特別研究報告

題目

コードの同時修正作業に着目した
コードクローン検出結果の比較表示ツール

指導教員
肥後 芳樹 教授

報告者
小林 亮太

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

コードの同時修正作業に着目したコードクローン検出結果の比較表示ツール

小林 亮太

内容梗概

ソースコード中に存在する互いに一致または類似するコード片であるコードクローンは，バグの複製や修正コストの増大などの理由から，ソフトウェア保守における大きな問題点の 1 つとして指摘されている．既存のコードクローン検出ツールは，検出手法や検出時のパラメータなどの違いにより，同一のソースコード集合に対して得られる結果が異なることが知られている．このため，ある検出ツールで発見されるコードクローンが別の検出ツールでは発見されない場合がある．この問題に対処するため、検出ツールの結果の差異や共通部分を明確にすることを目的として，複数の検出ツールを同一の環境で使用し，それらの検出結果を比較することができる分析ツール CCX が開発されている．

しかし，コードクローン検出結果を利用する目的であるソフトウェア保守を考えた際，CCX の比較結果の表示方法では，特定のファイル探索の難しさや，全てのコードクローンの確認の手間，および比較後のコードクローン検出数が分からないといった問題点が存在した．そのため，コードクローンの同時修正を行う際に，多くの時間とコストを必要としていた．

そこで本報告では，これらの既存ツールの問題点を踏まえ，異なる 2 種類のコードクローン検出結果をコードクローン単位で比較し，それらをファイルごとに表示する手法を提案した．また，その提案手法を，CCX を拡張することで，ツールとして実装した．

また，作成したツールの評価実験を行った。具体的には，4 つの異なるソフトウェアに対し 2 種類の検出ツールを用いてコードクローン検出を行った結果を，開発したツールと既存ツールを用いて表示させ，その内容を用いてコードクローン分析を行う際の工数を測定した。その結果，既存ツールを使用するよりも開発したツールを使用した方が，同時修正作業のために行う画面遷移回数や確認コードクローン数が少ないことを確認した．

主な用語

コードクローン
検出結果の比較
同時修正

目 次

1　まえがき

コードクローンとは，ソースコード中に存在する互いに一致または類似するコード片のことである [15]．コードクローンは，主に既存のソースコードのコピーアンドペーストや，コード生成ツールによる自動生成によって生じる．これらのコードクローンは，既存研究において，ソフトウェア保守における大きな問題点の 1 つとして指摘されている [3, 5]．その問題点の 1 つに，コードクローンにおける修正コストの増大が挙げられる．あるコード片に欠陥が含まれている場合，そのコードクローンにも同様の欠陥が含まれている可能性が高い．そのため，そのようなコード片を修正する際，そのコードクローン全てに対して，同様の修正を行うか検討する必要があり，コストが増大する．

ソースコード中から目視でコードクローンを探すことは非現実的である．そのため，コードクローンを自動で検出するための様々な手法が提案され，それらの手法を採用した複数の検出ツールが既存研究で開発されている [4, 6, 9, 12]．しかし，これらの検出ツールは，同一のソースコード集合に対しても，検出ツールの特性の違いによって，得られる検出結果が異なることが明らかになっている [10]．そのため，様々な検出ツールでコードクローン検出を行い，得られた結果の共通部分や差異を知ることは重要である．そこで，松島らは，異なる 2 種類の検出ツールの結果を比較するクローンペアマッピングを提案した [13]．また，松島らは，複数の検出ツールを同一の環境で動作させることができ，それらの検出結果を，クローンペアマッピングを用いて比較することができるコードクローン分析ツール CCX を開発した [14]．これにより，同一のソースコード集合に対して，複数の検出ツールを使用し，その検出結果を比較・表示することが容易になった．

しかし，コードクローン検出結果を利用する目的である「ソフトウェア保守」を考えた際，CCX の比較結果の表示方法は以下のような問題点が存在した．

- 特定のファイルを探すことが困難な点
- 全てのコードクローンの確認に手間がかかる点
- 比較後のコードクローン検出数が分からない点

これらの既存ツールの問題点を踏まえ，本報告では，異なる 2 種類のコードクローン検出結果をコードクローン単位で比較し，それらのコードクローン合計数やクローンセットをファイルごとに表示する手法を提案した．また，その提案手法を，CCX を拡張することで，ツールとして実装した．

作成したツールの評価実験では，Sweet Home 3D[1], jEdit[2], Process Hacker[3]および snns[4] の 4 つのソフトウェアを対象としたコードクローン分析を行った．まず，それぞれのソフトウェアに対して，CCFinderSW[18] と NiCad[6] を用いてコードクローン検出を行う，次に，2 つのツールで検出した結果を，既存ツールと開発したツールを用いて比較し，表示・分析を行った．これにより，既存ツールを使用するよりも開発したツールを使用した方が，同時修正作業のために行う画面遷移回数や確認コードクローン数が少ないことを確認した．

以降，2 章では本報告の背景として，コードクローンと，既存ツール CCX について述べる．3 章では，作成したツールの基盤となったアイデアと，そのアイデアの正当性を示すための調査について述べる．4 章では，作成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．最後に 6 章では，まとめと今後の課題について述べる．

[1] http://www.sweethome3d.com/
[2] http://www.jedit.org/?page=features
[3] https://processhacker.sourceforge.io/
[4] http://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

2　背景

本章では，本報告の背景として，コードクローンの定義と，異なる検出ツールの検出結果の比較およびコードクローン分析ツールについて述べる．

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一致または類似するコード片のことである [15]．コードクローンは，主に既存のソースコードのコピーアンドペーストや，コード生成ツールによる自動生成によって生じる．図 1 にコードクローンの例を示す．互いにコードクローンの関係となる 2 つのコード片の組をクローンペアと呼び，コードクローンの集合をクローンセットと呼ぶ．

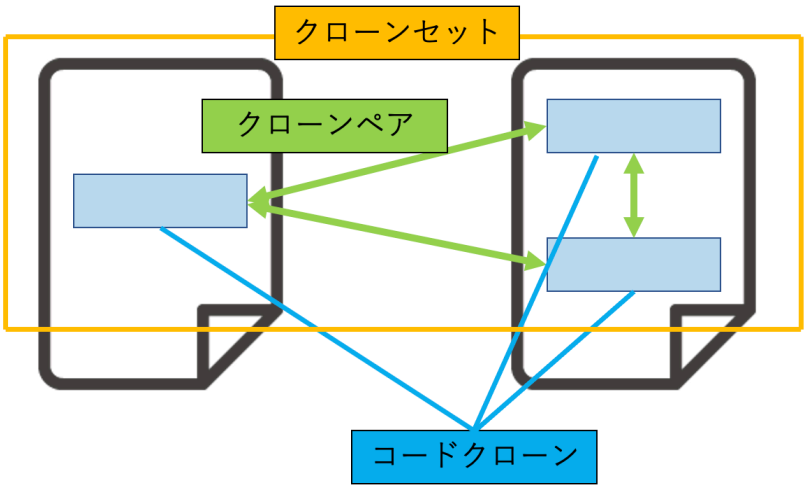

図 1: コードクローン

2.1.1　コードクローンの分類

コードクローンには普遍的定義が存在しない [8]．全てのクローン検出技術とツールについて、統一された概念的枠組みの中で、包括的な定性的比較と評価を提供するために，Roy らはコードクローンの定義として，それぞれのコードクローン間の違いに基づき，4 つのタイプに分類した [7]．

タイプ 1

空白，改行，コメントなどの違いを除いて一致するコードクローン．

タイプ 2

タイプ 1 に加えて識別子，リテラル，型の違いを除いて一致するコードクローン．

タイプ 3

タイプ 2 に加えて，文の変更・挿入・削除などの違いを除いて一致するコードクローン．

タイプ 4

構造上の実装は異なるが，同様の処理を行うコードクローン．

これら 4 つのタイプに分類することで，Roy らは，クローン検出技術と検出ツールを分類するためのスキーマの定義，およびこのスキーマに基づく検出ツールの分類を行った．

2.1.2 コードクローン検出ツール

全てのコードクローンを目視で探し出すことは非現実的であるため，コードクローンを自動で検出することを目的とした様々な手法が，既存研究において提案されている．以下に，代表的な検出手法とその検出手法を用いて実装した検出ツールを示す．

テキストを用いた検出

テキストを用いた検出では，ソースコード文字列を直接比較し，類似度が閾値以上のものをコードクローンとして検出する．これらの比較・検出は，事前に改行や空白などを取り除いた上で行われる．テキストベースの検出手法を採用しているツールとして NiCad[6] が存在する.NiCad は識別子の無視やソースコードの変形をオプションで行うことができ，タイプ 3 までのコードクローンを検出できる．

