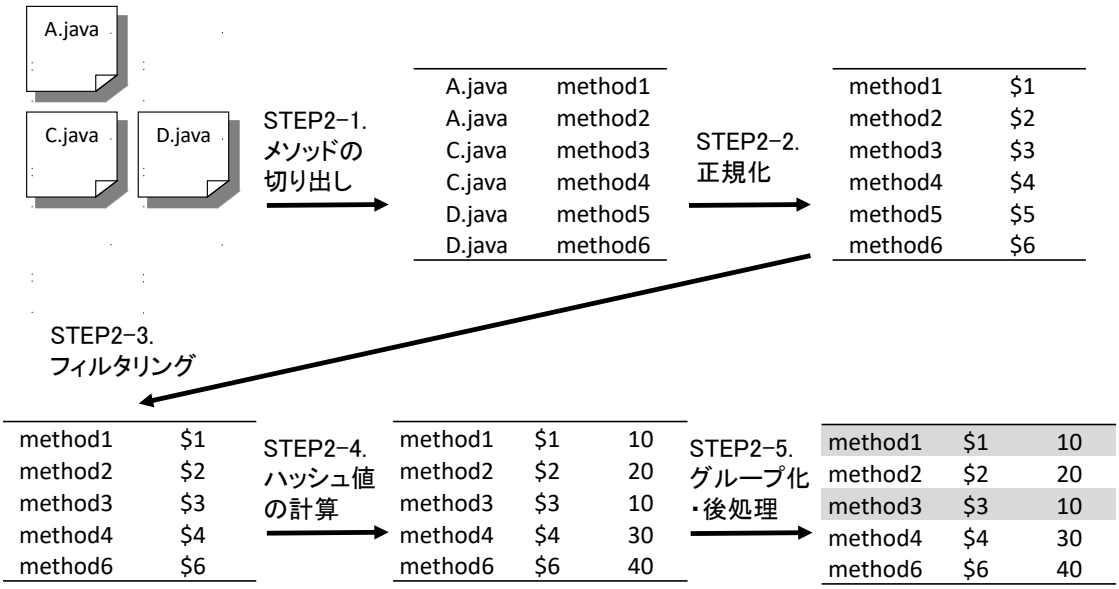

図 15: メソッドクローン検出の概要

4.2 メソッドクローン検出

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 15 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 14 では，A.java と B.java がファイルクローンである．図 15 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．また，入力は正規化前のファイルである．入力が正規化後のファイルの場合，メソッドクローンのタイプを分類できないためである．

