社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソースコード中の繰り返し部分に着目したコードクローン検出手法の提案

村上　寛明† 　堀田　圭佑† 　肥後　芳樹† 　井垣　　宏† 　楠本　真二†

† 大阪大学大学院情報科学研究科

あらまし　コードクローンとはソースコード中の一致，または類似したコード片の組を表す．コードクローンはソフトウェアの保守を困難にする要因の 1 つであり，これまでに様々な検出手法が提案されている．しかし，既存の検出手法では，ソースコード中の繰り返し部分で冗長なコードクローンを検出してしまう問題がある．本論文では，繰り返し部分を折りたたむという前処理を行うことで，冗長なコードクローンの検出を抑制する手法を提案する．また，提案手法を組み込んだ検出ツールを実装し，オープンソースソフトウェアに対して評価実験を行う．

キーワード　コードクローン，プログラム解析，ソフトウェア保守

Code Clone Detection Method Designed for Information of Repetition in Source Code

Hiroaki MURAKAMI†, Keisuke HOTTA†,

Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University

Abstract　A code clone is a code fragment that is identical or similar to another in a source code. One of the factors that makes software maintenance more difficult is the presence of code clones and many detection methods have been proposed. However, existing methods have a problem that they detect many unnecessary code clones for software maintenance. In this paper, we propose a preprocessing, which folds repeated instructions in source code, for code clone detection. Moreover, we implement a code clone detection tool including the preprocessing, and apply it to open source software systems.

Key words　Cord Clone, Program Analysis, Software Maintenance

1. はじめに

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェアの保守に要するコストが増加している．ソフトウェアの保守を困難にさせる要因の 1 つとしてコードクローンへの関心が高まっており，これまでにコードクローンに関する様々な研究が盛んに行われている [1].

しかし，既存の検出手法には，ソースコード中の繰り返し部分で多くのコードクローンを見つけてしまうという問題がある．繰り返し部分の例として，switch 文の各 case エントリや連続した変数宣言，及び連続した同一関数の呼び出しなどが挙げられる．図 1 は冗長なコードクローン検出の例を表している．図 1(a) と図 1(b) の switch 文は，それぞれ 5 つと 3 つの case エントリを含んでいる．これら 8 つの case エントリは，リテラルが違うのみの繰り返し構造となっている．この switch 文に対して既存の検出ツールを適用すると 6 つものコードクローンのペアを検出してしまう．しかし，このソースコードにおいて，ユーザが必要とする情報は，図 1(a) と図 1(b) の switch 文は共に StringBuffer を用いた文字列の追加処理を行なっているということであろう．よって，switch 文全体をコードクローンのペアとして検出することが望ましい．

そこで本研究では，より有益なコードクローン検出結果を得るために，ソースコード上の繰り返し構造を折りたたむという検出の前処理を提案する．繰り返し部分を折りたたむことで，着目する必要のない冗長なコードクローンの検出を抑止するとともに，把握すべきコードクローンが見つかるようになる．

そして，提案手法を用いたコードクローン検出ツールを作成し，複数のオープンソースソフトウェアに対して検出を行った．その結果，折りたたみ無しの場合に比べて，折りたたみ有りの場合の検出数が減少したという結果を得た．特に繰り返し部分