トークンを用いた検出

トークンを用いた検出では，ソースコード文字列を字句解析でトークン列に変換した後，トークン列における類似度が閾値以上であるものをコードクローンとして検出する．字句解析を行うため，テキストベースの検出のようにコーディングスタイルに依存しないが，文法規則を無視したコードクローンを検出してしまう可能性がある．トークンベースの検出手法を採用しているツールとして CCFinder[9] が存在する．CCFinder は変数名や関数名などの識別子を 1 つの共通のトークンに置換することで，タイプ 2 までのコードクローンを検出できる．

抽象構文木を用いた検出

抽象構文木を用いた検出では，ソースコード文字列を構文解析で抽象構文木に変換した後，部分木における類似度が閾値以上であるものをコードクローンとして検出する．構文解析を行うため，文法規則に従ったコードクローンが検出されるが，抽象構文木

の構築が必要なため，コストは比較的高くなる．抽象構文木ベースの検出手法を採用しているツールとして Deckard[4] が存在する．Deckard は数行の削除や入れ替えなどの違いを持つ部分木同士に対して類似度を高く計算するため，タイプ 3 までのコードクローンを検出できる．

ベクトルを用いた検出

ベクトルを用いた検出では，ソースコード文字列をベクトル表現に変換した後，ベクトル空間での距離が閾値以上であるものをコードクローンとして検出する．ベクトル表現を利用するため，構文的に類似しないコードクローンも検出することができる．ベクトルベースの検出手法を採用しているツールとして CCVolti[12] が存在する．CCVolti は TF-IDF[11] を利用した重み付けを用いて各コードブロックをベクトル表現に変換し，LSH[1] を利用してクラスタリングを行うことでコードクローンを検出する．構文に依存せずコードクローンを検出できるため，タイプ 4 までのコードクローンを検出できる．

メトリクスを用いた検出

メトリクスを用いた検出では，プログラムのファイルや，クラス，メソッドなどのモジュール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で，モジュール単位でのコードクローンを検出することができる．Mayrand らは関数に対して 21 種類のメトリクスを計測することによってコードクローンを検出する手法を提案している [2]．この手法では，タイプ 3 までのコードクローンを検出できる．

2.1.3 コードクローン検出結果を利用した保守作業

ソフトウェア開発者はコードクローン検出結果を分析し，コードクローンに対して保守作業を行う．主な保守作業として，以下の 2 つが挙げられる．

集約

集約とは，クローンセット中の同一の処理を行うコード片に対して，その処理と同様の処理を実行するサブルーチンを作成し，各コード片をそのサブルーチンの呼び出し処理に置き換えることである [16]．これにより，外部的な振る舞いは変更せず，ソースコード量とクローン数を減少する．

同時修正

同時修正とは，クローンセットを一貫して編集することである [17]．例えば，クローンセット中の 1 つのコード片に欠陥が存在し，そのコード片を修正する場合，そのクローンセット中の他のコード片にも一貫した修正を行うかどうか検討する必要がある．

2.2 既存ツール CCX

一般的にコードクローン検出ツールはコードクローンをファイル名と行番号などの組み合わせで出力するため，検出結果を直感的に理解することは難しい．そのため，検出結果を可視化し，効率的なコードクローンの分析を支援する環境が用意されている．しかし，2.1.2 章で説明したように，様々な検出手法を採用したコードクローン検出ツールが多数存在するため，同一のソースコードに対しても，異なる検出結果が得られることが明らかになっている．Wang らは，6 つのコードクローン検出ツールと 8 つのソフトウェアを用いてコードクローン検出を行い，それらの結果を比較することで，同一のソースコードにおいて 6 つ全ての検出ツールで検出されるコードクローンは全体の 10%程度であることを示した．また，6 つの検出ツールの内，どれか 1 つの検出ツールだけで検出されたコードクローンが全体のおよそ 30%から 60%を占めることも示した [10]．これらの結果を踏まえ，松島らは同一ソースコード集合に対し，異なる 2 種類の検出ツールの検出結果を比較するクローンペアマッピングを行い，その結果を分析することで，異なる検出ツールの検出結果を比較する重要性を示した [13]．

しかし，複数の検出ツールを同時に使用することは，検出パラメータの違いや入出力の違いから困難なことであった．そこで，複数の検出ツールを同時に使用することができ，それらの検出結果の可視化や比較を行える，効率的なコードクローンの分析を支援する環境として，松島らは CCX を提案した [14]．CCX は複数の検出ツールを Web 上で使用することができる Web アプリケーションであり，同一のソースコードに対して，異なる 2 種類の検出ツールの検出結果を比較し，散布図を表示する機能を有する．

本章では，はじめに CCX の主な機能について説明した後，それらの比較機能を利用したコードクローン分析を具体例を用いて紹介する．最後に，比較機能で用いる散布図表示の問題点について述べる．

2.2.1 概要

CCX は，図 2 に示すような，松島らによって提案されたコードクローン検出結果を分析する Web アプリケーションである [14]．ユーザーは複数のコードクローン検出ツールを，環境構築無しに Web 上で使用することができ，その検出結果も CCX 上で分析することができる．現在は CCFinderSW[18], CCVolti[12], Deckard[4], NiCad[6] の計 4 種類の検出ツールを使用することができる．

CCX には同一のソースコードに対して，異なる 2 種類の検出ツールの検出結果を比較し，その結果を表示する機能を備えている．検出結果の比較には，松島らによって提案された，*ok* 値と *good* 値を基に行うクローンペアマッピング [13] が利用されている．2 つのクローン

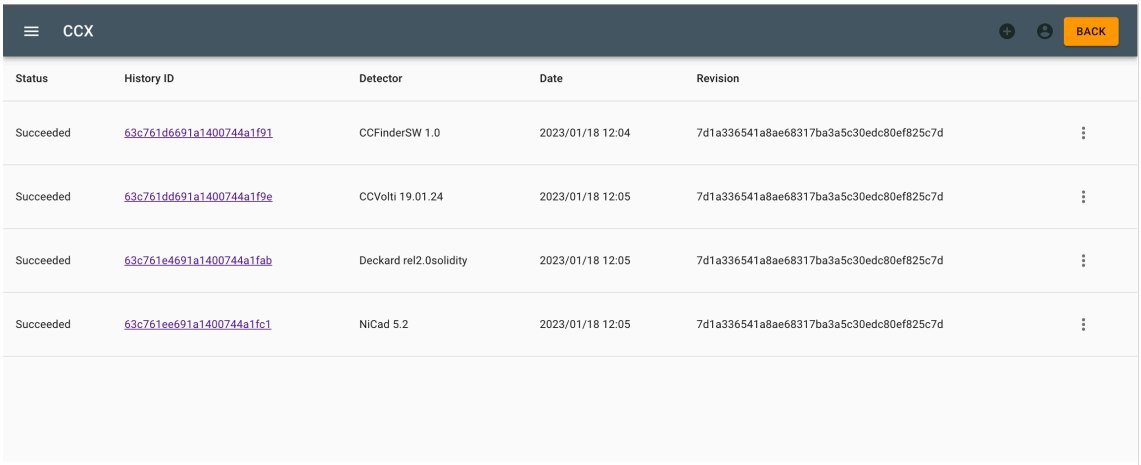

図 2: CCX の画面の例

ペア (a_1, a_2) と (b_1, b_2) に対して，それらの手順を簡単にまとめたものを以下に示す．ここで，a_1, a_2, b_1, b_2 はそれぞれコードクローンであり，a_1, b_1 は a_2, b_2 より検出対象のソースコード内で先に登場するコードとする．

1. a_1 から b_1，a_2 から b_2 へのマッピングを *ok* 値と *good* 値を基に行い，(a_1, a_2) から (b_1, b_2) へのマッピングを行う

2. b_1 から a_1，b_2 から a_2 へのマッピングを *ok* 値と *good* 値を基に行い，(b_1, b_2) から (a_1, a_2) へのマッピングを行う