そして，入力された各ファイルに対して抽象構文木を構築し，メソッドに該当するサブツリーを抽出する．

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・後処理

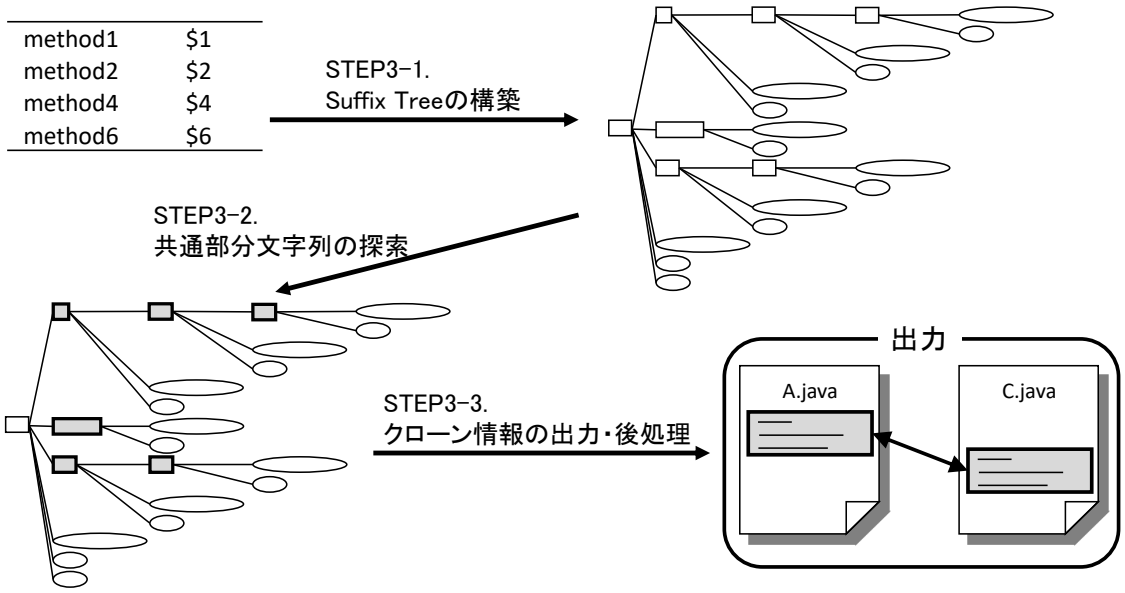

図 16: Suffix Tree アルゴリズムを用いたコード片クローン検出の概要

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 15 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．ここで検出されるコードクローンのタイプは Type-1，Type-2 である．

また，A.java は STEP1 にてファイルクローンとして検出されており，代表のファイルである．そのため，A.java とコードクローンの関係にある B.java 内のメソッドもメソッド 1，メソッド 3 とコードクローンの関係にある．そこで，後処理としてメソッド 1 とコードクローンの関係にある B.java 内のメソッドとメソッド 3 をメソッドクローンとして検出結果に追加する．

4.3 Suffix Tree アルゴリズムを用いたコード片クローン検出

Suffix Tree アルゴリズムを用いた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 16 に示す．

STEP3-1：Suffix Tree の構築

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 15 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 16 において，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，全てのメソッドに終端記号を付け加えて連結した文字列に対して Suffix Tree を構築する．構築の過程で葉を生成する際に，メソッドの識別子を付加することで，その葉が

