異なるコードクローン検出ツールの結果を比較して表示する手法の提案

小林 亮太[1,a)]　松下 誠[1,b)]　肥後 芳樹[1,c)]

概要：
コードクローン検出を行う際，検出手法などの違いから，同一のソースコード集合に対しても得られる結果が異なることが知られている．この問題に対処するため，コードクローン分析ツール CCX が開発された．しかし，ソフトウェア保守時において CCX を利用する場合，CCX の比較機能にはいくつかの問題点が存在した．そこで本研究では，既存ツールの問題点を踏まえ，コードの同時修正作業に着目した拡張のアイデアを提案する．また，それらのアイデアを採用したツールを CCX を拡張する形で実装した．作成したツールの評価実験を行うことで，既存ツールよりも提案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．

キーワード：コードクローン，ソフトウェア保守

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 [4]．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 [6]．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 [1,9]．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開発されている [5,6,10,13]．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 [12]．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．

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ている [7]．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 [8]．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，ソフトウェア保守において，CCX を用いてコードクローン検出結果の分析を行う場合，現状における比較結果の表示方法では以下のような問題点が存在していると考える．

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

これらの既存ツールの問題点を踏まえ，本研究では，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 を拡張し，ツールとして実装した．

作成したツールの評価実験では，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．その結果，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工数が少ないことを確認した．

以降，2 節では本研究の背景として，コードクローンと，既存ツール CCX について述べる．3 節では，拡張案の正当性を示すための調査について述べる．4 節では，提案した手法について述べる．5 節では，実装ツールの詳細について述べる．6 節では評価実験について述べる．最後に 7

1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Suita, 565–0871, Japan
a) ryt-kbys@ist.osaka-u.ac.jp
b) matusita@ist.osaka-u.ac.jp
c) higo@ist.osaka-u.ac.jp

© 1959 Information Processing Society of Japan

節では，まとめと今後の課題について述べる．

2. 背景

本節では，本研究の背景として，コードクローンの定義と，コードクローンを利用したソフトウェア保守作業およびコードクローン分析ツール CCX 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 [4]．コードクローンは，主に既存コードの使い回しや，コード生成ツールによる自動生成によって生じる [6]．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．

2.2 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 [2]．

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 [3]．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．

2.3 コードクローン分析ツール CCX

既存研究において, 松島らは, 複数の検出ツールを Web 上で使用できる Web アプリケーション CCX を提案した [8]．本節では，はじめに CCX の主な機能について説明した後，比較機能で用いる散布図表示の問題点について述べる．

2.3.1 概要

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 上で使用でき，また，その検出結果も CCX 上で分析できる．さらに，CCX 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，クローンペアマッピング [7] が利用されている．

比較結果を可視化し，表示される散布図の例を**図 1** に示す [8]．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び，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．有色点の場合，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，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 (左上矢印)，図 1 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 2** のような Code View で確認できる．この Code View には 3 つのペインがあり，ペイン a は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 は散布図縦軸に対応するファイルのソースコード，ペイン c は散布図横軸に対応するファイルのソースコードである．

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 1 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
(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 2 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，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)

2.3.2 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．2.3.1 節で述べたように，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．以下，2.3.1 節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