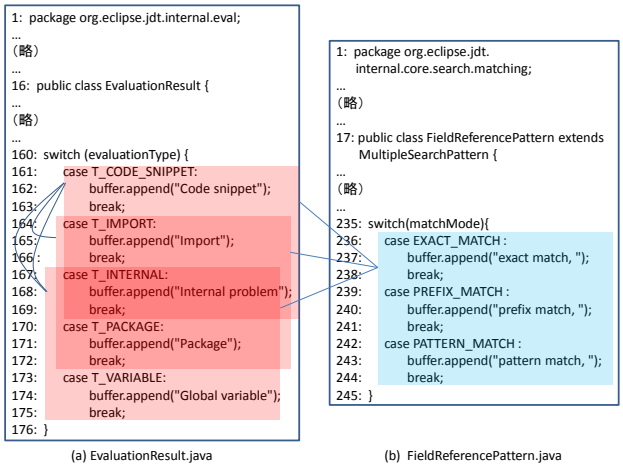


図 1　繰り返し部分における既存手法の検出例

を多く含むソフトウェアに対しては効果が大きく，検出数が半分以下になるものも確認できた．

2. 提 案 手 法

この節では，本研究の提案手法である繰り返し部分の折りたたみについて，1. で挙げた例を元に説明する．この例は switch 文の各 case エントリが繰り返されている．この繰り返し部分を折りたたむと図 2 のように 2 つの switch 文は 1 つの case エントリのみをもつ構造になる．繰り返し部分の繰り返し回数を記憶し，また，ユーザ定義名を特別な同一の字句に置き換えると，単純な字句単位の解析で switch 文全体をコードクローンとして検出することができる．検出例を図 3 に示す．

折りたたみのアルゴリズムを Algorithm1 に示す．引数の str は折りたたむ文字列，repeatCount は折りたたむ文の数を表す．例えば，repeatCount が 2 であれば「文 1，文 1」と「文 1，文 2，文 1，文 2」といった繰り返しを折りたたみ，「文 1，文 2，文 3，文 1，文 2，文 3」のような繰り返しは折りたたまない．アルゴリズム中の *isSentenceEnd*() は，与えられた文字が”;”，”{”，”}”のいずれかであれば true を，そうでなければ false を返す手続きである．また，*length*() は，与えられた文字列の長さを返す手続きである．6-18 行目と 21-30 行目で隣接する 2 つの文を取得する．31-37 行目で隣接する 2 つの文が同じだったときの折りたたみの処理を行う．repeatCount の数だけ文の折りたたみを行うと，最後に 40 行目で折りたたまれた文字列を返す．

3. 実　　装

3.1 概　　要

ツールは Java を用いて実装した．現在のところ，コードクローンの検出が可能な言語は Java と C である．提案手法を組み込んだコードクローン検出は，検出対象のソースコードを入力とし，コードクローンの情報を出力とする．また，3.2 で述べる各ステップにおいて，それぞれのステップ内の処理は互いの処理内容に影響を及ぼさないので並列処理が可能である．