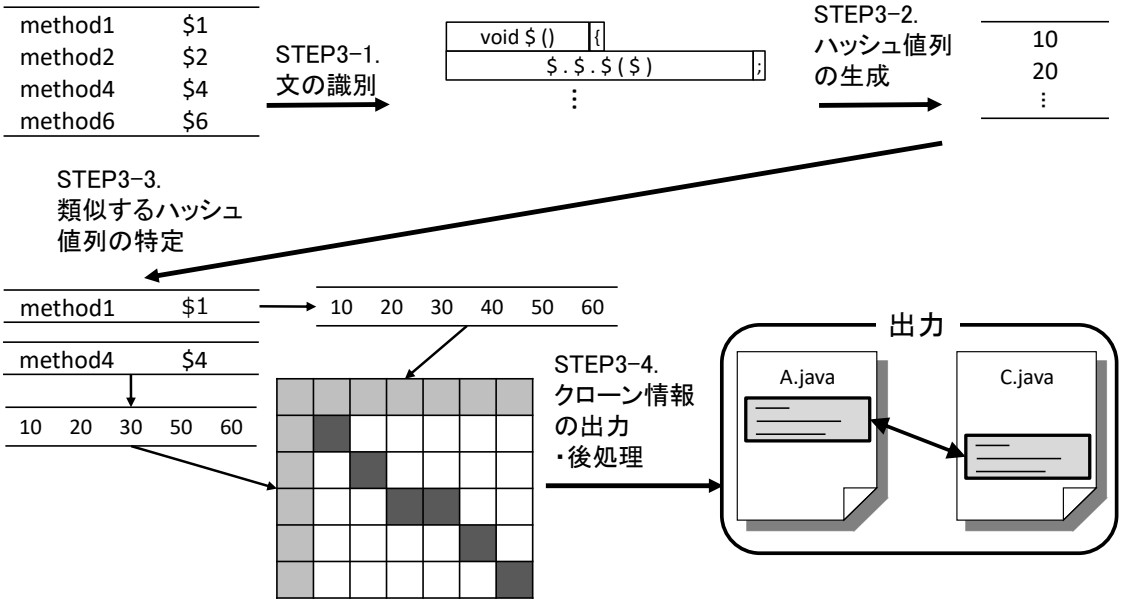

図 17: Smith-Waterman アルゴリズムを用いたコード片クローン検出の概要

どのメソッドの接尾辞を表しているかを区別する．

STEP3-2：類似部分文字列の探索

Suffix Tree を根から巡回し，共通部分文字列を探索する．Suffix Tree の場合，根と葉を除いた節には必ず 2 つ以上の子が存在する．よって，節を巡回して見つけた最長の部分文字列がその葉に該当するメソッドの類似部分文字列となる．

STEP3-3：コード片クローン情報の出力・後処理

STEP3-2 で特定した類似部分文字列に含まれるトークン数が最小クローン長以上の場合，クローンペアの位置情報とタイプを出力する．ここで検出されるコードクローンのタイプは Type-1，Type-2 である．そして，コード片クローンが代表のファイル・メソッドから検出された場合は，メソッド単位の検出時と同様の後処理を行う．

4.4 Smith-Waterman アルゴリズムを用いたコード片クローン検出

Smith-Waterman アルゴリズムを用いた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 17 に示す．

STEP3-1：文の識別

入力は，Suffix Tree アルゴリズムを用いたコード片単位クローンの検出時と同じである．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

Smith-Waterman アルゴリズムを用いて，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 3），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 4），クローンペアの位置情報とタイプを出力する．ここで検出されるコードクローンのタイプは Type-1，Type-2，Type-3 である．そして，コード片クローンが代表のファイル・メソッドから検出された場合は，メソッド単位の検出時と同様の後処理を行う．

$$match \geq \theta \tag{3}$$

$$gap/match \leq \phi \tag{4}$$

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・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 [8][38] を参考にしている．

5.2 調査項目

調査項目 1 : 粗粒度な検出手法・細粒度な検出手法と比較して，多粒度な検出手法は検出時間が短いか．

調査項目 2 : 粗粒度な検出手法・細粒度な検出手法と比較して，多粒度な検出手法は検出されるコードクローンの数が多いか．

調査項目 3 : 粗粒度な検出手法・細粒度な検出手法と比較して，多粒度な検出手法は分析しやすいコード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 2.90GHz Intel Xeon CPU（16 プロセッサ）であり，メモリサイズは 352.0GB 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

	APACHE
プロジェクト数	84
ファイル数	66,724
メソッド数	628,219
LOC	11,545,556

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 のリポジトリ [39] 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コード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．これらのプロジェクトを対象とした理由は，先行研究 [40] におけるコードクローン検出に用いられているためである．加えて，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，コードクローンとして検出する必要性がない．

5.5 実験結果

表 2 にコードクローンの検出結果を示す．粒度の項目下のチェックマークは検出のオン・オフを表す．ST，SW は Suffix Tree アルゴリズム，Smith-Waterman アルゴリズムを用いたコード片単位の検出を表す．以降，各調査項目ごとに結果を述べる．

5.5.1 調査項目 1：検出時間

コード片単位の検出手法が Suffix Tree アルゴリズムの場合

表 2: コードクローンの検出結果

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		ファイル	メソッド	コード片	合計	
		ST	SW					
✓				11,029	-	-	11,029	7.3
	✓			-	49,446	-	49,446	32.8
		✓		-	-	1,637,597	1,637,597	62,569.0
			✓	-	-	446,442	446,442	130,367.1
✓	✓			11,029	44,168	-	55,197	32.3
✓		✓		11,029	-	1,632,319	1,644,348	46,183.2
	✓	✓		-	49,446	1,588,151	1,637,597	1,885.7
✓	✓	✓		11,029	44,168	1,588,151	1,643,348	3,094.6
✓			✓	11,029	-	441,164	452,193	109,717.1
	✓		✓	-	49,446	396,996	446,442	13,013.6
✓	✓		✓	11,029	44,168	396,996	452,193	13,036.7