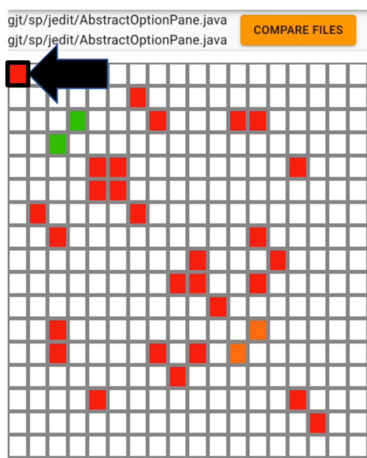

図 1　表示される散布図の例

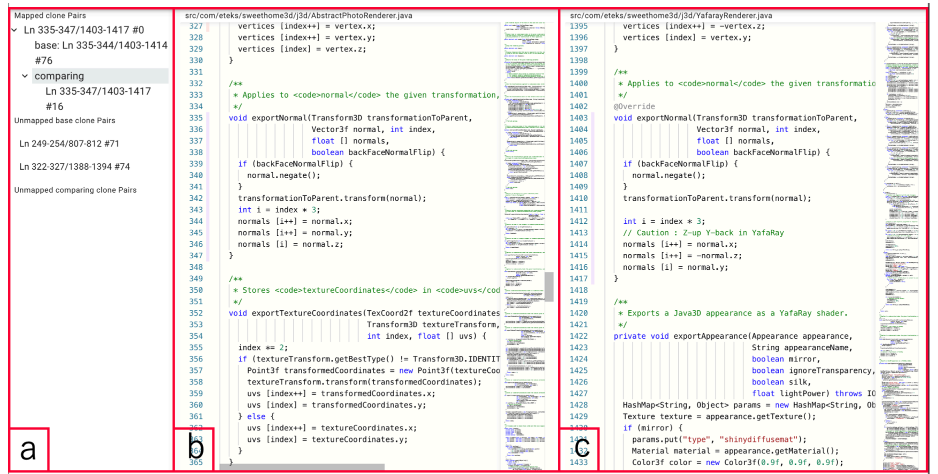

図 2　散布図表示のための Code View

は，まずは 2.3.1 節で紹介した (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，表示される図 1 の散布図では，順に 1 マスずつ選択してパスを確認することでしか，ファイルを発見できない．

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，2.3.1 節で述べた (2) で特定のファイルを探した後，(3) から (6) を繰り返す必要がある．この作業は，対象ファイルの縦の行の，全ての有色点を選択し，これらの 1 マスごとに図 2 のペイン a を確認する必要がある．

この (3) から (6) の繰り返し回数は，対象ファイルの縦行の有色点の個数に依存し，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，2.3.1 節で述べた (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．

表 1　検出結果の例

	検出結果 1 での クローンペア数	検出結果 2 での クローンペア数
ファイル (A,A)	1	1
ファイル (B,B)	50	50

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数がわかりにくくなってしまう．例えば，クローンペアを検出した結果が**表 1** のようになる場合を考える．ファイル (A,A) という表記は，ファイル A 内でクローンペアとなるコードクローンのことを表している．この時，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，散布図におけるファイル (A,A) とファイル (B,B) のマスは，共に赤色で表示される．しかし，実際に修正を検討するクローンペアの個数は，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，検出数の差異を散布図では理解することができない．

3.　既存ツールの拡張に向けた調査

2.3.2 節で述べた問題点 3 から，ファイルごとにコードクローン検出数を表示し，それらの差異を可視化するといった改善案を考えた．しかし，コードの同時修正を検討する必要があるコード片において，ファイル間での検出数の差異を理解することが有効であるか不明であった．これを明らかにするため，3 つの Java アプリケーションを題材とした調査を行った．

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

ⓒ 1959 Information Processing Society of Japan

表 2　各ソフトウェアの詳細

	10 個の revision 変化が行われた期間
jEdit	2016/11/29 - 2017/01/05
Logisim	2011/03/15 - 2011/03/21
FreeMind	2013/10/16 - 2013/12/08

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明らかにするため，調査を行った．本調査では，特定の検出ツールでしか検出することができないような検出ツールに依存するコードクローンの内，実際に revision 間で修正されたコード片を対象に調査する．調査の対象となった Java アプリケーションは，jEdit[*1], Logisim[*2], FreeMind[*3]の 3 つである．使用した検出ツールは，CCFinderSW [11], CCVolti [13], Deckard [5], NiCad [10] の 4 種類である．

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，コードクローンが検出されたファイル数を確認し，ファイル単位ごとの検出コードクローン数を計算したのち、降順でファイルの並び順を決める
(3) (1) の revision と，その次の revision の間で修正が行われたコード片を確認する
(4) (3) のコード片を含んでいるコードクローンが (1) で検出されているかどうかを調べる
(5) 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 (2) での計算結果の上位何番目であるかも確認する

上記の調査を，各ソフトウェアそれぞれ 10 個の revision 間で調査する．調査対象の詳細を**表 2** 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．

3.3　調査結果

調査結果を**表 3** に示す．ただし，表 3 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のことである．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．検出ファイル数は，コードクローンが検出されたファイル数である．上位何番目かは，コードクローンが検出されたファイルを検出数で降順に並べた際，対象のコード片が存在するファイルが上位何番目であったかを表している．対象のコード片の個数以外の値は，10 個の revision 間での結果の平均値 (小数点第 2 以下切り捨て) である．