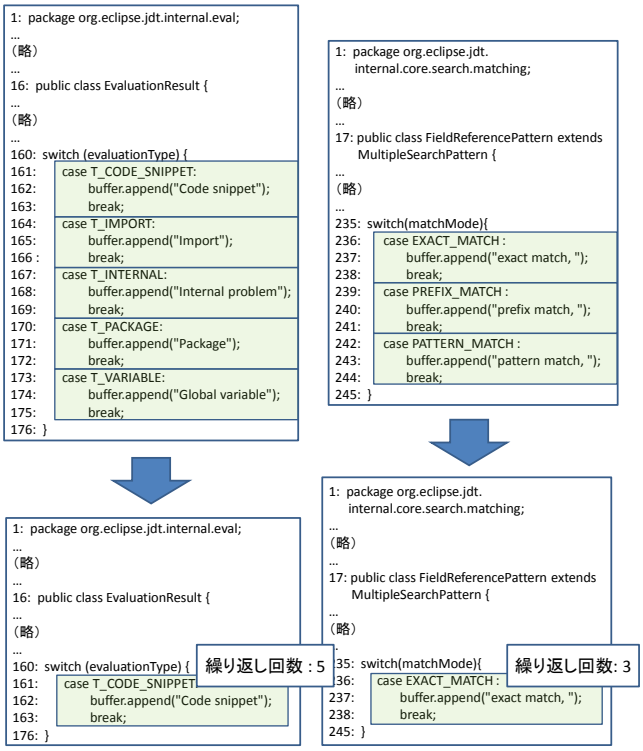


図 2　折りたたみの様子

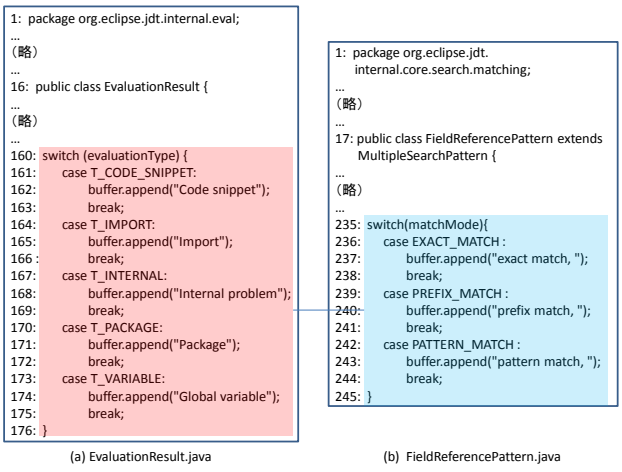


図 3　繰り返し部分における提案手法の検出例

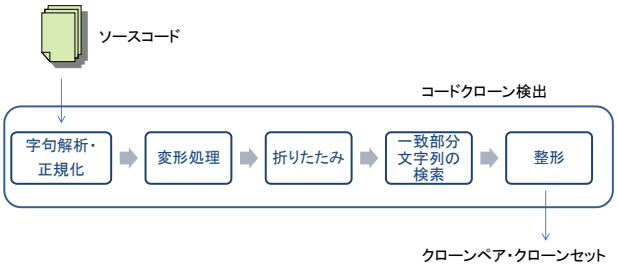


図 4　処理の流れ

実装したツールでは，使用する計算機のプロセッサの数だけのスレッドを作成し，それらを並列に処理させることで高速化を図っている．

Algorithm 1 fold Repetition

Input: $str, repeatCount(\geqq 1)$
Output: str after folded

```
1: strlen ← length(str)
2: for i = 0 to repeatCount do
3:    left ← 0
4:    loop
5:       flg ← true; index ← left; tmpleft ← left; count ← 0;
6:       while count ≦ i and index < strlen do
7:          if isSentenceEnd(str[index]) then
8:             if flg then
9:                k ← index + 1; flg ← false
10:            end if
11:            count ← count + 1
12:         end if
13:         index ← index + 1
14:      end while
15:      if index > strlen then
16:         break
17:      end if
18:      tmp ← str[left..index − 1]
19:      count ← 0
20:      left ← index
21:      while count ≦ i and index < strlen do
22:         if isSentenceEnd(str[index]) then
23:            count ← count + 1
24:         end if
25:         index ← index + 1
26:      end while
27:      if index > strlen then
28:         break
29:      end if
30:      tmp2 ← str[left..index − 1]
31:      if tmp = tmp2 then
32:         str ← str[0..left − 1] + str[index..strlen]
33:         strlen ← length(str)
34:         left ← tmpleft
35:      else
36:         left ← k
37:      end if
38:   end loop
39: end for
40: return str
```

3.2 処理の流れ

提案手法の全体の流れを図 4 に示す．コードクローン検出は以下の 5 ステップで行われる．

ステップ 1：**字句解析・正規化**

まず，字句解析によって入力ソースコードをトークン列に変換する．この際，コメントや改行記号，タブ，空白は取り除かれる．次に，ユーザ定義名を”$”に変換する．同時に，中括弧のネストを認識することで，メソッドや関数の境界を識別し，トークン列内におけるそれらの境界に”#”を埋め込む．