ファイル単位，メソッド単位で検出した場合と全粒度で検出した場合を比較すると，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合と全粒度で検出した場合を比較すると，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．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，コード片単位の検出における STEP3-1 から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，実行時間が約 1/30 倍になっている．どちらの場合も検出時間の大半をコード片単位における STEP3-1 から STEP3-2 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 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，STEP2-5 と STEP3-3 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，後処理を行っ

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]
STEP1-1 から STEP1-4	6.7	6.4
STEP1-5	4.0	-
STEP2-1 から STEP2-4	24.0	31.7
STEP2-5	1.2	-
STEP3-1 から STEP3-2	2,980.0	62,508.9
STEP3-3	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]
STEP1-1 から STEP1-4	6.7	6.6
STEP1-5	4.3	-
STEP2-1 から STEP2-4	20.0	24.4
STEP2-5	2.3	-
STEP3-1 から STEP3-3	12,988.8	130,328.5
STEP3-4	12.8	5.5

ている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・コード片単位で検出した場合（46,183.2 秒）とメソッド・コード片単位で検出した場合（1,855.7 秒）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．

コード片単位の検出手法が Smith-Waterman アルゴリズムの場合

ファイル単位，メソッド単位で検出した場合と全粒度で検出した場合を比較すると，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 →13,036.7 秒）．

また，コード片単位で検出した場合と全粒度で検出した場合を比較すると，全粒度で検出した場合は検出速度が大幅に向上した（130,367.1 秒 →13,036.7 秒）．表 4 に各処理に要した時間を示す．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短くなった（24.4 秒 →20.0 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，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（130,328.5 秒 →12,988.8 秒）．全粒度で検出した場合，実行時間が約 1/10 倍になっている．どちらの場合も検出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 Smith-Waterman 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，STEP2-5 と STEP3-4 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，後処理を行っている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・コード片単位で検出した場合（109,717.1 秒）とメソッド・コード片単位で検出した場合（13,013.6 秒）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．

調査項目 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より高速にコード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コード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 [5][41][42]．そのような場合において，多粒度な検出手法は検出時間を短縮することに特に有効であると想定される．

5.5.2 調査項目 2：検出数

コード片単位の検出手法が Suffix Tree アルゴリズムの場合

ファイル単位，メソッド単位で検出した場合と全粒度で検出した場合を比較すると，全粒度で検出した場合は検出数多かった（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と全粒度で検出した場合を比較すると，検出数にあまり変化