[*1] http://www.jedit.org/?page=features
[*2] https://sourceforge.net/projects/circuit/
[*3] https://sourceforge.net/projects/freemind/

表 3　10 個の revision に対して行った調査結果

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

3.4　考察

コードクローンが検出されたファイルを検出数で降順に並べた際に，revision 間で修正されたコード片のコードクローンを含むファイルが，全体の上位何%であるかを計算すると，表 3 の結果から，jEdit は 11.9%,Logisim は 25.5%,FreeMind は 34.8%であった．この結果から，revision 間で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分かった．このことから，2.3.2 節で述べた，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.　提案手法

本研究では，2.3.2 節の問題点と 3 節の調査結果に基づき，コードの同時修正作業に着目した新たな表示手法を提案する．本節では，手法のキーアイデアを述べ，その後，どのような UI でキーアイデアを表現するのかを説明する．

4.1　キーアイデア

ファイルツリーを表示する

同一のソースコード集合に対して，2 種類の検出ツールの検出結果で，コードクローンが検出されたファイルのパスをツリー状で表示する．これにより，2.3.2 節で述べた問題点 1 を解決する．ファイルツリーの表示により，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 種類の検出ツールの検出結果を比較し，その結果をファイル内で検出されたコードクローンのクローンセット単位で表示する．これにより，2.3.2 節で述べた問題点 2 を解決する．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ることで，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種類の検出ツールの検出結果を比較し，その合計コードクローン数をファイルごとに表示する．これにより，2.3.2節で述べた問題点3を解決する．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．

4.2 キーアイデアを実装するためのUI

4.1節で述べたキーアイデアを基に，同一のソースコード集合に対して，異なる2種類のコードクローン検出結果を比較した内容を表示する．

最大限画面遷移の回数を少なくするため，全体の画面描画は結果表示の1回のみにし，必要な部分のみを変化させる．画面上部には，3つ目のキーアイデアを反映したグラフを表示する．このグラフにより，散布図表示で理解できていた検出結果の差異 (割合) と，コードクローン数をファイル単位で確認できるようにする．これにより，問題点3を解決する．画面下部には，1つ目と2つ目のキーアイデアを反映したViewを表示する．このViewでは，修正コードクローンの確認を行うことを目的とする．4つのペインに分割し，ファイルツリー，クローンセット，確認するコードクローンを含むファイルのソースコード，そのクローンペアの実際のソースコードをそれぞれ表示する．これにより，問題点1,2を解決する．

5. 実装ツール

本研究では，CCXを拡張することで，4節で述べた手法を基に，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 [7] の過程で求める，マッピングされたコードクローンを利用する．あるソースコード集合に対して，異なる2つの検出ツールから得た出力結果を比較すると，**図3**のような画面が表示される．

5.1 複合グラフ

本研究で開発した，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 (図3赤枠部分)．

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[*4]を加えた値に設定される．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で昇順にソートされ，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る．図3中のiに注目すると，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，ii，iiiの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も分かる．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．このソート順により，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できるようになる．

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．

5.2 CloneSet View

複合グラフと同時に表示される，ファイル閲覧表示をCloneSet Viewと呼ぶ (図3緑枠部分)．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．

ペインaは，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するファイルツリーである．ペインbは，ファイルツリーで選択し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するクローンリストである．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が一致するコードクローンの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が一致するコードクローンの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トは，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

[*4] 縦軸1の積み上げ棒グラフがはっきりと認識できる値を，試行錯誤の結果決定した．

図 3 表示画面

終了行の組み合わせで確認できる．

また，ペイン c，d はソースコードを表示するペインである．c はファイルツリーで選択したファイルのソースコードを，d はペイン c でハイライトされている e のコード片のクローンペアとそのファイルパスを表示する．ハイライトされている e 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ペイン c で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 a 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作業を容易にし，問題点 1 を解決する．また，ペイン b のクローンリストで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るようになった．これにより，散布図と Code View の行き来が必要だった問題点 2 を解決する．さらに，ペイン c により，バグを含み修正対象となるソースコードを，コードクローンをハイライトしながら確認できるようになった．そして，ペイン d により，修正を検討するクローンペアを順に確認できるようになった．

6. 評価実験

本節では，4 つのソフトウェア Sweet Home 3D[*5], jEdit[*6], Process Hacker[*7]および snns[*8]に対して，CCX 上で使用できる 2 つの検出ツール CCFinderSW [11], NiCad [10] 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．

