特別研究報告

題目

ソースコードの同時修正支援における関数クローン検出ツールの有効性調査

指導教員

井上 克郎 教授

報告者

沼田 聖也

平成 28 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 27 年度 特別研究報告

ソースコードの同時修正支援における関数クローン検出ツールの有効性調査

沼田 聖也

内容梗概

コードクローンとは，ソースコード中に存在する同一，あるいは類似した部分を持つコード片を意味し，これは主に既存のコード片のコピーアンドペーストによって生成される．コードクローンは，ソフトウェア保守を困難にしている大きな要因の 1 つとして挙げられている．一般的に，互いにコードクローンになるコード片のことをクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合のことをクローンセットと呼ぶ．

コードクローンに対する保守作業の 1 つとして，ソースコードの同時修正が挙げられる．例えば，あるコード片にバグが見つかった場合，そのコード片のコードクローンにもバグがある可能性が高い．そのため，開発者はコードクローンにバグが見つかった場合，同一クローンセット内に含まれる全てのコードクローンに対して同一の修正をするか検討する必要がある．しかし，開発者がすべてのコードクローンを手作業で見つけて管理するのは困難である．そこで，開発者を支援するためにコードクローン検出ツールが利用される．

山中らは，情報検索技術に基づいて関数単位のコードクローンを検出するツールを開発した．このツールに対して，検出精度の評価は行われているが，ソースコードの同時修正支援における有効性という観点から評価実験は行っていない．そこで，本論文ではこのツールのソースコードの同時修正を行う場合の有効性を調査した．具体的には，同時修正するべきコードクローンを検出できるかどうかという観点から関数クローン検出ツールの有効性を判断した．

本論文では，関数クローン検出ツールに対して 2 つの評価実験を行った．1 つ目の評価実験では，バグを含むコードクローンを検出する際の性能を評価した．バグを含むコードクローンは同時修正されるべきであるため，これを検出できるかどうかによって有効性を調査できる．その評価にあたって，評価の指標とするために，字句解析ベースのコードクローン検出ツールとして多くの研究で利用されている CCFinder でも検出を行った．その結果，バグを含むコードクローンを検出する際に，関数クローン検出ツールは十分な性能を発揮した．

2 つ目の評価実験では，Late Propagation を関数クローン検出ツールで検出できるかを調査した．Late Propagation とは，クローンペアに対して一貫性を破壊する編集が行われた後に，一貫性を回復する編集が行われる現象である．この現象はバグを含みやすい現象として知られている．Late Propagation を引き起こす一貫性を破壊する編集は，後に一貫性

を回復する編集が行われるということから，そのコードクローンは本来同時修正するべきであるため，これを検出できるかどうかによって有効性を調査できる．その結果は，Late Propagation をいくつか検出することができ，一貫性を破壊する編集を検出することができた．これら 2 つの評価実験から，関数クローン検出ツールはソースコードの同時修正支援において，十分有効にはたらくと判断した．

主な用語

コードクローン
ソフトウェア保守
ソースコードの同時修正
バグ修正
late propagation

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同一，あるいは類似した部分を持つコード片を意味し，これは主に既存のコード片のコピーアンドペーストによって生成される [7]．コードクローンは，ソフトウェア保守を困難にしている大きな要因の 1 つとして挙げられている．一般的に，互いにコードクローンになるコード片のことをクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合のことをクローンセットと呼ぶ．

コードクローンに対する保守作業の 1 つとして，ソースコードの同時修正が挙げられる [8][12]．例えば，あるコード片にバグが見つかった場合，そのコード片のコードクローンにもバグが含まれる可能性が高い．そのため，開発者はコードクローンにバグが見つかった場合，同一クローンセット内に含まれる全てのコードクローン対して同一の修正をするか検討する必要がある．しかし，すべてのコードクローンを開発者が手作業で見つけて管理するのは困難である．そこで，開発者を支援するためにコードクローン検出ツールが利用される [11]．

山中らは，情報検索技術に基づいて関数単位のコードクローンを検出するツールを開発した [21]．このツールのことを本論文では，関数クローン検出ツールと呼ぶことにする．このツールは，検出精度の評価実験と，他の関数単位のコードクローン検出ツールとの検出精度と検出時間の比較実験は行っているが，ソースコードの同時修正支援における有効性という観点から評価実験は行っていない．そこで，本論文ではソースコードの同時修正を行う場合の関数クローン検出ツールの有効性を調査した．具体的には，同時修正するべきコードクローンを検出できるかどうかという観点からツールの有効性を判断した．同時修正するべきコードクローンをうまく検出できれば，開発者は検出したコードクローンに対して同時修正すれば良く，また検出したコードクローンの内，同時修正するべきコードクローンの割合が高ければ，検出したコードクローンの中から，同時修正するべきコードクローンを判断するコストを下げることができる．

本論文では，関数クローン検出ツールに対して 2 つの評価実験を行った．1 つ目の評価実験では，バグを含むコードクローンを検出する際の性能の評価を行った．バグを含むコードクローンは同時修正されるべきであるため，これを検出できるかどうかによって有効性を調査できる．その評価にあたって，評価の指標とするために，字句解析ベースのコードクローン検出ツールとして多くの研究で利用されている CCFinder でも検出を行った [13]．評価実験では，Li らが用意した評価セットを用いた [14][15]．この評価セットでは，バグが含まれているコード片とそのコード片と同じバグを含むコードクローンが事例としてリストの形で挙げられている．これらのバグが含まれているコード片のコードクローンを，見つけることが

できるかどうかで関数クローン検出ツールと CCFinder の比較を行った．その際の評価指標には，Li らが用いた false positive と false negative による指標を最初に適用し，続いて，再現率と適合率と F 値という指標を適用した．結果は，関数クローン検出ツールは CCFinder に対して劣らぬ十分な性能を発揮した．

2 つ目の評価では，関数クローン検出ツールで Late Propagation の検出を行うことで評価を行った．Late Propagation とは，クローンペアに対して，一貫性を破壊する編集が行われた後に一貫性を回復する編集が行われる現象である．この現象はバグの発生しやすい危険度の指標となりうる [3]．Late Propagation を引き起こす一貫性を破壊する編集は，後に一貫性を回復する編集が行われるということから，そのコードクローンは本来同時修正するべきであるため，これを検出できるかどうかによって有効性を調査できる．結果としては，Late Propagation をいくつか検出することができ，一貫性を破壊する編集を検出することができた．これら 2 つの評価から，ソースコードの同時修正支援において，関数クローン検出ツールは十分有効にはたらくと判断した．

以降，2 節では本研究の背景について説明し，3 節では本研究で行った 2 種類の評価の評価方法を説明する．4 節では評価の結果を説明し 5 節でその考察を述べ，最後に，6 節で本研究のまとめと今後の課題を述べる．

2　背景

本節では本研究の背景として，コードクローン，コードクローン検出ツール，本研究で評価を行う関数クローン検出ツール，Late Propagation についての説明を行う．

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同一，あるいは類似した部分を持つコード片のことを意味し，一般的にこのコードクローンの存在は，ソフトウェアの保守を困難にするものと言われている [7]．コードクローンの発生の主たる要因は，既存のソースコードのコピーアンドペーストによる再利用である．似たような処理を行うソースコードを書く際には，一から書くよりも既存のソースコードを再利用することが多い．他の要因には，定型処理による発生，コード自動生成ツールによる発生，偶然の一致による発生等も挙げられる [4]．一般的に，互いにコードクローンになるコード片の対のことをクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合のことをクローンセットと呼ぶ．

コードクローンに対する保守作業の 1 つに，ソースコードの同時修正が挙げられる [8][12]．例えば，あるコード片にバグが見つかった際に，そのコード片のコードクローンにも同様のバグが含まれている可能性が高い．開発者は，コード片にバグが見つかった際には類似したバグ全てに対して同一の修正をする必要があるかどうかを確認しなければならない．しかし，すべてのコードクローンを開発者が手作業で見つけて管理するのは困難である．そこで，開発者を支援するためにコードクローン検出ツールが利用される [11]．