ステップ 2：**変形処理**

このステップでは字句解析・正規化によって得られた文字列を変形ルールに従って変形する．Java と C の変形ルールを以下に示す．

・パッケージ名を取り除く

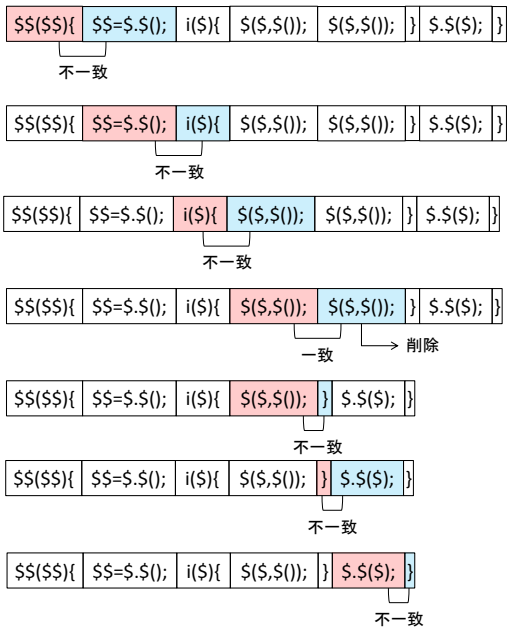


図 5　単一の文に対する折りたたみの例

PackageName + ’.’ + ClassName → ClassName

パッケージ内のクラスやインターフェースを呼び出す命令があった場合，パッケージ名を省略する．なお，この変形ルールは Java にのみ適応する．

・テーブルの初期化を取り除く

’=’ ’{’ InitalizationList ’}’ → ’=’ ’{’ ’}’

テーブルを初期化するときに要素が指定されていれば，それを取り除く

ステップ 3：**繰り返し部分の折りたたみ**

変形処理によって得られた文字列の中で繰り返している部分を取り除く．まず，”;”，”{”，”}”を「区切り文字」と定義し，区切り文字で終わる文字列を「文」と定義する．次に，変形処理で得られた文字列を文単位で区切る．隣接する文が同じであれば同じ文が繰り返されているということであり，繰り返されている後ろの文を取り除く．折りたたみの例を図 5 に示す．また，複数の文の繰り返し（「文 1，文 2，文 1，文 2」のような繰り返し）を含む文字列に対しても同様の処理を行う．

ステップ 4：**一致部分文字列の検索**

このステップでは，折りたたみによって得られた文字列の中から一致部分文字列を検索する．検索には Suffix Array [2] を用いる．

ステップ 5：**出力整形処理**

検索した 2 つの一致部分文字列の開始行と終了行をクローンペアの開始行と終了行として出力する．

4. 評価実験

4.1 目　　的

この実験の目的は，折りたたみの有無におけるコードクローン検出数と精度の変化を調べることである．

4.2 準　　備

これまでに，コードクローン検出ツールの性能評価実験がいくつか行われている [3] [4] [5]．この中でも Bellon らによる実験は最も大規模であり，今回の実験は Bellon らの実験に沿って行う．

4.2.1 正解クローン

Bellon らの実験により得られたデータ [6] を検出すべきコードクローンとした．このコードクローンの情報は下記の手順で作成された．

(1) Bellon が検出対象ソフトウェアを決め，6 人のコードクローン検出ツールの開発者に検出を依頼した．Bellon はコードクローン検出ツールの開発者ではないため，中立的な立場で検出対象ソフトウェアを選んでいると述べている [3]．

(2) 各開発者は，自身が開発した検出ツールを用いて，対象ソフトウェアからコードクローンを検出した．所定のフォーマットを用いて，検出したコードクローンの位置情報を Bellon に送付した．

(3) Bellon は各開発者から送られたコードクローンの全ペア（325,935 個）のうちの 2%を無作為に選択し，それらが本当にコードクローンであるかを手作業により判定した．ツールが検出したコードクローンがそのままコードクローンと判断される場合もあるが，Bellon が必要に応じてコードクローンを加工して 1 つのコードクローンと判断した場合もある．この結果 4 つの対象ソフトウェアから 2207 個のコードクローンのペアが抽出された．

以降，この実験では，ツールが検出したクローンペアの集合を**クローン候補**，Bellon が抽出したクローンペアの集合を**正解クローン**と呼ぶ．しかし，正解クローンの中にコードクローンとして不適切なものが見られた．例えば，複数のメソッドにまたがってコードクローンとしている例や，メソッドがないクラスのみをコードクローンとしている例である．今回の実験では，正解クローンの中からこれらのコードクローンを取り除いたものを新たな正解クローン（以降，**新正解クローン**と呼ぶ）とした．正解クローンと新正解クローンの個数を表 1 に示す．

4.2.2 評 価 基 準

クローン候補が正解クローンにどれだけ類似しているかを good 値と ok 値を用いて判断し，再現率と適合率を算出する．再現率は，新正解クローン数に対する検出した中で新正解クローンであったものの数の割合である．適合率は，検出数に対する検出した中で新正解クローンであったものの数の割合である．good 値と ok 値については文献 [3] を参照されたい．本実験では，good 値と ok 値の閾値として 0.7 を用いた．