3. 手順 1 と 2 で得られたクローンペア同士の対応関係の和集合を求めることで，一致しているクローンペアを求める

このように，クローンペアのそれぞれのコードクローンを比較することで，クローンペアマッピングを実現している．

これらの比較結果を可視化し，表示される散布図の例を図 3 に示す．散布図の縦横には，コードクローンが検出されたファイルが並んでおり，各マス目はそれらのファイルの組み合わせにクローンペアが存在するかどうかを表現している．無色点のマスは，それらのファイルの組み合わせにおいてクローンペアが存在しないことを表しており，有色点のマスはクローンペアが存在することを表している．また，有色点には違いがあり，それぞれ基準とした検出結果と比較する検出結果の差異の大きさを表している．赤 (差異 100%) から黄色 (差異 1%) までの階調色，または緑 (差異 0%) で表現している．各マス目を選択することにより，選択したマス目のファイルのファイルパスを上部に表示する．有色点を選択した場合，図 3 で示しているように，COMPARE FILES ボタンが選択できるようになる．このボタンを選

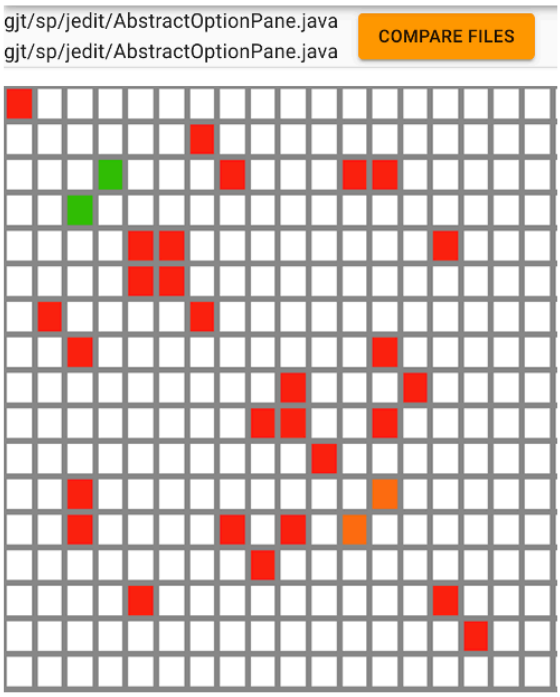

図 3: 表示される散布図の例

択すると，選択したファイルの組み合わせで検出されたクローンペアを図 4 のような Code View で確認することができる．この Code View は 3 つのペインに分割されており，各ペインの役割は表 1 のようになっている．

表 1: Code View の各ペインの役割

ペイン a	選択したファイルの組み合わせに存在するクローンペアのリスト
ペイン b	縦軸に対応するファイルのソースコード
ペイン c	横軸に対応するファイルのソースコード

この比較機能により，1 種類のみの検出ツールの検出結果を利用したソフトウェア保守作業よりも，重要なコードクローンを見逃すことなく集約・同時修正が行える環境を用意している．これらの機能を利用して行うことができるコードの同時修正作業の手順を，具体的に示す．

1. 検出ツール A と検出ツール B のコードクローン検出結果を比較し，図 3 のような散布図を表示する
2. 左上のマスから右に 1 マスずつクリックしていき，表示されるファイルパスを確認することで，修正を行うコード片が含まれるファイルのマスを探す

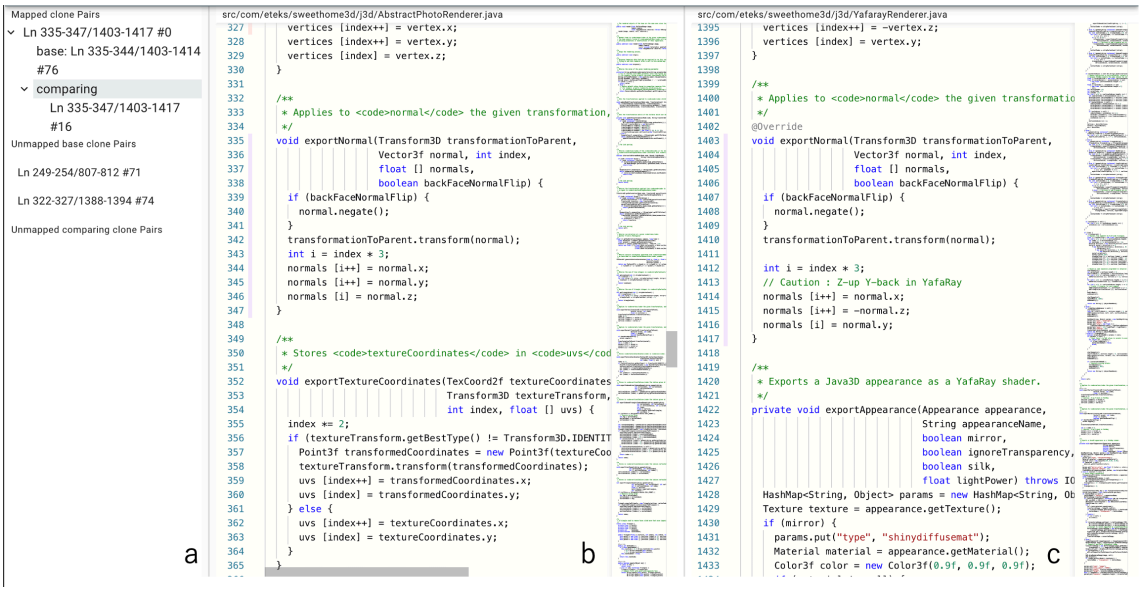

図 4: 散布図表示のための Code View

3. 対象のファイルのマスを発見したら，そのファイルを表している縦の行で有色点のマスを上から順に選択し，図 4 の Code View を表示する

4. 対象のコード片を含むクローンペアを，図 4 のペイン a を 1 つずつ見て確認する (あれば 5. へ，なければ 6. へ)

5. そのクローンペアを全て選択して同時修正が必要かどうかを，ペイン b,c で実際のコードを確認することで検討する

6. 散布図のページへ戻り，他に有色点のマスが存在するならそれを選択し，Code View を表示する (マスが存在するなら 4. へ，存在しないなら終了)

2.2.2 CCX を用いた異なる検出ツール結果の比較

松島らが行った実験の具体例を挙げる [13]．この実験では，Java で記述されたオープンソースソフトウェア jEdit に対して，CCFinderX と NiCad を用いて，それぞれデフォルトパラメータでコードクローンを検出した．その後，クローンペアマッピングを用いてそれらの検出結果を比較・表示し，実際に修正されたバグを含んだコード片のコードクローンを発見した．

実験に用いられた jEdit の詳細を表 2 に示す．また，対象となったコード片をソースコード 1 に示す．このコード片はファイル PrinterDialog.java に含まれていた．以降このコード片をコード片 A と表記する．

表 2: jEdit の詳細

プログラミング言語	リビジョン	LOC
Java	r24577	113,826 行

ソースコード 1: バグを含んでいる jEdit のコード片

```
String checkMarginsMessage = pageSetupPanel.recalculate();
if ( checkMarginsMessage != null )
{
    JOptionPane.showMessageDialog(
                :
                :(略)
                :
PageRanges pr = ( PageRanges )PrinterDialog.this.attributes.get(
    PageRanges.class );
try
{
    pr = mergeRanges( pr );
    PrinterDialog.this.attributes.add( pr );
}
catch ( PrintException e )
{
    e.printStackTrace();
    JOptionPane.showMessageDialog( PrinterDialog.this, jEdit.getProperty("
        print-error.message", new String[]{e.getMessage()} ), jEdit.
        getProperty( "print-error.title" ), JOptionPane.ERROR_MESSAGE );
    return;
}
```

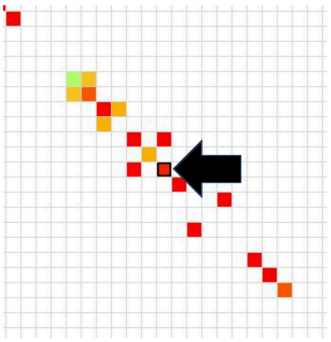

図 5: CCFinderX の検出結果を基準とした散布図の一部

まず，松島らは CCFinderX の検出結果を基準とした比較結果の散布図を分析した．実際に表示された散布図の一部を図 5 に示す．2.2.1 章で説明した同時修正作業を行った結果，コー

ド片 A を含んだクローンペアは，ファイル (PrinterDialog.java, PrinterDialog.java) の組み合わせ中でのみ検出されたことが分かった．図 5 の中央黒枠で囲まれた部分が点 (PrinterDialog.java, PrinterDialog.java) である．この点にはクローンペアが 20 個存在しており，そのうち NiCad で検出されたクローンペアと一致したものは 2 個であった．また，コード片 A を含むクローンペアは CCFinderX で検出された 1 個のみであり，NiCad では検出されていなかった．