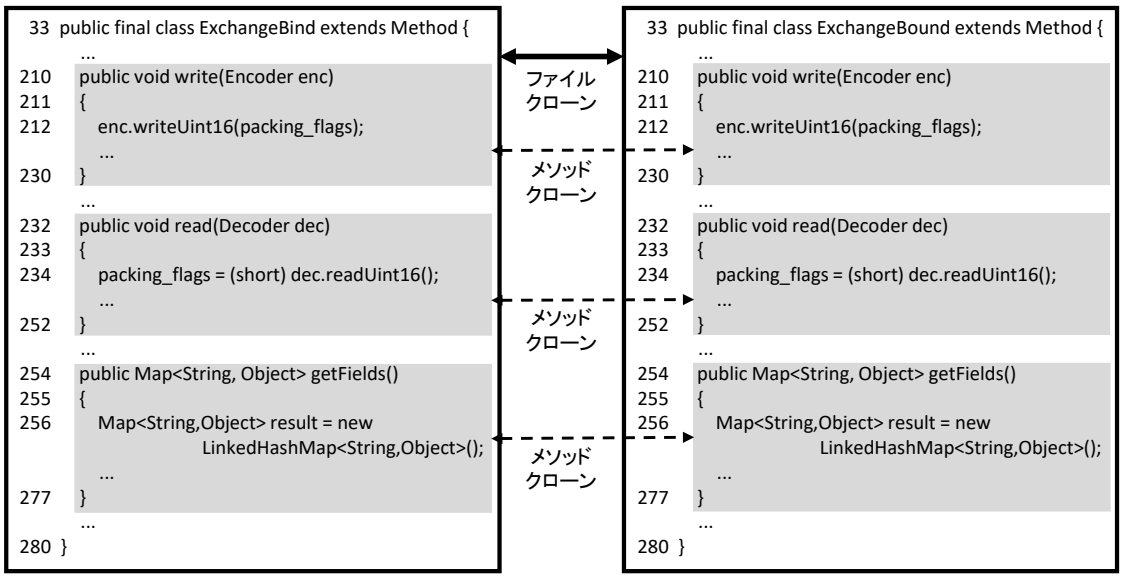

図 18: 分析しやすい検出結果の例 1

はなかった（1,637,597 個 →1,643,348 個）．この結果と調査項目 1 の結果から全粒度で検出した場合は，コード片単位で検出した場合と比較して，検出されるコードクローンの数を減らすことなく検出時間が短縮できていることが分かった．

コード片単位の検出手法が Smith-Waterman アルゴリズムの場合

ファイル単位，メソッド単位で検出した場合と全粒度で検出した場合を比較すると，全粒度で検出した場合は検出数多かった（11,029 個，49,446 個 →452,193 個）．

また，コード片単位で検出した場合と全粒度で検出した場合を比較すると，検出数にあまり変化はなかった（446,442 個 →452,193 個）．この結果と調査項目 1 の結果から全粒度で検出した場合は，コード片単位で検出した場合と比較して，検出されるコードクローンの数を減らすことなく検出時間が短縮できていることが分かった．

調査項目 2 の結果

これらの結果から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コードクローンを検出できることを示した．これはコードクローンの検出数を減らすことなく検出時間が短縮できていることを示す．

5.5.3 調査項目 3：分析のしやすさ

粗粒度な検出手法・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コード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 18 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