表 1　対象ソフトウェアと正解クローン数

ソフトウェア	言語	正解クローン	新正解クローン
netbeans	Java	55	39
jdtcore	Java	1,345	1,160
weltab	C	275	261
postgresql	C	555	551

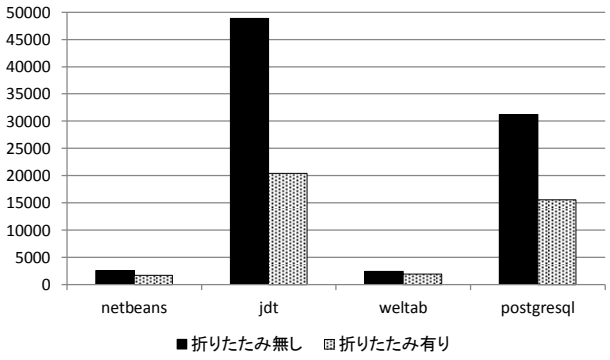
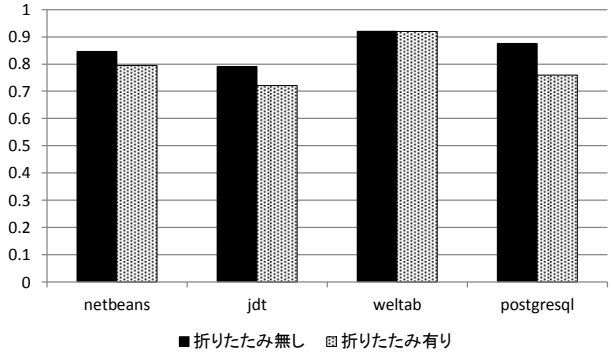
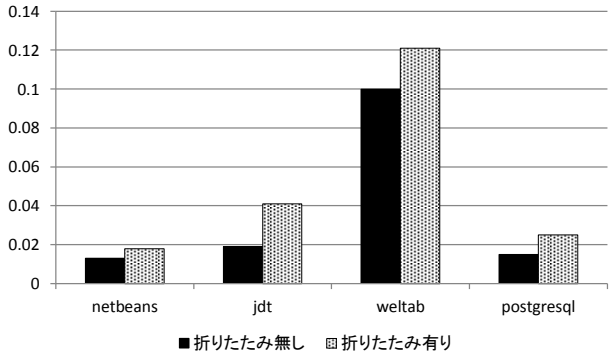


図 6　折りたたみの有無におけるコードクローンの検出数



(a) ok-recall



(b) ok-precision

図 7　折りたたみの有無における対象プログラムの ok-recall と ok-precision

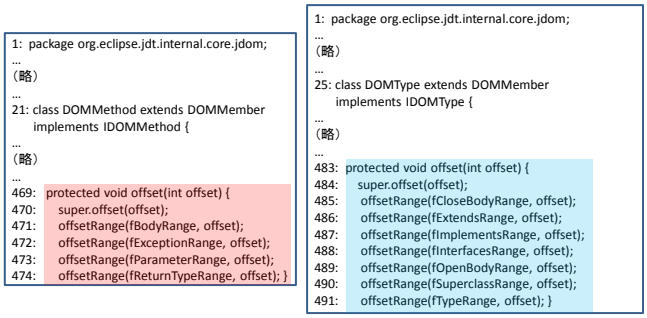
4.3 結　　果

検出数についての実験結果を図 6 に示す．図 6 に示すように，すべてのソフトウェアにおいてコードクローンの検出数が減少した．recall と precision についての実験結果を図 7 に示す．繰り返し部分を折りたたむことですべてのソフトウェアについて recall は減少し，precision は上昇した．

5. 考　　察

5.1 折りたたみを用いた検出例

提案手法による繰り返し部分の検出結果の 1 つを図 8 に示す．繰り返し部分において，各 if-else 節を別々にコードクローンと



```
1:  package org.eclipse.jdt.internal.core.jdom;
...
（略）
...
21: class DOMMethod extends DOMMember
      implements IDOMMethod {
...
（略）
...
469:  protected void offset(int offset) {
470:     super.offset(offset);
471:     offsetRange(fBodyRange, offset);
472:     offsetRange(fExceptionRange, offset);
473:     offsetRange(fParameterRange, offset);
474:     offsetRange(fReturnTypeRange, offset); }
```