2.1.1　コードクローンの定義

コードクローンには，普遍的定義は存在しない．本論文では，コードクローンの定義として以下の 4 つのタイプの分類を用いる [17][21]．

タイプ 1

空白の有無，レイアウト，コメントの有無などの違いを除き完全に一致する．

タイプ 2

タイプ 1 の違いに加えて，変数名などのユーザ定義名，関数の型などが異なる．

タイプ 3

タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われている．

タイプ 4

類似した処理を実行するが，構文上の実装が異なる．

タイプ 4 のコードクローンとしては，以下のものが挙げられる．

- 条件分岐処理や繰返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

2.1.2 コードクローンの検出

コードクローン検出における粒度はいくつかある [11]．以下にその例を挙げる．

文字

プログラムテキストの構成要素を文字列とみなして，文字列のパターンマッチングを行い，コードクローンを発見する．この方法は，空白等も含めて厳密に同形のコード片しか検出されず，また，文字列レベルでの等価判定は，手間がかかり，大規模なプログラムの解析には向かないため，文字列レベルの比較を行う検出ツールは無い．

字句 (トークン)

プログラムの字句解析を行った後，その字句を要素とした系列に対してコードクローンを発見する．字句解析を行うことにより，空白やコメントを無視することができ，識別子や定数等の特定の種類の字句を特殊な 1 つの字句に固定することで，変数名や関数名の変更されたコード片もコードクローンとして検出することができる．

行

字句よりも粗い粒度にして，プログラムのテキストの各行をハッシュ関数を用いてハッシュ値に変換し，そのハッシュ値の列を対象として，コードクローンを発見する [5]．大きなプログラムテキストを比較的小さな要素列に圧縮できるため，効率良くコードクローンを見つけることができるが，空行の削除や挿入，改行位置の変更によって，検出できなくなる場合があるので，あらかじめ空行の除去等が必要である．

文

プログラムテキスト中の文を取り出して，それをハッシュ関数で 1 つの要素にし，その系列に対してコードクローンを発見する [16]．この方法では，文の認識のために簡単な字句解析と構文解析が必要だが，空白やコメント等の影響は受けない．変数名等の変更に対応するためには，識別子等をパラメータ化する必要がある．

関数

プログラムテキストの関数を 1 つの要素とし，等価な要素対を見つけることでコードクローンを発見する [21]．この方式では，関数全体ではなく，一部のみがコードクローンになっているものを発見することはできない．コードクローン検出のために，プログラムの性質を計測した特徴メトリクスを用いる．

本論文で，関数クローン検出ツールの評価のために利用したコードクローン検出ツールを以下で説明する．

CCFinder

CCFinder はトークン単位のコードクローンを検出するツールである [13]．このツールはソースコードをトークン単位に分解し，変数名や関数名等は 1 つのトークンと考えることにして，ソースコードに対してトークン単位で直接比較を行うことによって，変数名や関数名等が異なるようなタイプ 2 のコードクローンまでを検出することができる．検出時間も非常に高速であり，数百万規模のシステムに対しても，実用的な時間でコードクローンを検出することができる．CCFinder は国内外で広く利用され，コードクローン検出ツールのデファクトスタンダードとなりつつある．そこでこのツールを，関数クローン検出ツールの評価指標とするために用いた．

CBCD

CBCD は PDG（Program Dependence Graph）[6] を用いて，バグを含むコード片のコードクローンを検出するツールである．このツールはバグを含むコード断片を発見する際の能力が他のツールと比較されており，その際の評価セットも公開されている [14][15]．本論文ではこのツールを評価した評価セットを利用している．

2.2 関数クローン検出ツール

関数単位のコードクローンを検出する関数クローン検出ツール [21] の説明を行う．関数クローン検出ツールでは情報検索技術を利用し，タイプ 1 からタイプ 4 のすべてのタイプの関数単位のコードクローンを検出することができる．ここでは，その検出手法を述べる．

関数クローン検出ツールは，入力されたソースコード中のワードに基づいて各関数を特徴ベクトルに変換する．ここでワードは以下の 2 つを示す．

- 変数や関数などにつけられた識別子名を構成する単語
- 条件文や繰返し文などの構文に利用される予約語

そして，その特徴ベクトル間の類似度を計算して，クローンペアの集合をリストとして出力する．また，検出の高速化のために，類似度の計算の直前に LSH（Locality-Sensitive Hashing）

アルゴリズム [10] を用いて特徴ベクトルのクラスタリングを行っている．関数クローン検出ツールのコードクローン検出手法は大きく 4 つのステップに分けられる．その関数クローン検出ツールのコードクローン検出プロセスを図 1 に示す．

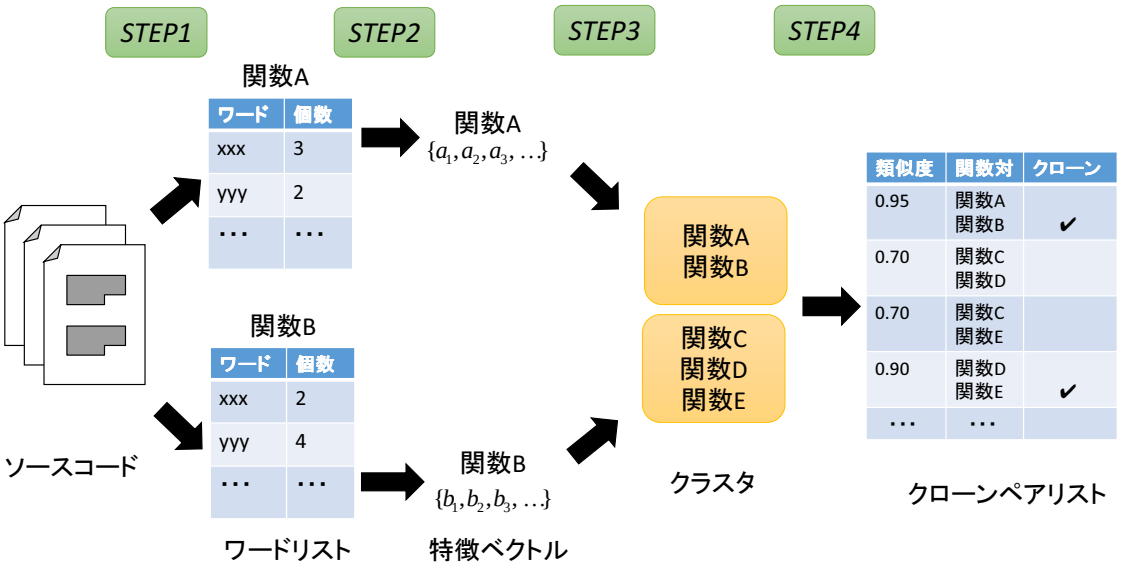

図 1: 関数クローン検出ツールのコードクローン検出プロセス

STEP1：ワードの抽出

このステップでは，ソースコード中の各関数からワードの抽出を行う．識別子名が複数の単語で構成されている場合，分割を行い新たに複数のワードとする．その分割方法には，ハイフン等の区切り記号（デリミタ）による分割と，識別子名中の大文字になっているアルファベットによる分割がある．繰返し文等でよく用いられる“ i ”や“ j ”等，意味情報が込められていない変数も同一のものとして扱うために，2 文字以下の識別子名はすべて同一のワードとして認識する．

STEP2：特徴ベクトルの計算

STEP1 で抽出したワードに対して TF-IDF 法 [2] を利用して各ワードの重みを計算し，その値を特徴量として利用して，各関数を特徴ベクトルに変換する．TF-IDF 法による値は tf 値（関数中のワードの出現頻度）と idf 値（ソースコード全体のワードの希少さ）の積で与えられる．この手法での tf 値と idf 値は以下の計算式で与えられる．

$$tf_x = \frac{\text{関数中のワード x の出現回数}}{\text{関数中に出現する全ワードの出現回数の合計}}$$

$$idf_x = log \frac{\text{全関数の数}}{\text{ワード x が出現する関数の数}}$$

この手法では，全関数中の各ワードに対して重みを計算し，それらを特徴量として用いることによって特徴ベクトルを求めているため，各関数の特徴ベクトルの次元はソースコード中に存在する全ワードの数となる．