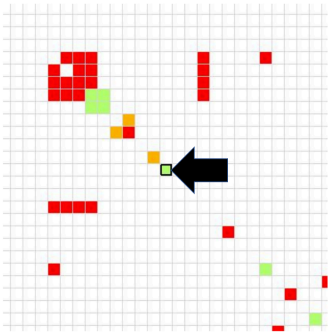

図 6: NiCad の検出結果を基準とした散布図の一部

次に，松島らは NiCad の検出結果を基準とした比較結果の散布図を分析した．実際に表示された散布図の一部を図 6 に示す．中央黒枠で囲まれた部分が点 (PrinterDialog.java, PrinterDialog.java) である．この点は緑で表示されていることから，NiCad で検出されたクローンペアは全て CCFinderX でも検出されたことが分かる．

これらの実験から，バグを含んだコード片 A は NiCad のデフォルトパラメータでは検出できないことが分かった．このことから松島らは，同一のソースコード集合に対して，異なる検出結果の比較を行うことで，重要なコードクローンを見逃すことなく正確な分析を行えることを明らかにした．

2.2.3 問題点

コードの同時修正作業時には，修正が必要なコードクローンとその全てのクローンペアを確認する必要がある．2.2.2 章で述べたように，異なる検出ツールの比較を行うことでより正確な分析をすることができるが，比較した結果を分析するため，1 種類のみの検出結果を分析するよりも多くの時間とコストが必要になる．本章では，2.2.1 章で紹介した手順におけるコードの同時修正作業を行う際の問題点と，散布図表示では理解できない点について述べる．

問題点 1: 特定のファイルを探す際の問題

発見したバグのコードクローンを完璧に取り除くためには，まずは 2.2.1 章で紹介した手順 2 を行い，対象のバグが存在しているファイルを見つける必要がある．しかし，表示される散布図は図 3 であり，左から 1 マスずつ選択してパスを確認することでしか，ファイルを見つけることができない．

また，この散布図上での縦横のファイルの並びは，検出結果におけるファイルの順番となっているため，パス順に並んでいるとは限らず，検出ツールに依存する．そのため，該当するファイルのパスが表示されるまで，マスを左から順に選択し続ける必要があり，コードクローンが検出されたファイルの個数を n とすると，最悪で n 回の試行を必要とする．ソフトウェアの規模が大きく，コードクローンが検出されるファイル数が増加すればするほど，多くのマウスクリックと，パスの確認作業が必要となってしまう．

問題点 2: ファイル単位での比較結果に着目する際の問題

比較結果の散布図の 1 マスはファイルの組み合わせごとである．そのため，1 つのファイルに着目してコードクローンを確認しようとした場合，2.2.1 章で述べた手順 2 で特定のファイルを探した後，手順 3 から 6 を繰り返す必要がある．図 7 に示すように，対象のファイルの縦の行の，全ての有色点を選択する必要があり，これらの 1 マスごとに図 8 に示すような Code View のペイン a を確認する必要がある．

この手順 3 から 6 の繰り返し回数は，図 7 の有色点のマス目の数に依存し，コードクローンが検出されたファイルの個数を n とすると，最悪の場合 n 回となる．この時，1 つのマスを確認する度に，Code View から散布図表示に戻る必要があり，多くの画面遷移が必要となる．既存ツールの実装方法では，散布図を表示する処理を行うたびに検出結果の比較処理を行うようになっているので，手順 3 から 6 の繰り返し回数が多いほど比較に要する時間も増加してしまう．これらの理由から，ファイル単位での比較結果に着目する場合，クローンペアが他のファイルに広がっていればいる程，多くの時間とコストが必要となってしまう．

問題点 3: 比較後のコードクローン検出数に着目する際の問題

表示される散布図は比較結果の差異 (割合) を閾値としている．そのため，異なるファイル間での，コードクローン検出量の違いはわからなくなっている．例えば，ファイル (A,A) の組み合わせにおいて，検出結果 1 では 1 つのクローンペアが，検出結果 2 でも 1 つのクローンペアが検出され，ファイル (B,B) の組み合わせでは，検出結果 1 で 50，検出結果 2 でも 50 のクローンペアが検出されていた場合を考える．検出結果を比較した結果は，全て一致しなかったものとすると，ファイル (A,A) とファイル (B,B) のマスは，共に図 3 中に存在する

図 7: ファイル特定後に選択する必要があるマス目の例

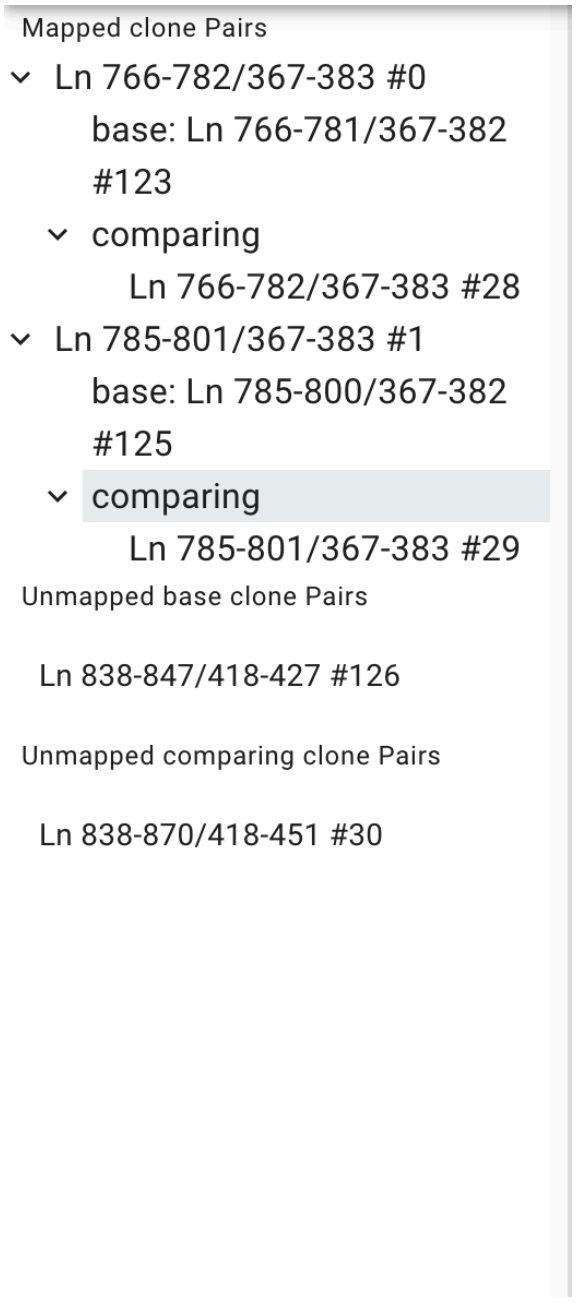

図 8: クローンペアリストが確認できるペイン a

赤色で表示される．しかし，実際に修正を検討するコードクローンの量は，ファイル (A,A) は 2 つであるのに対し，ファイル (B,B) は 100 もあることから，ファイル A よりも，ファイル B の方が修正の検討箇所は多い．このような異なるファイル組み合わせ間での，検出されたコードクローンの個数の差異は散布図では理解することはできない．

3　既存ツールの拡張に向けた調査

本章では，2.2.3 章で述べた問題点を解決するためのキーアイデアと，そのアイデアの正当性を示すために行った調査について述べる．

まず，2.2.3 章で述べた問題点を解決するためのキーアイデアを以下に示す．

ファイルツリーの表示

2.2.3 章で述べた問題点 1 である,「特定のファイルを探す際の問題」を解決するためのアイデアである．これにより，特定のファイルの探索を，複数回のマス目の選択で闇雲に探していた既存ツールよりも，明らかに早く，手間のかからない作業にすることができる．

クローンセットの表示

2.2.3 章で述べた問題点 2 である,「ファイル単位での比較結果に着目する際の問題」を解決するためのアイデアである．全てのクローンペアを確認するために複数回の画面遷移が必要だった既存ツールと異なり，あるファイルに存在するコードクローンのクローンセットを 1 つのビューに表示すれば，同時修正を検討する必要があるコードクローンとその全てのクローンペアを画面遷移なしに確認することができる．

ファイルごとのコードクローン数の表示

2.2.3 章で述べた問題点 3 である,「比較後のコードクローン検出数に着目する際の問題」を解決するためのアイデアである．異なる 2 種類の検出ツールで検出されたコードクローンの合計数をファイル単位で表示することで，ファイルごとの合計コードクローン検出数の違いが理解できる．