表 4 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

6.1 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，2.3.1 節で紹介した散布図を利用する既存ツールと，5 節で述べ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 4** 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 5** に示す．

*5 http://www.sweethome3d.com/
*6 http://www.jedit.org/?page=features
*7 https://processhacker.sourceforge.io/
*8 http://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

表 5　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

6.2　実験手順

本節では，まず，何らかのコード片修正時に行うコードクローン分析作業において，必要な手順を述べる．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，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．

6.2.1　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，2.2 節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

6.2.2　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 2.3.1 節で記載した，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，B は NiCad である．手順 3 の対象のファイルとは，表 5 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．

2.3.1 節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できる．そのため，6.2.1 節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．

6.2.3　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 3 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

表 6　画面遷移回数 (全体/一部)[回]

	既存ツール	新規ツール
Sweet Home 3D	23/1	1/3
jEdit	5/1	1/3
Process Hacker	27/1	1/3
snns	3/1	1/3
平均	14.5/1	1/3

表 7　マウスクリック回数 [回]

	既存ツール	新規ツール
Sweet Home 3D	31/66	2/97
jEdit	67/6	2/30
Process Hacker	74/42	3/27
snns	65/4	2/13
平均	59.2/29.5	2.2/44

表 8　確認クローン (個数 [個]/行数 [行])

	既存ツール	新規ツール
Sweet Home 3D	402/5190	239/3952
jEdit	68/1033	56/886
Process Hacker	52/1239	48/1156
snns	44/463	32/356
平均	141.5/1981.2	86.2/1587.5

(3) バグ箇所が含まれるファイルをファイルツリーから探し，選択する
(4) クローンリスト中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できる．そのため，6.2.1 節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．

6.3　実験結果

表 6 は，画面遷移の回数を測定した実験結果である．全体の画面遷移回数と，一部の画面遷移回数を全体/一部の形で表記している．

表 7 は，マウスクリック回数を測定した実験結果である．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

© 1959 Information Processing Society of Japan

での回数/その他の回数の形で表記している．

表 8 は，確認クローン数を測定した実験結果である．確認したコードクローンの個数と，その行数を，個数/行数の形で表記している．

複合グラフの表示において，revision 間で修正されたコード片を含んだファイルが左から何番目であったかを測定した結果，Sweet Home 3D では 49 番目，jEdit では 78 番目，Process Hacker では 66 番目，snns では 13 番目であった．これらの値を平均すると，51.5 番目であった．

6.4 考察

6.3 節の実験結果から，新規ツールを使用した方が，全体の画面遷移数が減少していることが分かる．既存ツールは，図 1 の散布図と図 2 の Code View を行き来する必要があった．そのため，2.3.2 節の問題点 2 で述べたように，様々なファイルにクローンペアが広がっているほど，全体の画面遷移数は増加した．これに対し新規ツールは，全体の画面遷移数は，ソフトウェアによらず図 3 の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数は少なくなり，効率よく修正を検討することができる．

また，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．本研究で実装したツールで，この役割を担うのは，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程度である．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，ファイルツリーで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．

さらに，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．

7. まとめ

本研究では，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．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，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．

謝辞 本研究は JSPS 科研費 21K18302, 20H04166 の助成を受けたものです．

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] 肥後芳樹，吉田則裕：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).
[3] 本田紘貴，徳井翔梧，横井一輝，崔　恩瀞，吉田則裕，井上克郎：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).
[4] 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).
[5] Jiang, L., Misherghi, G., Su, Z. 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).
[6] 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, pp. 654–670 (2002).
[7] 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).
[8] 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, コンピュータソフトウェア，Vol. 39, No. 4, pp. 129–143 (Nov. 2022).
[9] Mondal, M., Roy, B., Roy, C. K. and Schneider, K. A.: An empirical study on bug propagation through code cloning, *Journal of Systems and Software*, Vol. 158, p. 110407 (2019).
[10] Roy, C. K. R. and Cordy, J. R.: 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).
[11] 瀬村雄一，吉田則裕，崔　恩瀞，井上克朗：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).
[12] Wang, T., Harman, M., Jia, Y. and Krinke., J.: 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).
[13] 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ブ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).

© 1959 Information Processing Society of Japan