STEP3：特徴ベクトルのクラスタリング

STEP2 で計算した各関数の特徴ベクトルに対してクラスタリングを行うことによって，クローンペアになりうる候補を絞る．ここでは，LSH アルゴリズム [10] を用いて特徴ベクトルのクラスタリングを行う．LSH アルゴリズムを利用することによって，クエリとして 1 つの特徴ベクトルを与えると，特徴ベクトル集合からそのクエリと近似した特徴ベクトル集合のクラスタを取得することができ，クローンペアになりうる候補を絞ることによって，検出時間にかかる計算コストを削減している．なお，LSH アルゴリズムが実装されている E2LSH[1] をこの手法では利用している．

STEP4：特徴ベクトルの類似度の計算

最後に，STEP3 で求めた各クラスタ中の関数のついに対してコサイン類似度を用いてクローンペアであるか否かの判定を行う．次元が d である 2 つの特徴ベクトル $\vec{a}$，$\vec{b}$ 間の類似度は以下の式で表される．

$$cos(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} a_i^2} \sqrt{\sum_{i=1}^{d} b_i^2}}$$

TF-IDF 法の計算式より，特徴量は常に正の値をとるため，コサイン類似度は 0 から 1 の範囲となる．もし，コサイン類似度が閾値以上であれば，その関数の対はクローンペアであると判定する．

こうして STEP1 から STEP4 を経て検出したクローンペアリストを結果として出力する．

また，関数クローン検出ツールを用いた変更管理システム [18] が提案されている．本論文では以後このシステムのことを変更管理システムと呼ぶ．変更管理システムは，入力されたソースコードの 2 つのバージョン間で変更が行われたコードクローンを，関数クローン検出ツールを用いて自動的に検出して通知する．その通知には，コードクローンの分類，クローンセットの分類を利用する [20]．変更管理システムのコードクローンの分類は以下の 5 つである．

Stable Clone

2 つのバージョン間で変更がないコードクローン．

Modified Clone

コードクローンに対して編集が行われたが，編集後も同じクローンセットに属するコードクローン．

Moved Clone

コードクローンに対して大幅な編集がなされたため，異なるクローンセットに属するようになったコードクローン．

Added Clone

古いバージョンではコードクローンではなかったが，新しいバージョンで発生したコードクローン．

Deleted Clone

古いバージョンではコードクローンであったが，新しいバージョンでは除去されたコードクローン．

クローンセットの分類は以下の 4 つである．

Stable Clone Set

2 つのバージョンにわたって，属するコードクローンがすべて Stable Clone に分類されるクローンセット．

Changed Clone Set

2 つのバージョンにわたって，属するコードクローンに 1 つでも Stable Clone 以外のコードクローンが含まれるクローンセット．

New Clone Set

2 つのバージョン間で，古いバージョンには存在しなかったが，新しいバージョンのみに存在するクローンセット．

Deleted Clone Set

2 つのバージョン間で，古いバージョンには存在したが，新しいバージョンでは除去されたクローンセット．

通知方法には，html 形式による通知があり，html ファイルにはクローンセットの分類，コードクローンの ID，コードクローンの分類，コードクローンが含まれるファイル名，ファイル内でのコードクローンの位置，コードクローンのメソッド名が図 4 で示すような形で含まれている．また，Changed Clone Set と Deleted Clone Set では，前バージョンのコード

Changed Clone Set

クローンセットID:50

ID	分類	ファイル名	位置	メソッド名
50.1	MODIFIED	src¥ab¥cd¥efg.java	30.1-62.1	Abc()
50.2	MODIFIED	src¥ab¥cd¥hij.java	71.1-103.1	Defg()
50.3	STABLE	src¥ab¥cd¥klm.java	50.1-82.1	Hij()

クローンセットID:51

ID	分類	ファイル名	位置	メソッド名
51.1	ADDED	src¥ab¥cd¥no.java	100.1-140.1	Kl()
51.2	ADDED	src¥ef¥gh¥pq.java	150.1-190.1	Mn()
51.3	MOVED	src¥ij¥kl¥stu.java	30.1-70.1	Op()
前バージョンのコードクローン				
51.4	DELETED	Src¥vw.java	82.1-120.1	XyZ()

図 2: 変更管理システムの通知内容例

クローンという項目があり，ここには，古いバージョンではそのクローンセットに含まれたが，新しいバージョンではコードクローンではなくなったものが表示される．

関数クローン検出ツールは評価実験として，コーパス [19] を用いた検出精度の評価と，既存の関数単位のクローン検出ツールとの検出精度と検出時間の比較は行われているが，ソースコードの同時修正における評価は行われていないため，ソースコードの同時修正を支援するにあたっての有効性はまだ示されていない．そこで本論文では，同時修正するべきコードクローンを検出できるかどうかでソースコードの同時修正における有効性を調査した．

2.3 Late Propagation

Late Propagation とは，クローンペアに対して，一貫性を破壊する編集が行われた後に一貫性を回復する編集が行われる現象である．その様子を図 2 に示す．

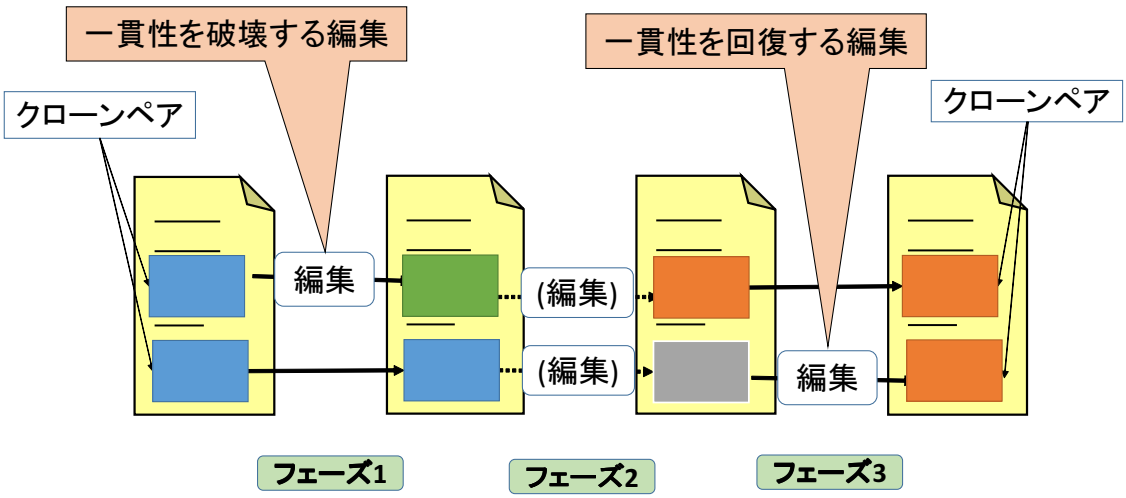

図 3: Late Propagation の流れ

まず，図 2 のようにクローンペアであった 2 つのコード片が，フェーズ 1 で一貫性を破壊する編集が行われることにより，クローンペアでなくなる．次に，その後のバージョン変更が行われるフェーズ 2 で各々のコード片の編集が行われる．フェーズ 2 では何も編集が行われないこともある．そして，最後にフェーズ 3 で一貫性を回復する編集が行われることにより，2 つのコード片が再びクローンペアに戻る．この一連の現象が Late Propagation である．Late Propagation はクローンペアであるコード片 A，B の編集の種類によって 8 つに分類される．その分類を表 1 に示す．表中のハイフンはクローンペアに編集が行われていないことを示している．また，LP の分類の例として図 3 に LP8 の例を示す．既存の研究で，LP7 と LP8 がバグを含む危険性が高いと言われている [3].