上記に示したキーアイデアの内，ファイルごとのコードクローン数を表示し，それらの違いを理解することに意味があるのかどうかは疑問が残った．そのため，コードの同時修正作業を検討する必要があるようなコード片と，比較後のファイル単位での合計コードクローン検出数との関係を調査した．はじめに，2.2.2 章で述べた既存研究の例を題材にして行った調査結果を示し，その後，その他の revision やソフトウェアを題材とした調査結果を示す．

3.1　概要

コードの同時修正作業を検討する必要があるようなコード片と，比較後のファイル単位での合計コードクローン検出数との関係を調べるため，調査を行った．ここでは 2.2.2 章で述べたような，特定の検出ツールでしか検出できないコードクローンでバグを含んでいるものを対象に調査する．

調査の対象となったソフトウェアは，jEdit[5], Logisim[6], FreeMind[7]の 3 つの Java プログラムである．使用した検出ツールは，CCFinderSW[18], CCVolti[12], Deckard[4], NiCad[6] の 4 種類である．また，バグを含んでいるコード片とは，revision 間で何らかの修正が行われたコード片のことを指す．

3.2 調査手順

調査の手順は以下のとおりである．

1. ある revision のソースコードを，4 種類の検出ツールを使用してコードクローン検出をする
2. それぞれの検出結果を比較し，比較後のファイル単位ごとの検出コードクローン数を計算する
3. 手順 1 の revision で修正されたコード片を含んでいるコードクローンが検出されているかを確認する
4. 検出されていれば，そのファイル中に存在しているコードクローン数 (比較した合計値) を確認し，その値が全体の何番目であったかどうかも確認する

上記の調査を，各ソフトウェアそれぞれ 10 個の revision について調査する．調査対象の詳細を表 3 に示す．これにより，比較後のファイル単位での合計コードクローン検出数と，実際に修正されたコード片との関係を探った．

表 3: 各ソフトウェアの詳細

ソフトウェア名	10 個の revision 変化が行われた期間	プログラミング言語
jEdit	2016/11/29 - 2017/01/05	Java
Logisim	2011/03/15 - 2011/03/21	Java
FreeMind	2013/10/16 - 2013/12/08	Java

3.3 調査結果

まず，始めに 2.2.2 章で述べた既存研究の題材を使用してこの調査を行った調査結果を表 4 に示す．ソースコード 1 のコード片をコードクローンとして検出できたのは CCFinderSW のみであった．

[5] http://www.jedit.org/?page=features
[6] https://sourceforge.net/projects/circuit/
[7] https://sourceforge.net/projects/freemind/

表 4: 既存研究の題材を使用した調査結果

比較の組み合わせ	ファイル (PrinterDialog.java) 内で検出されたコードクローン合計数	何番目か/検出ファイル数
CCFinderSW/CCVolti	54	4/283
CCFinderSW/Deckard	52	4/275
CCFinderSW/NiCad	54	4/286

この調査結果を見てみると，コードクローン合計数が 4 番目に大きいことが分かる．また，このような形で 3 つのソフトウェアの 10 個の revision に対して本調査を行った結果を表 5 に示す．ただし，表 5 内の，対象のコード片の個数とは，実際に revision 間で修正されたコード片のコードクローンの内，特定の検出ツールでしか検出されなかったコード片の合計数を記載している．また，コードクローン合計数は，対象のコード片が含まれているファイル内での比較後の合計コードクローン検出数である．この値と何番目であるかの値は，それぞれのソフトウェアごとの平均値 (小数点第 2 以下切り捨て) である．

表 5: 10 個の revision に対して行った調査結果

ソフトウェア名	対象のコード片の個数	コードクローン合計数	何番目か/検出ファイル数
jEdit	30 個	61.5	20.3/170.0
Logisim	41 個	11.4	73.1/285.6
FreeMind	4 個	7.0	101.5/291.5

3.4 考察

表 4 と表 5 の結果から，revision 間で修正されたコード片のコードクローンが含まれるファイル内でのコードクローン検出数が，全体の上位何%であったかを計算すると，既存研究を題材にした調査では，上位 1.4%であり，3 つのソフトウェアでは順に，11.9%,25.5%,34.8%であった．この結果から，バグを含み修正されるようなコード片のコードクローンの内，特定の検出ツールでしか検出されないコードクローンは，比較後のファイル単位でのコードクローン検出数が大きいファイルに存在している可能性が高いことが，本調査から分かった．このことから，2.2.3 章で述べた，比較後のファイル単位でのコードクローン検出数が分からないという問題点 3 は解決する必要があり，ファイル間でのコードクローン検出数の違いが理解できる表示をする必要があると考えた．

4　拡張した比較表示ツール

本報告では，3 章で述べたキーアイデアを基に，コードクローン検出結果を比較した内容を表示する複合グラフと，それに対応した CloneSet View を開発した．表示するデータは，クローンペアマッピング [13] の手順 1,2 で求めていた，マッピングされたコードクローンを利用した．本章では，本ツールの詳しい機能を説明する．

4.1　複合グラフ

本報告で開発した，ファイル単位でのコードクローン検出数の違いが理解できる積み上げ棒グラフと折れ線グラフの組み合わせ表示を，複合グラフと呼ぶ．この複合グラフは，2 つのコードクローン検出結果を入力として与え，コードクローンの比較を行うことで表示される．このグラフの横軸はファイルが並んでおり，各グラフの値はファイル単位の値である．積み上げ棒グラフの縦軸は，検出結果を比較した結果，各ファイル中で検出されたコードクローンの個数である．積み上げ棒グラフは，赤・青・紫の 3 つの棒で構成され，紫は一致したコードクローンの個数，赤は 1 つ目の検出結果で一致しなかったコードクローンの個数，青は 2 つ目の検出結果で一致しなかったコードクローンの個数である．また，積み上げ棒グラフの縦軸の最大値は，各ファイル中のコードクローン検出数の平均値に 30 を加えた値に設定される．この 30 という値は，様々な値を設定した結果，縦軸 1 の積み上げ棒グラフがはっきりと認識できる値であったためである．そのため，最大値で見切れている積み上げ棒グラフの値は最大値以上であり，実際の値はマウスホバーで確認することができる．また，折れ線グラフの縦軸は 1 つ目と 2 つ目の検出結果の一致率である．これらの複合グラフは比較を行った結果，1 つ目と 2 つ目の検出結果の一致率が低い順にソートされ，同じ一致率であれば積み上げ棒グラフの値が大きいファイルから順に並べられている．一致率を基準にソートしているのは，ファイルサイズが小さいことが原因で，コードクローン検出数が少なくなってしまうファイルを見逃さないようにするためである．この表示により，比較を行った結果，考慮するコードクローン数が増加する可能性の高いファイルから順に注目することができるようになる．

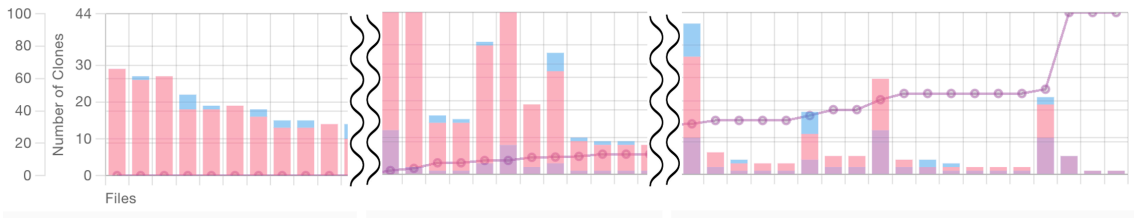

図 9: 表示される複合グラフの例

ある java のソースコードに対して，CCFinderSW[18] と NiCad[6] のコードクローン検出結果を比較した際，表示される複合グラフを一部抜粋したものを図 9 に示す．同じ一致率の範囲では積み上げ棒グラフの値が大きい順にソートされていることが分かる．また，折れ線グラフの値が右上がりなことから，一致率が低いファイルから高いファイルの順に表示されていることが分かる．

この複合グラフにより，ファイル単位でのコードクローン検出数が可視化され，それらの違いが理解できるようになり，問題点 3 を解決した．これにより，コードクローン検出数が多いファイルを発見し，そのファイルの修正を検討することが容易になった．