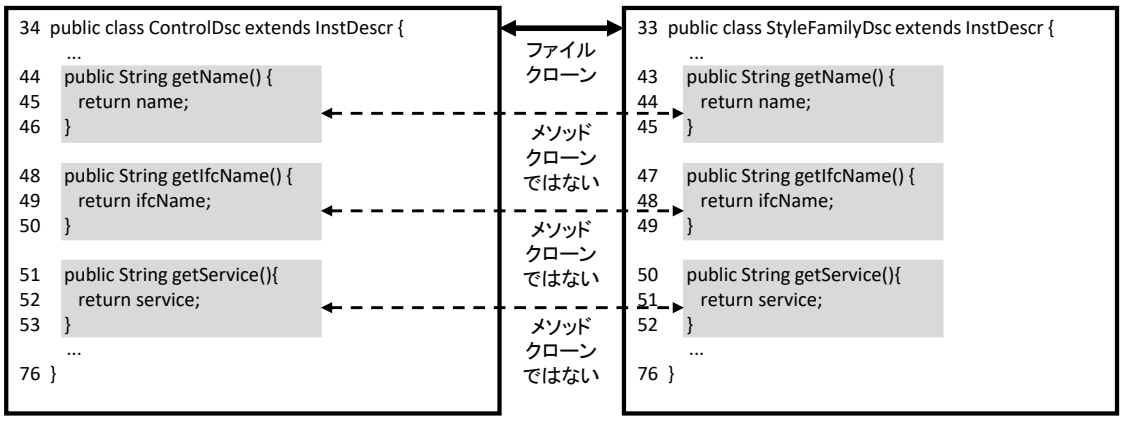

図 19: 分析しやすい検出結果の例 2

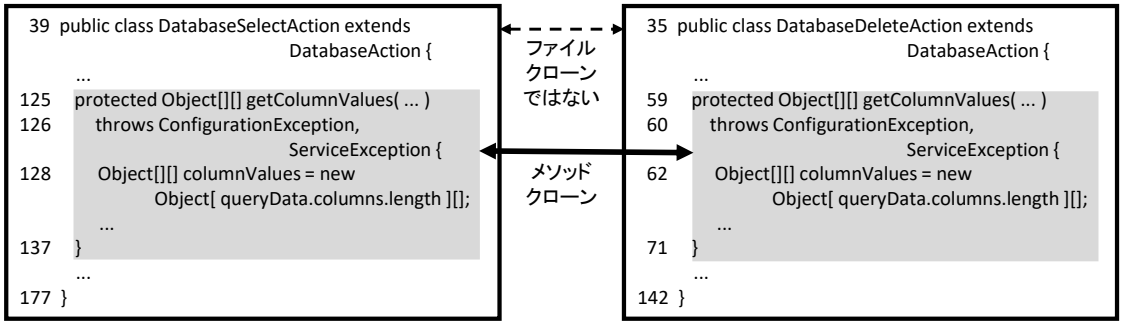

図 20: 分析しやすい検出結果の例 3

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 つ分析することで，ExchangeBind・ExchangeBound クラス内の全てのメソッドがコードクローンの関係にあり，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，メソッドクローン以外のコード片も分析する必要があり，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っているため，ExchangeBind・ExchangeBound 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をすることができる．多粒度な検出手法によって，5,278 個のメソッドクローンが 2,345 個のファイルクローンとして検出されたことを確認した．

2 つ目の例を図 19 に示す．メソッド単位のみで検出した場合，getName・getIfcName，getService メソッドは検出されなかった．これは，ControlDsc・StyleFamilyDsc クラス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，メソッド単位の検出前にファイル単位の検出を行うことで，ControlDsc・StyleFamilyDsc クラスをファイル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 20 に示す．ファイル単位のみで検出した場合，DatabaseSelectAction･DatabaseDeleteAction クラスは検出されなかった．しかし，メソッド単位の検出を行うことで，getColumnValues メソッドが検出された．DatabaseSelectAction・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．このことから，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．また，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それぞれ異なる．検出されたコード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することができる．

調査項目 3 の結果

粗粒度な検出手法・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コード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対する評価

対象ソフトウェア

本研究では，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，誤ったコード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．

7 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）コード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，多粒度な検出手法を粗粒度な検出手法・細粒度な検出手法と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．

- 粗粒度から細粒度へ段階的に（多粒度で）コード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コードクローンを検出できることを示した．
- 粗粒度な検出手法と比較して，多粒度な検出手法がコード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・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コードクローンの検出結果を生成できることを示した．

今後の課題は以下の通りである．

- 被験者実験によって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評価
- Java 以外の言語に拡張．
- その他のコード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．

謝辞

本研究を行うにあたり，理解ある親身なご指導を賜り，常に励まして頂きました 楠本 真二 教授に心より感謝申し上げます．

本研究の全過程を通して，終始熱心かつ丁寧なご指導を頂き，多大なるご助力，ご協力を頂きました 肥後 芳樹 准教授に心より感謝申し上げます．

本研究を行うにあたり，多くのご意見を頂き，また日々の研究室生活を盛り上げて頂きました 柗本 真佑 助教 に心より感謝申し上げます．

研究室生活の中で，事務作業を行う際に多大なるご支援を頂きました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楠本研究室事務員の 神谷 智子 氏，同 中埜 由美 氏に深く感謝申し上げます．

本研究を行うにあたり，ご指導，ご協力を頂き，さらに日常でも声をかけて頂き，様々な問題に対して深い理解のもと助言を頂きました，卒業生の 堀田 圭佑 氏，同 村上 寛明 氏，同 楊 嘉晨 氏，同 江川 翔太 氏，同 大谷 明央 氏，同 高 良多朗 氏に深く感謝申し上げます．

本研究を行うにあたり，常に様々な相談に乗って頂き，また日々の研究室生活を豊かにして頂きました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博士前期課程 2 年の 小倉 直徒 氏，同 佐飛 祐介 氏，同 鷲見 創一 氏，同 古田 雄基 氏，同 横山 晴樹 氏に心より感謝申し上げます．