Late Propagation を引き起こす一貫性を破壊する編集は，後に一貫性を回復する編集が行われるということから，本来同時修正するべきであると考えられる．この一貫性を破壊する編集を，関数クローン検出ツールで検出できるかどうかの評価はまだされていない．そこで，本論文では関数クローン検出ツールを用いて Late Propagation を検出することによって一貫性を破壊する編集を検出して評価を行った．

表 1: Late Propagation 分類表

タイプ	フェーズ 1 で 編集されたコード片	フェーズ 2 で 編集されたコード片	フェーズ 3 で 編集されたコード片
LP1	A	- , A	B
LP2	A	B , AB	B
LP3	A	- , A	AB
LP4	A	B , AB	A
LP5	A	- , A , B , AB	AB
LP6	AB	- , A , B , AB	A
LP7	AB	- , A , AB	AB
LP8	A	- , A	A

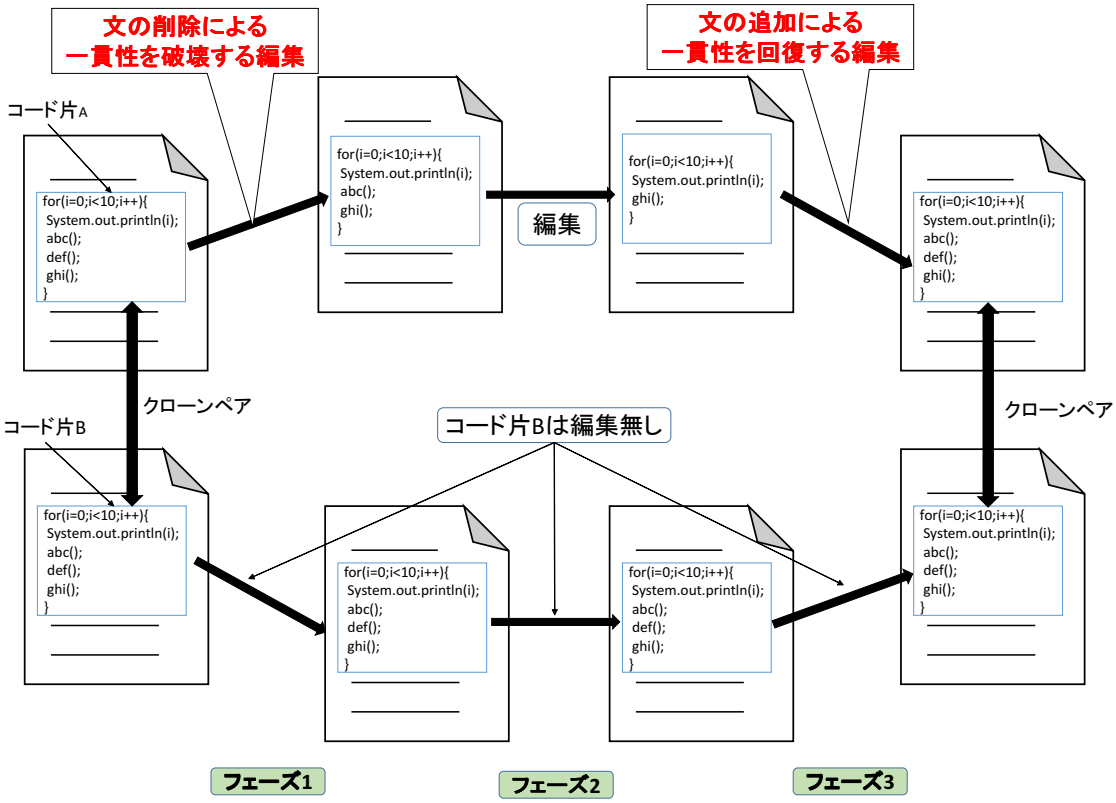

図 4: LP8 の例

3　調査手法

本研究では，ソースコードの同時修正支援における関数クローン検出ツールの有効性を調査するために 2 種類の評価を行った．1 つ目の評価ではバグを含むコードクローン検出における評価を行い，2 つ目の評価では Late Propagation 検出に基づく評価を行った．以降，2 つの評価の詳細について述べる．

3.1　バグを含むコードクローン検出における評価

バグを含むコードクローン検出における評価の調査手法を説明する．この評価では，バグを含むコード片のコードクローンを検出する際の関数クローン検出ツールの有効性を調査した．評価セットとして，Li らが用意したものを利用した [14][15]．最初にその評価セットの説明を行い，その後，調査の手順を説明する．

3.1.1　調査対象の評価セット

Li らはオープンソースソフトウェアから，バグを含むコード片とそのコードクローンを見つけた．その調査対象のオープンソースソフトウェアとして，Git，Linux kernel，PostgreSQL の 3 つを利用している．Li らが，これらのオープンソースソフトウェアを利用したのには 3 つの理由がある．その理由は，(1) これら 3 つが主に C/C++を使って書かれており，(2) それらのバージョン履歴からバグを含むコードとそのコードクローンを見つけることが可能であり，(3)Git は 10 万行以上，PostgreSQL は 30 万行以上，Linux kernel は 100 万行以上ものコードであり，評価において十分なスケーラビリティを有しているという理由からである．Li らは，バグを含むコード片と，そのコード片と同じバグを持つコードクローンの事例をテクニカルレポートで表 2 に示すような形で公開している [14]．そのため，バグを含むコード片の git リポジトリのコミット ID と，同じバグを持つコードクローンの git リポジトリのコミット ID を得ることができる．また，Li らはバグ修正の際のコミット情報も表 3 に示すような形で公開している．

本研究ではこのテクニカルレポートで公開されている事例を利用する．しかし，その事例の中にはバグを含むコード片とそのコードクローンが同じ関数内に存在するものも存在する．関数クローン検出ツールは関数単位のコードクローンを検出するため，そのようなコードクローンは検出することができない．よって，そのようなコードクローンは調査対象から取り除いて，全 38 種類のクローンセットに含まれる 58 個のコードクローンの事例に対して調査を行った．

表 2: バグを含むコード片とそのコードクローンの例 (Li らのレポートより引用 [14])

ID	コミット ID	バグを含むコード片	コードクローン	クローンのタイプ
1	PostgreSQL-2618fcd	2618fcd - pg_dump.c: 2672-2675 sprintf(q, ”CREATE %s INDEX %s on %s using %s (”,(strcmp(indinfo[i].indisunique, ”t”)==0) ? ”UNIQUE” :””, fmtld(indinfo[i].indexrelname), fmtld(indinfo[i].indrelname), indinfo[i].indrelname),	87d96ed - pg_dump.c: 2673-2676 sprintf(q, ”CREATE %s INDEX %s on %s using %s (”,(strcmp(indinfo[i].indisunique, ”t”)==0) ? ”UNIQUE” :””, fmtld(indinfo[i].indexrelname), fmtld(indinfo[i].indrelname), indinfo[i].indrelname),	1

2	PostgreSQL-dcb09b5	dcb09b5 - Plperl.c: 2132-2133
		
		perm_fmgr_info(typeStruct-> typoutput, &(prodesc-> arg_out_func[i]));
			dcb09b5 - Plperl.c: 2088-2088
			
			perm_fmgr_info(typeStruct-> typoutput, &(prodesc-> result_in_func));
			
			dcb09b5 - Plperl.c: 2720-2720
			
			perm_fmgr_info(typlnput, &(qdesc-> arginfuncs[i]));
				3

3	PostgreSQL-9dbfcc2	9dbfcc2 - Plperl.c: 758-763 for (i = 0; i < tupdesc-> natts; i++){/********** Get the attribute name**********/ attname = tupdesc-> attrs[i]-> attname.data;	6d239ee - Plperl.c: 758-763 for (i = 0; i < tupdesc-> natts; i++){/********** Get the attribute name**********/ attname = tupdesc-> attrs[i]-> attname.data;	1

表 3: バグ修正の際のコミット情報の例 (Li らのレポートより引用 [14])

ID	コミット ID	バグのコミット情報	コードクローンのコミット情報
5	PostgreSQL-9dbfcc2	http://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=fe055e928095658eb2a8cD52ff32f090720de3de looks like plperl has same bug as pltcl. for(i = 0; i < tupdesc-> natts; i++){ + /* ignore dropped attributes */ + if (tupdesc-> attrs[i]-> attisdropped) + continue;	http://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=9dbfcc22613379e89283282db5cd616898bf6e4f Fix some problems with dropped columns in pltcl functions.