4.2 CloneSet View

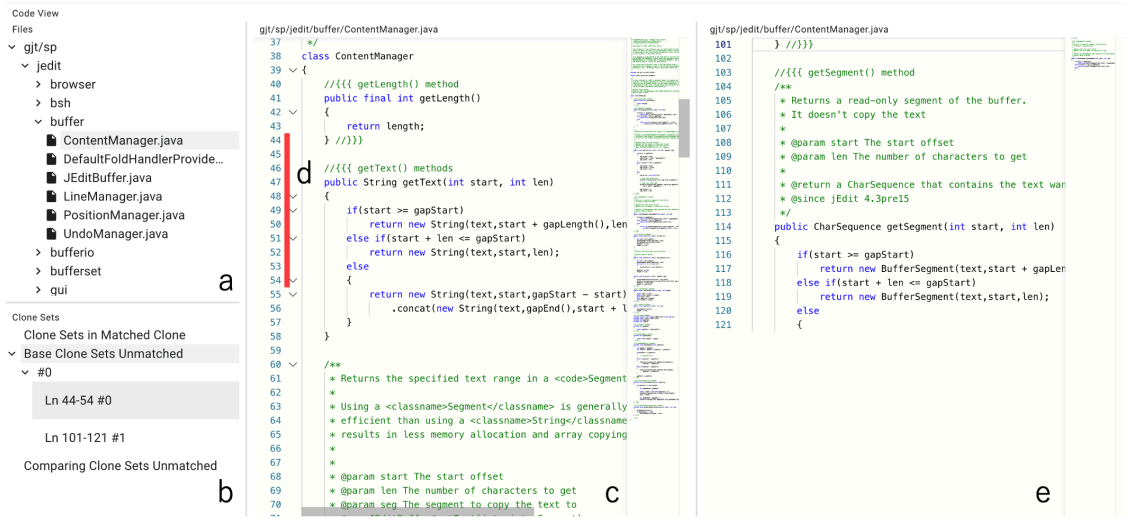

図 10: 表示される CloneSet View の例

図 9 と同時に表示される，CloneSet View を図 10 に示す．この CloneSet View は，1 つ目と 2 つ目の検出ツールで検出されたコードクローンのクローンセットを，ファイル単位で確認することができる View である．この CloneSet View は 4 つのペインで構成されている．各ペインの役割を表 6 に示す．また，各ペイン間の関係を図 11 に示す．

ペイン a はファイルツリーである (図 12)．このファイルツリーにより，散布図では難しかった特定のファイルを探す問題 (問題点 1) を解決した．

ペイン b はクローンリストである (図 13)．図 13 に示すように，一致するコードクローンが存在するクローンセット，1 つ目の検出結果で検出された一致するコードクローンがなかったクローンセット，2 つ目の検出結果で検出された一致するコードクローンがなかったクローンセット，の順に上から表示される．一致するコードクローンが存在するクローンセッ

表 6: CloneSet View の各ペインの役割

ペイン	名称	役割
a	ファイルツリー	2 つの検出結果でコードクローンが検出された ファイルをパス順に表示
b	クローンリスト	選択されたファイル中で検出された コードクローンのクローンセットを表示
c	ソースコードビュー	選択されたファイルのソースコードを表示
d	クローンペアビュー	ペイン c でハイライトされている d のコードクローンの クローンペアとそのファイルパスを表示

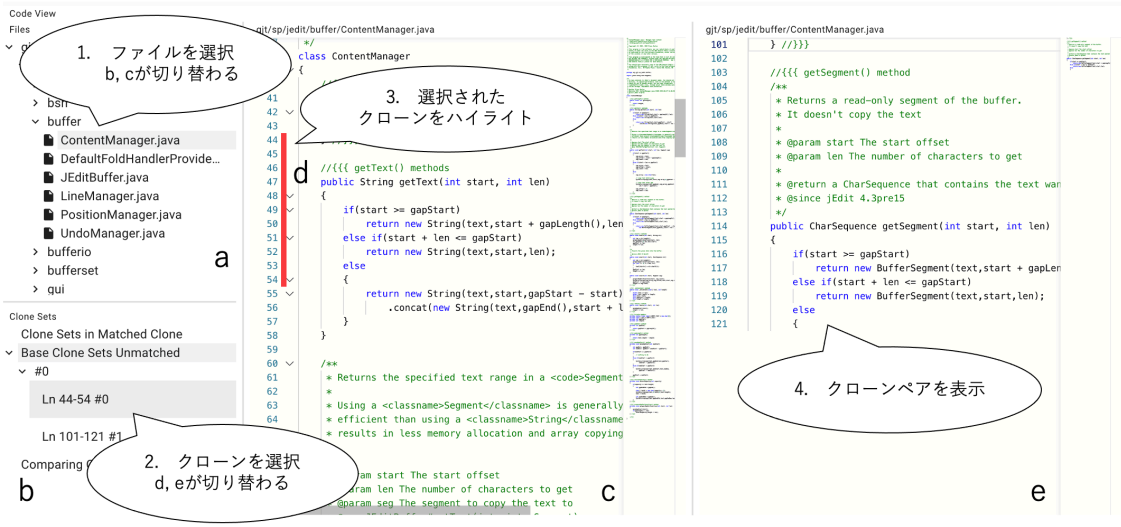

図 11: 各ペイン間の関係

トは，クローンセット中の全てのコードクローンが一致しているのではなく，少なくとも 1 つ一致するコードクローンが存在しているクローンセットである．各クローンセットのタブはトグルになっており，展開することで，実際に検出されたコードクローンを開始行と終了行の組み合わせで確認できる．このように，クローンセットを表示することにより，選択したファイル中で検出されたコードクローンの全てのクローンペアを 1 度に確認することができ，散布図と Code View の行き来が必要だった問題点 2 を解決した．

ペイン c はソースコードビューである．ハイライトされている d は，クローンリストで選択したコードクローンのいずれかである．各クローンセットの#0 のクローンを選択した場合は，そのクローンがハイライトされる．それ以外のクローンを選択した場合は，そのクローンが現在表示しているソースコード中のコードクローンであったなら，そのクローンを

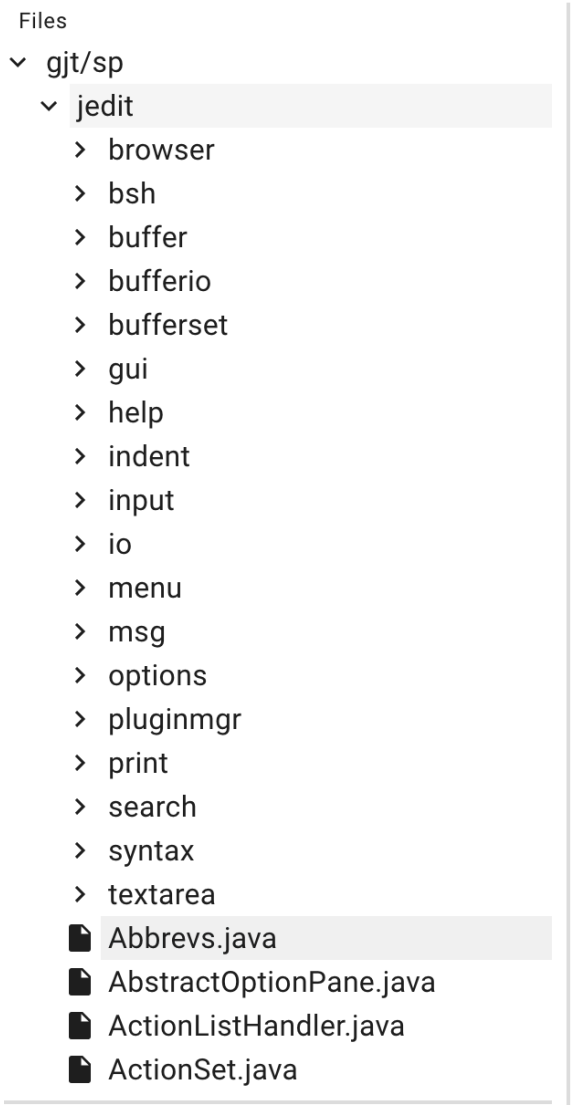
Files
gjt/sp
jedit
browser
bsh
buffer
bufferio
bufferset
gui
help
indent
input
io
menu
msg
options
pluginmgr
print
search
syntax
textarea
Abbrevs.java
AbstractOptionPane.java
ActionListHandler.java
ActionSet.java