本研究を行うにあたり，様々な場面で心強いお力添えを頂き，また日々の研究室生活を豊かにして頂きました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博士前期課程 1 年の 下仲 健斗 氏，同 中島 弘貴 氏，同 山田 悠斗 氏，同 山本 将弘 氏に心より感謝申し上げます．また，本研究で実装したツール名を考案して頂いた 山本 将弘 氏に加えて心より感謝申し上げます．

本研究を行うにあたり，様々な場面において親切なご助力を頂きました，大阪大学基礎工学部情報科学研究科 4 年の 有馬 諒 氏，同 佐々木 美和 氏，同 谷門 照斗 氏，同 松尾 裕幸 氏，同 山田 涼太 氏に心より感謝申し上げます．

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[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] Chanchal K Roy, James R Cordy, and Rainer Koschke. 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.

[4] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[5] 石原知也, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二ほか. 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.

[6] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二. 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.

[7] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータ ソフトウェア, Vol. 18, No. 5, pp. 529–536, 2001.

[8] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[9] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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.

[10] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. of the 14th IEEE International Conference on Software Maintenace*, pp. 368–377, 1998.

[11] Shinji Uchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Ken-Ichi Matsumoto, and Hideo Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. 45, No. 3, pp. 1–11, 2005.

[12] J Howard Johnson. Substring matching for clone detection and change tracking. In *Proc. of the 15th International Conference on Software Maintenance*, Vol. 94, pp. 120–126, 1994.

[13] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the 15th IEEE International Conference on Software Maintenance*, pp. 109–118, 1999.

[14] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 1, pp. 37–58, 2006.

[15] Brenda S Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pp. 49–49, 1993.

[16] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Proc. of the 2nd Working Conference on Reverse Engineering*, pp. 86–95, 1995.

[17] Brenda S Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1343–1362, 1997.

[18] 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.

[19] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.

[20] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, Vol. 8, No. 11, p. 1016, 2002.

[21] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, Vol. 32, No. 3, pp. 176–192, 2006.

[22] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 253–262, 2006.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of the 29th International Conference on Software Engineering*, pp. 96–105, 2007.

[24] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. of the 8th Working conference on Reverse Engineering*, pp. 301–309, 2001.

[25] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, 2001.

[26] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線, Vol. 2009, pp. 97–104, 2009.

[27] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the 12th IEEE International Conference on Software Maintenace*, Vol. 96, pp. 244–253, 1996.

[28] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, Vol. 3, pp. 77–108, 1996.

[29] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. of the 4th Working Conference on Reverse Engineering*, pp. 44–54, 1997.

[30] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of the 16th International Conference on Program Comprehension*, pp. 172–181, 2008.

[31] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[32] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proc. of the 8th International Symposium on Software Metrics*, pp. 67–76, 2002.

[33] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, Vol. 147, No. 1, pp. 195–197, 1981.

[34] Eunjong Choi, Norihiro Yoshida, Yoshiki Higo, and Katsuro Inoue. 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.

[35] Liliane Barbour, Foutse Khomh, and Ying Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, Vol. 25, No. 11, pp. 1139–1165, 2013.

[36] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proc. of ACM SIGSOFT Software Engineering Notes*, Vol. 30, pp. 187–196, 2005.

[37] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proc. of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 231–240, 2006.

[38] Jeffrey Svajlenko and Chanchal K Roy. Evaluating clone detection tools with bigclonebench. In *Proc. of the 31st International Conference on Software Maintenance and Evolution*, pp. 131–140, 2015.

[39] Apache. http://www.apache.org/.

[40] Yoshiki Higo and Shinji 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*, pp. 294–305, 2014.

[41] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in freebsd ports collection. In *Proc. of the 7th Working Conference on Mining Software Repositories*, pp. 102–105, 2010.

[42] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 283–292, 2011.