3.1.2 調査手順

関数クローン検出ツールの性能を比べるために，Li らが用意した評価セットを用いて以下の手順で調査を行った．CCFinder を評価の指標として用いたのは，CCFinder は多くの実験で利用されており実績を持っているからである．

1. Git，Linux kernel，PostgreSQL の git リポジトリを取得する．
2. バグを含むコード片と，それと同じバグを持つコードクローンの例として挙げられているコード片が含まれているコミットのスナップショットを取得する．
3. 関数クローン検出ツール，CCFinder でコードクローン検出を行う．
4. 検出結果の評価を行う．

まず，調査対象となるオープンソースソフトウェアである，Git，Linux kernel，PostgreSQL の git のリポジトリを取得する．続いて，評価セットのバグを含むコード片と，そのコード片と同じバグを持つコードクローンがあるコミットをチェックアウトする．そして，関数クローン検出ツール，CCFinder 各々でクローン検出を行う．その際，Git と PostgreSQL はプロジェクト全体のソースコードに対してクローン検出を行ったが，Linux kernel はプロジェクトが大きすぎて，プロジェクト全体に対してコードクローン検出を行うことができなかったため，コードクローンが存在するディレクトリの周辺に範囲を絞ってコードクローン検出を行った．CCFinder と関数クローン検出ツール間で，適用範囲は統一している．

関数クローン検出ツールでの検出の方法としては，まず，buggy.c というファイルを作り，その中にバグを含む関数のコード片を抜き取り，buggy.c に保存をする．次に，その buggy.c を，バグを含むコード片と同じバグを持つコードクローンが含まれているバージョンの中に保存をする．そして，そのバージョンに対して，関数クローン検出ツールをかけて buggy.c に保存した関数と，コードクローンの例が検出されるかどうかを調べた．buggy.c を用意したのは，バグのあるコード片を含む関数のある場所をわかりやすくして，検出結果を見るときに特定しやすくするためである．関数クローン検出ツールの閾値はデフォルトの 0.9 と 0.5 の 2 種類で調査を行った．閾値をデフォルトだけでなく 0.5 でも検出を行ったのは，閾値を下げることにより，検出できる数が増えるのではないかという仮定があるからである．

CCFinder は Libra を用いた [9]．Libra は特定のコード片を引数として与えると，そのコードクローンを見つけるツールである．Libra に，引数としてバグを含むコード片を与えて，バグを含むコード片と同じバグを持つコードクローンの例があるバージョンに対してコードクローン検出を行い，コードクローンの例を検出することができるかどうかを調べた．CCFinder の最小一致トークン数は，Li らの実験に合わせて 10 として用いた．

そして，これらの検出を　38 個のバグを含むコード片の例すべてに対して繰り返し行いデータを集め，次に説明する N1〜N4 の評価指標と，再現率と適合率の評価指標を用いて評価を行った．

3.1.3　評価指標

評価の指標として 2 種類を用いた．1 つ目は Li らの研究で用いられていたものと同じ false positive と false neative に基づく指標，2 つ目は再現率と適合率と F 値いう指標を用いた．

false positive と false negative

false positive とは，バグを含むコード片に対して，同じバグを持つコードクローンの事例として挙げられていないが，評価対象のツールではコードクローンとして検出したものを指す．falise negative とは，バグを含むコード片に対して，同じバグを持つコードクローンの事例として挙げられているが，評価対象のツールではコードクローンとして検出できなかったものを指す．この 2 つを用いて評価対象のツールによる検出結果を以下の 4 つに分類する．

N1 : no false positives, no false negatives
バグを含むコード片に対して，同じバグを持つ事例として挙げられているコードクローンを全て検出し，事例として挙げられていないコードクローンは 1 つも検出していない場合これに分類される．

N2 : no false positives, some false negatives
バグを含むコード片に対して，同じバグを持つ事例として挙げられているコードクローンの中に検出漏れがあり，事例として挙げられていないコードクローンは 1 つも検出していない場合これに分類される．

N3 : some false positives, no false negatives
バグを含むコード片に対して，同じバグを持つ事例として挙げられているコードクローンを全て検出し，さらに事例として挙げられていないコードクローンもいくつか検出した場合これに分類される．

N4 : some false positives, some false negatives
バグを含むコード片に対して，同じバグを持つ事例として挙げられているコードクローンの中に検出漏れがあり，事例として挙げられていないコードクローンをいくつか検出した場合これに分類される．

再現率と適合率と F 値

false positive と false negative による評価指標では，false positive と false negative の数量の大小を考慮していない．そこで，false positive と false negative の数量をはっきりとさせるために再現率と適合率と F 値いう評価指標を用いた．再現率と適合率と F 値の説明を以下に示す．

再現率

再現率とは，正解のうち検出されたものを指すものであり，網羅性に関する指標として用いられる．ここでは，評価セットで，バグを含むコード片に対して，同じバグを持つ事例として挙げられているコードクローンのうち検出ツールが検出したものの割合を表し，以下の式で得られる．なお，再現率が高いほどそのツールは性能が良いと判断できる．

$$\text{再現率} = \frac{\text{事例として挙げられているコードクローンのうち検出ツールが検出できた数}}{\text{事例として挙げられているコードクローンの数}}$$

適合率

適合率とは，結果において本当に正しかったものの割合を指すものであり，正確性に関する指標として用いられる．ここでは，検出ツールが実際に検出したコードクローンのうち，評価セットで，バグを含む事例として挙げられているコードクローンの割合を表し，以下の式で得られる．なお，適合率が高いほどそのツールは性能が良いと判断できる．

$$\text{適合率} = \frac{\text{検出ツールが検出したコードクローンのうち事例として挙げられている数}}{\text{検出ツールが検出したコードクローンの数}}$$

F 値

F 値とは，再現率と適合率という網羅性と正確性の総合的な評価の際に利用される尺度として用いられるものである．再現率と適合率は互いにトレードオフの関係であるため，F 値を高くできれば総合的に良い評価となる．F 値は以下の式のように，再現率と適合率の調和平均によって求められる．

$$F\text{ 値} = \frac{2 \cdot \text{再現率} \cdot \text{適合率}}{\text{再現率} + \text{適合率}}$$

3.2 Late Propagation に基づく評価

Late Propagation の検出による評価の調査手法を説明する．まず，関数クローン検出ツールを使って Late Propagation を検出する手法を述べて，続いて，評価方法を説明する．

3.2.1 検出手法

関数クローン検出ツールを使って Late Propagation を検出する手法を説明する．まず，対象となるプロジェクトの git リポジトリから対象の期間のリビジョンのソースコードを取得する．次に，1 番最初のリビジョンから 2 リビジョン取り，1 リビジョンずつずらしながら，変更管理システムに入力として与えてすべてのリビジョンに対してクローン変更の検出を行い，その通知結果を保存する．この結果を分析することにより Late Propagation を検出する．

Late Propagation の検出は，クローンペアの一貫性の破壊の検出と，一貫性の回復の検出の 2 つに分けることができる．一貫性の破壊は，2 リビジョン間で古いリビジョンではクローンペアであったが，新しいリビジョンではクローンペアでは無くなった場合に起こる．よって，2.2 節で説明した Changed Clone Set と Deleted Clone Set に着目する．その中で，古いリビジョンのコードクローンの項目にある Deleted Clone と Moved Clone が一貫性が破壊されたクローンペアの片方の候補になる．そして，そのペアの候補には Stable Clone と Modified Clone が挙げられ，また，古いリビジョンの Deleted Clone と Moved Clone 同士も一貫性が破壊されたクローンペアとなる．それらのクローンペアをリビジョン，ファイル名，メソッド名の情報とともにリストに保存しておく．

一貫性の回復は，2 バージョン間で古いリビジョンではクローンペアでなかったが，新しいリビジョンではクローンペアとなった場合に起こりうる．よって，2.2 節で説明した New Clone Set と Changed Clone Set に着目する．その中で，古いリビジョンのコードクローンの項目の中のものではない Moved Clone と，Added Clone が一貫性が回復したクローンペアの片方の候補になる．そして，そのペアの候補には Deleted Clone 以外のものが挙げられる．それらのペアが一貫性の破壊の検出で保存したリストの中にあれば，それらのペアは一貫性の回復が行われたことになる．