図 12: ファイルツリー

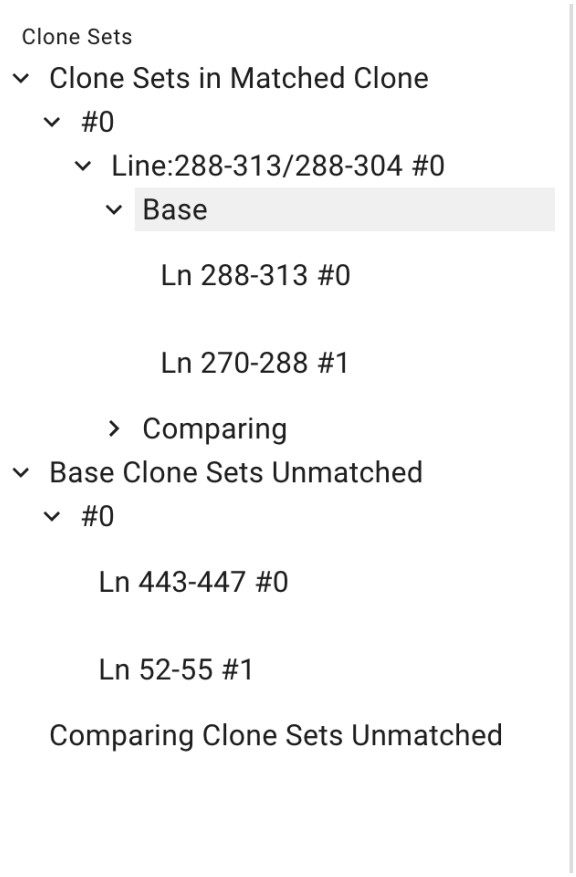
Clone Sets
Clone Sets in Matched Clone
#0
Line:288-313/288-304 #0
Base
Ln 288-313 #0
Ln 270-288 #1
Comparing
Base Clone Sets Unmatched
#0
Ln 443-447 #0
Ln 52-55 #1
Comparing Clone Sets Unmatched

図 13: クローンリスト

ハイライトする．そうでない場合は，#0 のコードクローンをハイライトする．ペイン c により，バグを含み修正対象となったソースコードを，コードクローンをハイライトしながら確認することができる．

ペイン e は，クローンペアビューを表示する．上部にそのクローンペアが存在するファイルのパスを表示し，ペイン内にはクローンリストで選択したコードクローン，または#0 のコードクローンの開始行から終了行までのソースコードが表示される．ペイン e により，修正を検討するクローンペアを順に確認することができる．

4.3 表示画面

図 14: 表示画面

4.1 章，4.2 章でそれぞれ説明した機能を 1 つの画面に表示したものを図 14 に示す．本報告で作成したツールでは，比較機能を使用すると，この画面が表示される．画面上部に複合グラフが，画面下部に CloneSet View を表示している．検出されたコードクローン数が多いファイルにおいて，ソフトウェア保守作業を行う場合は，複合グラフでファイル名を確認し，そのファイルをファイルツリーで選択することで，コードクローンの修正を検討できる．また，何らかのバグ箇所を修正する際に，そのコードクローンと全てのクローンペアの修正を検討する場合には，画面下部の CloneSet View で修正の検討ができる．

5　評価実験

本章では，4 つのソフトウェア Sweet Home 3D[8], jEdit[9], Process Hacker[10]および snns[11] に対して，2 つのコードクローン検出ツール CCFinderSW[18], NiCad[6] でコードクローン検出を行い，それらの検出結果を比較する評価実験について述べる．

5.1　実験内容

本評価実験では，異なる 2 種類の検出ツールの検出結果を比較した結果を利用してソフトウェア保守作業を行う際に，いかに効率よく行うことができるかを評価する．評価対象は，2.2.1 章で紹介した散布図を利用した既存ツールと，4 章で紹介した本報告で追加した新規ツールである．

何らかのコード片を修正する際，そのコード片を含むコードクローンが検出されているならば，その全てのクローンペアに対して，修正を検討する必要がある．このような場合を想定し，この作業を行う際の，画面遷移の回数，マウスクリックの回数，検討するコードクローンの個数と行数を測定し，これらの値が小さいほど効率よく修正を検討できているものとする．ただし，画面遷移の回数に関しては，一部のみの画面遷移と全体の画面遷移を区別して考える．また，修正を検討するコードクローンの元となるコード片は，実際に revision 間で修正が行われているコード片を対象とした．実験で使用した 4 つのソフトウェアの詳細を表 7 に，対象となったコード片が含まれているファイルとその行を表 8 に示す．

表 7: ソフトウェアの詳細

ソフトウェア名	revision	ファイル数	LOC	言語
Sweet Home 3D	r8383	260	115943	Java
jEdit	r24577	558	113826	Java
Process Hacker	r6322	248	167863	C
snns	714d60	149	79512	C

5.2　実験手順

本章では，まず，何らかのコード片を修正する際に行うコードクローン分析作業において，必要な手順について述べる．その後，それらの手順を達成するための具体的な実験手順を，

[8] http://www.sweethome3d.com/
[9] http://www.jedit.org/?page=features
[10] https://processhacker.sourceforge.io/
[11] http://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

表 8: 修正されたコード片が含まれているファイルとその行

ソフトウェア名	ファイル名	行
Sweet Home 3D	YafarayRender.java	182-215
jEdit	PrinterDialog.java	260-293
Process Hacker	updater.c	234-256
snns	ui_config.c	126-132

既存ツールと新規ツールを使用した場合で別々に述べる．

5.2.1 分析手順

何らかのコード片を修正する際に，コードクローン検出結果を利用する場合，修正するコード片を含んでいるコードクローンのクローンセットに対して，2.1.3 章で説明した集約や同時修正を検討する必要がある．それらの検討作業を実現するための手順は以下の通りである．

1. 修正するコード片を含んでいるコードクローンが検出されているか確認する
2. 検出されている場合，そのコードクローンの全てのクローンペアを確認し，修正を検討する
3. 修正するコード片を含むコードクローンが複数個検出されているならば，それら全てに対して，手順 1,2 を行う

5.2.2 既存ツールを使用する場合

既存ツールを使用した場合の実験手順は 2.2.1 章で記載した，コードの同時修正作業の手順である．手順 1 での検出ツール A は CCFinderSW であり，B は NiCad である．手順 3 の対象のファイルとは，表 8 に示すファイル名のことである．これらの手順が完了するまでの，画面遷移の回数，マウスクリックの回数，確認クローンの個数と行数，を測定する．ただし，手順 2 の散布図上でのファイル探索のためのマウスクリック回数と，手順 3 から 6 のマウスクリック回数は別々に測定する．

2.2.1 章で説明した手順により，修正するコード片が存在しているファイルの全てのクローンペアを確認することができる．そのため，5.2.1 章で説明した分析手順と同様の処理を行っていることが担保でき，既存ツールを使用した実験手順は，正確にコードクローンの分析が行えていると言える．

5.2.3 新規ツールを使用する場合

新規ツールを使用した場合の実験手順は以下のようになる．新規ツールを使用する場合は，複合グラフにおいて，対象のコード片が存在するファイルが左から何番目に表示されているのかも測定する．

1. CCFinderSW と NiCad のコードクローン検出結果を比較し，図 14 の画面を表示する
2. 対象のコード片が存在するファイルが，複合グラフの左から何番目かを測定する
3. バグ箇所が含まれるファイルを図 12 のファイルツリーから探し，選択する
4. 図 13 のクローンセットを上から順に全て展開して，コードクローンを 1 つずつ確認し，対象のコード片を含むクローンセットを見つける
5. 発見したらそのクローンセット中のコードクローンを 1 つずつ確認する

これらの手順が完了するまでの，画面遷移の回数，マウスクリックの回数，確認クローンの個数と行数，を測定する．ただし，マウスクリック回数は，手順 3 のクリック回数と，その他のクリック回数を区別して測定する．

これらの手順により，修正するコード片が存在している全てのクローンセットを確認することができる．そのため，5.2.1 章で説明した分析手順と同様の処理を行っていることが担保でき，新規ツールを使用した実験手順は，正確にコードクローンの分析が行えていると言える．

5.3 実験結果

実験結果をそれぞれのソフトウェアごとに示す．既存ツールのマウスクリック数は，既存ツールの実験手順 2 でのファイル探索のクリック回数とその他の回数を，手順 2 での回数/その他の回数の形で表記している．新規ツールのマウスクリック数は，新規ツールの実験手順 3 でのファイル探索のクリック回数とその他の回数を，手順 3 での回数/その他の回数の形で表記している．

Sweet Home 3D

新規ツールの複合グラフの表示において，ファイル YafarayRender.java は左から 49 番目のファイルだった．その他の詳しい実験結果を表 9 に示す．

jEdit

新規ツールの複合グラフの表示において，ファイル PrinterDialog.java は左から 78 番目のファイルだった．その他の詳しい実験結果を表 10 に示す．

表 9: 実験結果 (Sweet Home 3D)

使用ツール	画面遷移回数 (全体/一部)	マウスクリック数	確認クローン (個数/行数)
既存ツール	23 回/1 回	31 回/66 回	402 個/5190 行
新規ツール	1 回/3 回	2 回/97 回	239 個/3952 行

表 10: 実験結果 (jEdit)

使用ツール	画面遷移回数 (全体/一部)	マウスクリック数	確認クローン (個数/行数)
既存ツール	5 回/1 回	67 回/6 回	68 個/1033 行
新規ツール	1 回/3 回	2 回/30 回	56 個/886 行

Process Hacker

新規ツールの複合グラフの表示において，ファイル updater.c は左から 66 番目のファイルだった．その他の詳しい実験結果を表 11 に示す.