関数クローン検出ツールは，ソースコード全体のコード片の特徴ベクトルに対してクラスタリングを行い関数間の類似度を求める．よって，関数内に変更が加えられなくても，ソースコードの他の関数に変更が加えられると，関数クローン検出ツールの検出結果が変わる可能性がある．これを Late Propagation として検出することを防ぐために，一貫性を破壊する編集と一貫性を回復する編集が行われた可能性があるクローンペアの候補に対して，2 リビジョン間で関数間に差分を取ることによりフィルタリングをかけて，クローンペア両方に対して差分がなかった場合には，候補から取り除くようにした．

一貫性の破壊と一貫性の回復の検出を対象プロジェクトの取得した全リビジョンに対して行うことにより，Late Propagation を検出する．

3.2.2 評価方法

この評価ではオープンソースソフトウェアである Apache Ant の git リポジトリの 2000 年 1 月から，2015 年 12 月までの全 13204 リビジョンのソースコードを利用して，3.2.1 節で述べた方法で Late Propagation を検出し，Late Propagation をどれくらい検出することができるかどうかを調べた．続いて，実際に検出結果の例を確認して Late Propagation の検出量や分類を確認した．

4　調査結果

4.1　バグを含むコードクローン検出における評価

評価セットの全 38 種類のクローンセットに含まれる，バグを含むコード片と同じバグ事例を持つ全 58 個のコードクローンに対する，関数クローン検出ツールと，CCFinder の検出結果を表 4 に示す．表 4 では，評価セットで与えられている，バグを含むコード片と同じバグ事例を持つコードクローンの数を“例の数”として示しており，それぞれのツールがバグを含むコード片に対して検出したコードクローンの数を“検出数”として示し，それぞれのツールが検出したコードクローンのうち，バグを含むコード片と同じバグを持つコードクローンの数を“正解検出数”として示している．また，この結果からそれぞれのツールの，3.1.3 節で説明した N1〜N4 の数をまとめたものを表 5 に示す．

N1〜N4 の特性をふまえてこの結果からわかることは，まず，N1 と N3 に分類されたものは，検出漏れがなかったということである．よって，評価セットの 38 種類のクローンセットの例の内，関数クローン検出ツールの閾値 0.9 の場合は 15 個，閾値 0.5 の場合は 21 個，CCFinder は 26 個の例を漏れなく検出することができたということになる．また，N2 と N4 に分類されたものは，評価セットのクローンセットに含まれる，バグ事例を持つコードクローンを少なくとも 1 つ以上検出できなかったということがわかる．よって，関数クローン検出ツールの閾値 0.9 の場合は 23 個，閾値 0.5 の場合は 17 個，CCFinder は 12 個のクローンセットのコードクローンを少なくとも検出できなかったということになる．そして，N1 と N2 に分類されたものは，バグ事例を持つコードクローン以外にはコードクローンを検出しておらず，N3 と N4 に分類されたものは，バグ事例を持つコードクローン以外にもいくつかコードクローンを検出していることがわかる．よって，関数クローン検出ツールの閾値 0.9 の場合は 5 個，閾値 0.5 の場合は 15 個，CCFinder は 17 個の例で，バグ事例として挙げられているコードクローン以外のコードクローンを検出したことになる．この結果をまとめると，関数クローン検出ツールに関しては，閾値 0.9 より閾値 0.5 の場合のほうが，バグ事例を持つコードクローンを多く検出することができ検出漏れは少なくなるが，バグ事例を持たないコードクローンも多く検出したことがわかる．また，CCFinder と比べると，関数クローン検出ツールの閾値に関係なく CCFinder のほうがバグ事例を持つコードクローンを多く検出することができるが，バグ事例を持たないコードクローンも多く検出したことがわかる．

N1〜N4 の評価では，それぞれの例に対して false positive と false negative の数量を考慮できていない．同じ例のクローンセットの中にも複数コードクローンがある場合もあり，どちらか片方だけを検出できるということもある．よって，このあいまいさを解消するために，false positive の数量と false negative の数量も考慮に入れて，それぞれのツールで再現

率，適合率，F 値の指標を取り入れた結果を表 6 に示す．結果を見ると，関数クローン検出ツールは閾値を 0.9 から 0.5 に下げると再現率が上がり，適合率が大幅に下がっている．また，CCFinder と比べると，関数クローン検出ツールの閾値が 0.9 の場合は，CCFinder に比べて再現率が低くなり，適合率は大幅に高くなっており，閾値が 0.5 の場合は似たような結果となっている．F 値を比べると，関数クローン検出ツールの閾値 0.9 の場合が 1 番良い値になり，閾値 0.5 の場合と CCFinder は低い値となった．以上のように，関数クローン検出ツールは，多くの研究で用いられている CCFinder に比べても良い結果を示したため，バグの同時修正において十分な性能を発揮したと判断できる．

表 4: 38 種類のクローンセットに対する検出結果

		関数クローン検出ツール				CCFinder	
		閾値 0.9		閾値 0.5			
ID	例の数	検出数	正解検出数	検出数	正解検出数	検出数	正解検出数
1	1	1	1	1	1	1	1
3	1	0	0	0	0	0	0
5	1	1	1	1	1	1	1
6	13	11	6	14	7	0	0
8	1	1	1	9	1	0	0
9	1	2	1	2	1	27	1
10	3	3	3	3	3	73	3
11	1	0	0	0	0	2	1
12	1	0	0	0	0	0	0
14	3	0	0	1	0	3	3
15	1	1	1	1	1	3	1
16	1	1	1	1	1	1	1
18	2	0	0	0	0	2	2
19	1	0	0	2	1	0	0
22	1	0	0	0	0	3	1
23	1	0	0	2	1	0	0
25	1	0	0	2	0	3	1
27	1	0	0	26	1	30	1
28	2	0	0	0	0	0	0
33	1	0	0	0	0	1	1
35	1	0	0	1	1	7	1
36	2	5	2	46	2	0	0
37	1	0	0	0	0	0	0
38	1	0	0	0	0	2	1
39	1	3	1	9	1	13	1
41	1	1	1	4	1	29	1
42	1	0	0	0	0	218	1
43	1	0	0	0	0	2	1
44	1	0	0	1	1	0	0
45	1	0	0	1	1	1	1
46	1	1	1	1	1	0	0
47	1	1	1	1	1	1	1
48	1	0	0	0	0	15	1
49	1	0	0	5	0	1	1
50	1	7	1	20	1	1	1
51	1	1	1	6	1	2	1
52	1	1	1	133	1	6	1
53	2	0	0	0	0	1	0
計	58	41	24	293	31	449	31

表 5: N1〜N4 の分類結果

	関数クローン検出ツール		CCFinder
分類	閾値 0.9	閾値 0.5	
N1	11	10	10
N2	22	13	11
N3	4	11	16
N4	1	4	1

表 6: 再現率と適合率と F 値

	関数クローン検出ツール		CCFinder
	閾値 0.9	閾値 0.5	
正解クローン数	58		
検出数	41	293	449
正解検出数	24	31	31
再現率	0.41	0.53	0.53
適合率	0.59	0.11	0.07
F 値	0.48	0.18	0.12

4.2 Late Propagation に基づく評価

関数クローン検出ツールで Late Propagation の検出を行うと，図 5 に示すような形式で検出結果が出力される．この結果から late propagation の起こったクローンペアのファイルの位置とメソッド名，一貫性が破壊されたバージョン，一貫性が回復したバージョンを知ることができる．

関数クローン検出ツールは結果として，2000 年 1 月から 2015 年 12 月までの Apache Ant プロジェクトから 12 個の Late Propagation を検出できた．検出された全 12 個の Late Propagation の分類表を表 7 にまとめた．この分類表から，検出できた Late Propagation のタイプの個数を知ることができる．関数クローン検出ツールは Apache Ant プロジェクトから，4 種類の Late Propagation を検出することができた．また，バグを含む危険度の高い LP8 を多く検出できた．検出された Late Propagation の例を図 6 に示す．最初，ソースコード片 A と B はタイプ 3 のクローンペアである．しかし，ソースコード片 A に文が追加される一貫性を破壊する編集が行われることにより，一度クローンペアではなくなる．その後，ソースコード片 A で一貫性を回復する編集が行われることによりタイプ 2 のクローンペアとなっている．このように，関数クローン検出ツールを用いて Late Propagation を検出することができた．

表 7: 検出した Late Propagation の分類結果

タイプ	検出個数
LP1	0
LP2	4
LP3	1
LP4	0
LP5	1
LP6	0
LP7	0
LP8	6

クローンID:1
一貫性が破壊されたバージョン:[2584]
一貫性が回復したバージョン:[2685]
クローンペア
src¥main¥org¥apache¥tools¥ant¥taskdefs¥Deltree.java:Deltree.execute()
src¥main¥org¥apache¥tools¥ant¥taskdefs¥Mkdir.java:Mkdir.execute()

クローンID:2
一貫性が破壊されたバージョン:[4053]
一貫性が回復したバージョン:[4415]
クローンペア
src¥main¥org¥apache¥tools¥ant¥types¥optional¥depend¥DependScanner.java:DependScanner.getIncludedFiles()
src¥main¥org¥apache¥tools¥ant¥DirectoryScanner.java:DirectoryScanner.getIncludedFiles()

クローンID:3
一貫性が破壊されたバージョン:[4053]
一貫性が回復したバージョン:[4415]
クローンペア
src¥main¥org¥apache¥tools¥ant¥types¥optional¥depend¥DependScanner.java:DependScanner.getIncludedFiles()
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥ide¥VAJWorkspaceScanner.java:VAJWorkspaceScanner.addDefaultExcludes()

クローンID:4
一貫性が破壊されたバージョン:[3210]
一貫性が回復したバージョン:[5387]
クローンペア
src¥main¥org¥apache¥tools¥ant¥util¥regexp¥RegexpFactory.java:RegexpFactory.newRegexp(Project p)
src¥main¥org¥apache¥tools¥ant¥util¥regexp¥RegexpMatcherFactory.java:RegexpMatcherFactory.newRegexpMatcher(Project p)

クローンID:5
一貫性が破壊されたバージョン:[5153]
一貫性が回復したバージョン:[7072]
クローンペア
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥clearcase¥CCCheckin.java:CCCheckin.execute()
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥ccm¥CCMReconfigure.java:CCMReconfigure.execute()

クローンID:6
一貫性が破壊されたバージョン:[5153]
一貫性が回復したバージョン:[7072]
クローンペア
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥clearcase¥CCUnCheckout.java:CCUnCheckout.execute()
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥ccm¥CCMReconfigure.java:CCMReconfigure.execute()

クローンID:7
一貫性が破壊されたバージョン:[5153]
一貫性が回復したバージョン:[7072]
クローンペア
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥clearcase¥CCUpdate.java:CCUpdate.execute()
src¥main¥org¥apache¥tools¥ant¥taskdefs¥optional¥ccm¥CCMReconfigure.java:CCMReconfigure.execute()

図 5: Late Propagation 検出結果の例

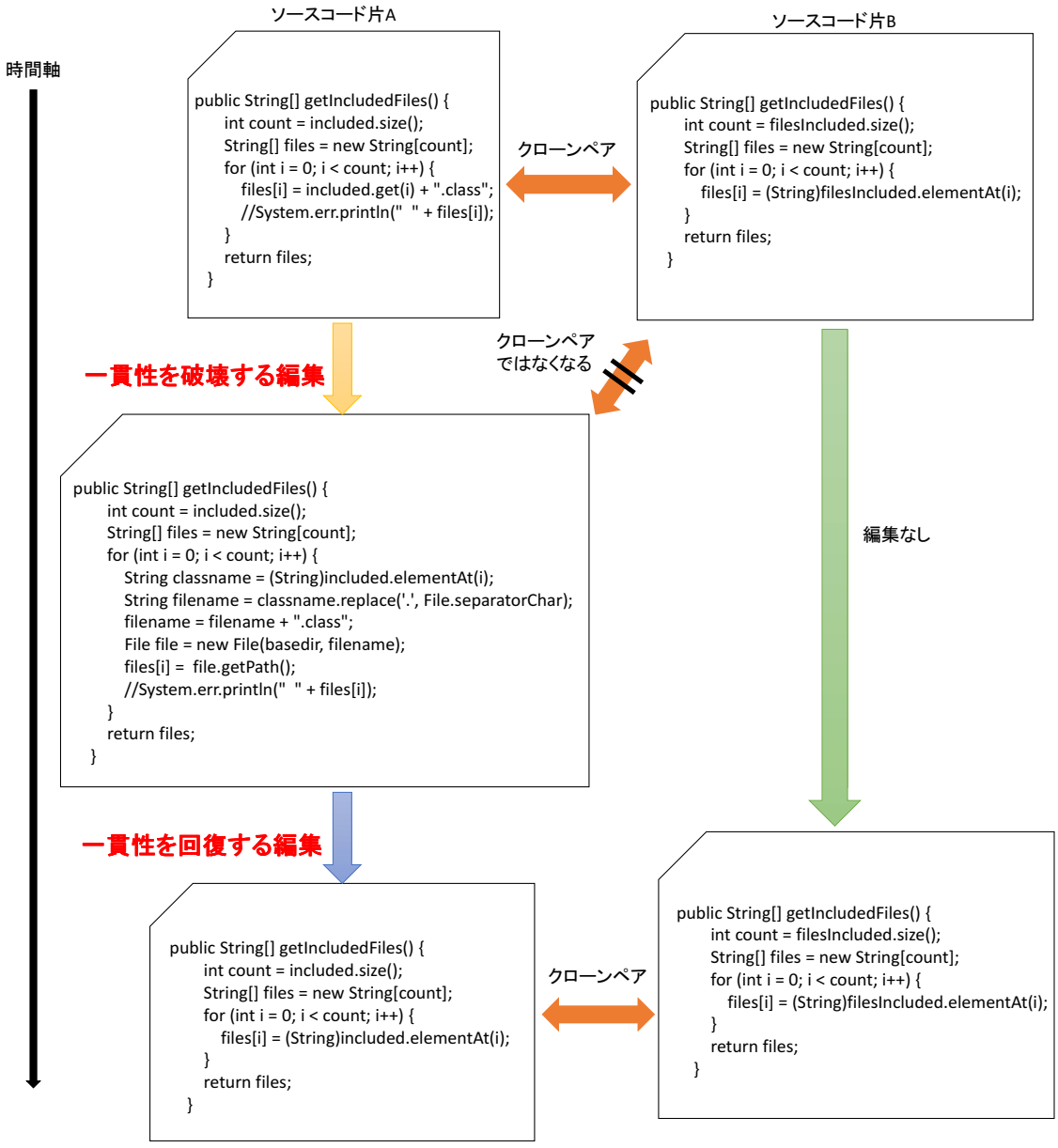

図 6: 検出した Late Propagation の例

5　考察

5.1　バグを含むコードクローン検出における評価

関数クローン検出ツールの検出結果は，閾値によって大きく変わった．閾値を下げると，再現率は上がり適合率は下がる．これら 2 つはトレードオフの関係にある．よって，適切な閾値を定める必要がある．閾値を 0.6，0.7，0.8 の場合も調べると，よりよい閾値が見つかるかもしれない．

本研究では，CCFinder の最小一致トークン数を Li らの調査に合わせて 10 に設定した．開発者がバグ事例ごとに，最小一致トークン数を適切に設定することができれば，CCFinder の適合率と F 値は上昇することが考えられる．しかし，コードクローン分析に詳しい開発者でなければ，最小一致トークン数を適切に設定することは難しいと考えられる．よって，今後の課題として，開発者に最小一致トークン数をバグ事例ごとに設定してもらいながら CCFinder を利用してもらう評価も必要であると考えられる．

検出結果から，関数クローン検出ツールと CCFinder の長所，短所を表 8 にまとめた．

表 8: 関数クローン検出ツールと CCfinder の特徴まとめ

	関数クローン検出ツール	CCFinder
長所	・タイプ 3，4 のコードクローンまで検出することができる ・＜，＞等演算子が違うクローンでも検出することができる ・適合率が比較的高めである	・関数単位で類似していないコードクローンでも検出することができる
短所	・関数単位ではなくその中のごく 1 部が類似しているようなコードクローンを検出することはできない	・＜，＞等演算子が違うコードクローンを検出することができない ・タイプ 1，2 のコードクローンしか検出できない ・短い単純な構文のコードクローンだと適合率が下がってしまう