表 11: 実験結果 (Process Hacker)

使用ツール	画面遷移回数 (全体/一部)	マウスクリック数	確認クローン (個数/行数)
既存ツール	27 回/1 回	74 回/42 回	52 個/1239 行
新規ツール	1 回/3 回	3 回/27 回	48 個/1156 行

snns

新規ツールの複合グラフの表示において，ファイル ui_config.c は左から 13 番目のファイルだった．その他の詳しい実験結果を表 12 に示す.

これらの結果を平均した値 (小数第 2 切り捨て) を表 13 に示す．新規ツールの複合グラフの表示において，対象のファイルは平均で左から 51.5 番目のファイルだった.

5.4 考察

5.3 章の実験結果の内，対象ファイルを発見するための散布図のマス目のクリック数と，複合グラフでの左から何番目かの値を比べると，散布図と複合グラフでのファイル探索にはあまり変化がないことが分かった．複合グラフは，ファイル単位でのコードクローン検出数の違いを理解するために表示したものであり，特定のファイル探索をするための表示ではない．本報告で実装したツールで，この役割を担うのは，図 12 に示すファイルツリーである．このファイルツリーにおいて，特定のファイルを発見するために必要なクリック数は，最悪の場合でもファイルパスに含まれるディレクトリの数+1 である．これはたかだか 10 以内に

表 12: 実験結果 (snns)

使用ツール	画面遷移回数 (全体/一部)	マウスクリック数	確認クローン (個数/行数)
既存ツール	3 回/1 回	65 回/4 回	44 個/463 行
新規ツール	1 回/3 回	2 回/13 回	32 個/356 行

表 13: 実験結果 (平均)

使用ツール	画面遷移回数 (全体/一部)	マウスクリック数	確認クローン (個数/行数)
既存ツール	14.5 回/1 回	59.2 回/29.5 回	141.5 個/1981.2 行
新規ツール	1 回/3 回	2.2 回/44 回	86.2 個/1587.5 行

は収まるだろう．実際の値を確認してみると，散布図表示における対象ファイルを発見するための必要平均クリック数が 59.2 回なのに対し，新規ツールにおける対象ファイルを発見するための必要平均クリック回数は 2.2 回であり，明らかに減少している．このことから，ファイルツリーは問題点 1 の解決に貢献していると言える．

また，問題点 3 で指摘したように，散布図表示だけでは，ファイルの組み合わせ中に存在するコードクローンの量は分からなかった．しかし，積み上げ棒グラフの値を見ることで，コードクローン検出数をファイル単位で理解することができるようになっている．このことから，コードクローン検出数の多いファイルを発見し，そのファイルのコードクローンを表示する作業において，複合グラフとファイルツリーの組み合わせは，非常に効果的であると考えられる．

さらに，新規ツールを使用した方が，全体の画面遷移の回数が減少していることが分かる．既存ツールは，散布図と図 4 の Code View を行き来する必要がある．そのため，様々なファイルにクローンペアが広がっていればいるほど全体の画面遷移の回数は多くなり，ファイル数を n としたとき，最悪 n 回である．また，この値は，コードクローンが検出されるファイルが増加するほど大きくなる可能性が高く，対象ソフトウェアに大きく依存する．これに対し新規ツールは，全体の画面遷移の回数は，いかなるソフトウェアの場合でも，図 14 のような全体画面を表示する 1 回だけである．そのため，新規ツールの方が，既存ツールよりも常に全体の画面遷移回数は少なくなり，効率よく修正を検討することができる．

最後に，確認したコードクローンの個数と行数も，新規ツールを使用した方が少なくなっていることが分かる．このことから，クローンペアで表示するよりも，ファイル単位でクローンセットを表示した方が，確認するコードクローンの数が減り，効率よく修正を検討できると言える．

6　まとめと今後の課題

本報告では，異なる 2 種類の検出結果を比較し，その結果を利用したコードの同時修正をするためのツールを開発した．クローンペアを基に表示していた既存ツールとは異なり，本ツールではファイル単位でクローンセットを表示するようにしている．また，本ツールではファイル単位でのコードクローン検出数が理解できる複合グラフも表示する．

ツールの評価実験では，実際に修正されたコード片を含むコードクローンとその全てのクローンペアを確認し，その作業がいかに効率良くできるかを，CCX の既存ツールと比較した．これにより，既存ツールに比べて新規ツールの優位性を確認した．

今後の課題は，比較結果の可視化を行うグラフ上に，より重要な情報を提示することである．本報告では，ファイル単位でのコードクローンの量と，2 つの検出結果の一致率を基に，複合グラフを表示した．しかし，評価実験で示したように，散布図上でのファイルの探索と，グラフ上でのファイルの探索には，優位性は見られなかった．より重要な情報を提示できるようなメトリクスを設定し，それが理解しやすいグラフを表示することができれば，ソフトウェア保守にかかるコストを減らすことができるのではないかと考えている．

謝辞

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 肥後 芳樹 教授にはご多忙の中，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝申し上げます．

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 松下 誠 准教授には，研究の着想から研究活動の直接の御指導，論文の執筆に至るまで，あらゆる場面で多くの御指導を賜りました．松下 誠 准教授の適切な御指導により，本論文を完成させることができました．心より深く感謝申し上げます．

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 神田 哲也 助教には，発表の問題点の提示において，多くの御助言を賜りました．心より深く感謝申し上げます．

最後に，その他様々な御指導，御助言等を賜りました，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肥後研究室の皆様に，心より深く感謝申し上げます．

参考文献

[1] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *In Proceedings of the 13th Annual ACM Symposium on Theory of Computing, STOC 1998*, pp. 604–613, ACM, 1998.

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

[3] F. Khomh L. Barbour and Y. 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.

[4] Z. Su L. Jiang, G. Misherghi and S.Glondu. Deckard: Scalable and accurate tree-based detection of code clones. *In Proceedings of the 29th international conference on Software Engineering, ICSE 2007*, pp. 96–105, IEEE Computer Society, 2007.

[5] C. K. Roy M. Mondal, B. Roy and K. A. Schneider. An empirical study on bug propagation through code cloning. *Journal of Systems and Software*, Vol. 158, p. 110407, 2019.

[6] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *In Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008*, pp. 172–181, IEEE Computer Society, 2008.

[7] Chanchal K. Roy, James R. Cordy, , 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.

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

[9] S. Kusumoto T. Kamiya and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670, 2002.

[10] Yue Jia Tiantian Wang, Mark Harman and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. *In European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pp. 455–465, Aug. 2013.

[11] R. B. Yates and B. R. Neto. Modern information retrieval. *ACM press New York*, Vol. 463, pp. 96–105, 1999.

[12] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく細粒度プロッククローン検出. コンピュータソフトウェア, Vol. 35, No. 4, pp. 16–36, 2018.

[13] 松島一樹, 井上克朗. 複数コードクローン検出結果の比較・表示法. 信学技報, Vol. 119, No. 112, pp. 147–152, July. 2019.

[14] 松島一樹, 小池耀, 井上克朗. Ccx: Saas 型コードクローン分析システム. 日本ソフトウェア科学会, Vol. 39, No. 4, pp. 129–143, Nov. 2022.

[15] 井上克朗, 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. 47–54, 2001.

[16] 吉田則裕肥後芳樹. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, pp. 42–56, 2011.

[17] 横井 一輝 崔 恩瀞 吉田 則裕 井上 克郎本田 紘貴. コードクローン保守支援を目的とした変更履歴可視化システム. 信学技報, Vol. 118, No. 471, pp. 115–120, March. 2019.

[18] 瀬村雄一, 吉田則裕, 井上 克朗崔 恩瀞. 多様なプログラミング言語に対応可能なコードクローン検出ツール ccfindersw. 電子情報通信学会論文誌 D, Vol. J103-D, No. 4, pp. 215–227, Dec. 2019.