上記の表記から，両ツールに長所と短所があるため，これら 2 つのツールを上手く使い分ければ，バグの同時修正を有効的に行えることがわかる．CCFinder はタイプ 1，2 のコードクローンしか検出できないが，関数クローン検出ツールはタイプ 3，4 のコードクローンまで検出することができる．しかし，関数クローン検出ツールでは，関数単位で類似しておらず，関数のごく一部が類似しているようなコードクローンを検出することができない．そ

こで，両ツールを組み合わせて利用することを考えてみた．両ツールの和集合を取った場合の，再現率と適合率と F 値を表 9 にまとめた．

表 9: 関数クローン検出ツールと CCFinder の和集合の再現率と適合率と F 値

	CCFinder と 関数クローン検出ツール (閾値 0.9)	CCFinder と 関数クローン検出ツール (閾値 0.5)
再現率	0.71	0.76
適合率	0.11	0.08
F 値	0.19	0.14

CCFinder と関数クローン検出ツールを組み合わせて利用すると，表 6 にまとめている両ツールを単体で利用したときに比べて再現率が大幅に上がる．適合率と F 値に関しては，関数クローン検出ツール単体の閾値 0.9 の時に比べると大幅に下がってしまうが，閾値 0.5 の時と CCFinder 単体のときとはほとんど変わらない．よって，検出する際のコストが許されるのであれば，関数クローン検出ツールと CCFinder の両者を併用すれば，バグを含むコード片に対して同じバグを含むコードクローンをより多く見つけることができると考える．

5.2 Late Propagation に基づく評価

関数クローン検出ツールの Late Propagation の検出結果の中に，クローンペアに対して一貫した編集が行われているのにも関わらず，一貫性を破壊する編集と一貫性を回復する編集を検出して，Late Propagation に分類された例が 1 つあった．3.2.1 節で述べた通り，関数クローン検出ツールは，ソースコード全体の特徴ベクトルに対してクラスタリングを行い，関数間の類似度を求めることによってコードクローンを検出する．そのため，クローンペアに対して，一貫した編集を行ったとしても，コードクローン以外の部分に大幅に変更が加えられると，クローンペアとして検出されない可能性がある．よって，このような例には注意をする必要がある．

今回は Apache Ant プロジェクトに対して Late Propagation の検出を行い，12 個の Late Propagation を検出することができた．しかし，1 つのプロジェクトに対してしか適用ができていないため，他のプロジェクトに対しても適用する必要がある．また，本手法では Late Propagation をファイル名と関数名を保存することにより検出しているため，関数名が変わるコードクローンに対しては Late Propagation を検出することができない．そのようなコードクローンに対しても Late Propagation を検出する必要がある．

6　まとめと今後の課題

本研究では，関数クローン検出ツールについて，バグを含むコードクローン検出における評価と，Late Propagation の検出による評価の 2 つの評価を行った．バグを含むコードクローン検出における評価では，関数クローン検出ツールと CCFinder を使って，Li らが用意した評価セット [14] に対してコードクローン検出を行った．その結果，関数クローン検出は閾値がデフォルトの 0.9 の場合には，再現率でやや CCFinder に劣ったが，適合率と F 値は大きく上回った．閾値が 0.5 の場合には，再現率，適合率，F 値ともに CCFinder と同じくらいの値になった．よって，バグの同時修正において関数クローン検出ツールの有効性を確認できた．Late Propagation の検出による評価では，関数クローン検出ツールを用いて，Late Propagation を検出した．結果としては，12 個の Late Propagation を検出することができ，一貫性を破壊する編集を検出することができた．2 つの評価から，関数クローン検出ツールはソースコードの同時修正支援において有効であると判断した．

今後の課題としては，まず，関数クローン検出ツールの適切な閾値を求めることが考えられる．本研究で利用した評価セットに対して，関数クローン検出ツールの閾値を 0.6，0.7，0.8 のそれぞれに設定した場合についても検出を行って，その結果の再現率と適合率と F 値を求めて比べる予定である．そして，CCFinder の最小一致トークン数についても適切な値を考える必要があると考えられる．本研究では，Li らの調査に合わせて最小一致トークン数を 10 に設定した．開発者が，バグ事例ごとに適切な最小一致トークン数を設定することができれば，適合率と F 値は上昇すると考えられるため，開発者に最小一致トークン数をバグ事例ごとに設定してもらいながら CCFinder を利用してもらう評価も必要であると考えられる．続いて，手法の改善が挙げられる．Late Propagation の検出において，現段階ではファイル名と関数名を軸において検出を行っており，関数名が変わるコードクローンに対しては Late Propagation を検出することができない．よって，そのようなコードクローンに対しても Late Propagation を検出することができるようにすることを目指す．また，CCFinder のような関数クローン検出ツール以外のコードクローン検出ツールを用いて Late Propagation を検出するツールを作成し，Late Propagation 検出の性能を比べることも考えられる．そして，評価対象についても Late Propagation の検出には，本論文では Apache Ant プロジェクトにしか適用していないため，他のプロジェクトに対しても適用してみて一般性を確かめることも必要であると考える．

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 井上 克郎 教授には，研究に関する適切な御指導及び御助言を賜りました．井上教授の御指導及び御助言のおかげで本論文を完成させることができました．井上 教授に心より深く感謝いたします．

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，研究の各段階において多くの御助言を賜りました．多くの御指導及び御助言を頂いた 松下 准教授に心より深く感謝いたします．

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 石尾 隆 助教には，研究においてたくさんの貴重な御意見を賜りました．多くの御助言を頂いた 石尾 助教に心より深く感謝いたします．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田 則裕 准教授には，研究に関する直接の御指導を賜りました．常に適切な御指導及び御助言を頂いたことにより，本論文を完成することができました．吉田 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名修介 特任教授には，常に適切な御指導及び御助言を賜りました．多くの御助言を頂いた 春名 特任教授に心より深く感謝いたします．

大阪大学大学院国際公共政策研究科 崔 恩瀞 助教には，研究に関する多くの貴重な御助言を賜りました．多くの御助言を頂いたおかげで本論文を完成させることができました．崔 恩瀞 助教には心より深く感謝いたします．

日本電気株式会社 鈴木 明彦 氏，前田 直人 氏には，研究に関して企業のソフトウェア開発者の観点から多くの御意見を頂きました．心より深く感謝いたします．

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 佐野 真夢 氏，雑賀 翼 氏，中村 勇太 氏には，研究に関する相談に乗って頂き，また，本論文の修正を行って頂くなど研究の様々な場面でご協力していただきました．有意義な研究室生活を送りながら本論文を完成させることができたことのは御三方のおかげであると，心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった，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井上研究室の皆様に，心より深く感謝いたします．

参考文献

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceeding of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pp. 459–468, 2006.

[2] R. Baeza-Yates and B Ribeiro-Neto. *Modern information retrieval: The concepts and technology behind search.* Addison-Wesley, 2011.

[3] 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.

[4] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pp. 368–377, 1998.

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

[6] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 9, No. 3, pp. 319–349, 1987.

[7] 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, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[8] 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. 43–56, 2011.

[9] Yoshiki Higo, Yasushi Ueda, Shinji Kusumoto, and Katsuro Inoue. Simultaneous modification support based on code clone analysis. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pp. 262–269, 2007.

[10] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[11] 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. 47–54, 2001.

[12] 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.

[13] 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.

[14] Jingyue Li and Michael D Ernst. CBCD:Cloned buggy code detector. Technical report, UW-CSE-11-05-02, 2011.

[15] Jingyue Li and Michael D Ernst. CBCD: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 310–320, 2012.

[16] 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.

[17] 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.

[18] 佐野真夢, 吉田則裕, 春名修介, 井上克郎. 情報検索技術に基づく関数クローン検出を用いた変更管理システムの開発. 情報処理学会研究報告, Vol. 2015-SE-190, No. 4, pp. 1–8, 2015.

[19] Ewan Tempero. Towards a curated collection of code clones. In *Proceedings of the 7th International Workshop on Software Clones*, pp. 53–59, 2013.

[20] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹. コードクローン変更管理システムの開発と実プロジェクトへの適用. 情報処理学会論文誌, Vol. 54, No. 2, pp. 883–893, feb 2013.

[21] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.