(a) DOMMethod.java

```
1:  package org.eclipse.jdt.internal.core.jdom;
...
（略）
...
25: class DOMType extends DOMMember
      implements IDOMType {
...
（略）
...
483:  protected void offset(int offset) {
484:     super.offset(offset);
485:     offsetRange(fCloseBodyRange, offset);
486:     offsetRange(fExtendsRange, offset);
487:     offsetRange(fImplementsRange, offset);
488:     offsetRange(fInterfacesRange, offset);
489:     offsetRange(fOpenBodyRange, offset);
490:     offsetRange(fSuperclassRange, offset);
491:     offsetRange(fTypeRange, offset); }
```

(b) DOMType.java

図 8　繰り返し部分における提案手法の検出例

して検出することなく，if-else 節全体で検出できている．つまり，繰り返し部分をまとめて検出するのに成功し，提案手法の有効性を示すことができた．

5.2　折りたたみによる検出数の減少

jdt との比較において，検出数が 48,888 から 20,463 と半数以下に減少した．jdt のソースコードを調査したところ，jdt のソースコード中に if 文，if-else 節，switch 文内の case エントリ，try-catch 節などの繰り返しが非常に多く見られた．このような繰り返し部分に対し，提案する折りたたみ機能が効果的に働いたため，検出数の大幅な減少につながったといえる．jdt のプログラムにおける if 文の繰り返し例を図 9 に示す．また，jdt の検出結果のうち正解クローンであったものの数は 918 個から 836 個に減少した．これは，折りたたみによって繰り返し部分内の新正解クローン（図 10）を検出しなくなったためである．繰り返し部分内のコードクローンは 1. で述べたように検出する利点がないので，新正解クローンの検出数の減少は妥当である．

6.　実験結果の妥当性について

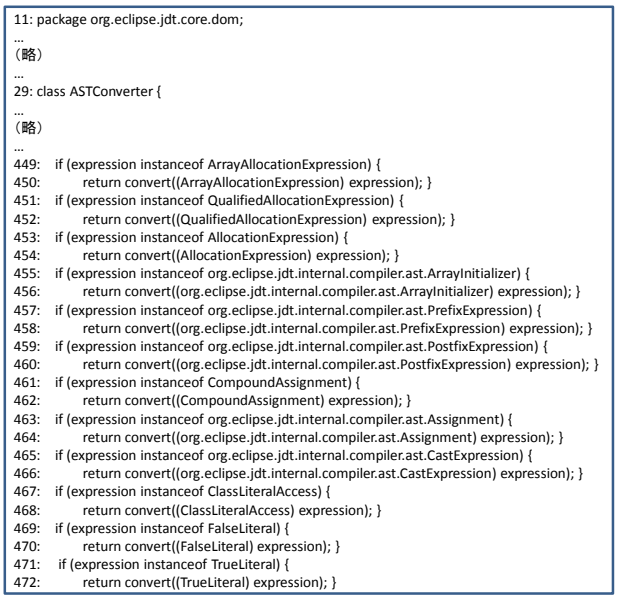
本研究の結果の妥当性に関して，以下で挙げる点に留意する必要がある．

6.1　正解クローンの作成法

本論文の実験では，Bellon らが作成した正解クローンを用いて新正解クローンを作った．そして，新正解クローンを用いて，提案手法を実装したツールの性能について調査を行った．正解クローンは対象ソフトウェアに含まれるすべてのコードクローンではなく，それを用いて作成した新正解クローンも同様である．そのため，すべてのコードクローンを対象にして同様の実験を行った場合は，異なる実験結果が得られる可能性がある．しかし，ソースコード中に存在するすべてのクローンを対象にすることは現実的ではないので，正確な recall，precision の値を求めることは難しい．recall，precision の値は相対的な評価にのみ用いることができる．

6.2　検出手法におけるコードクローンの定義の違い

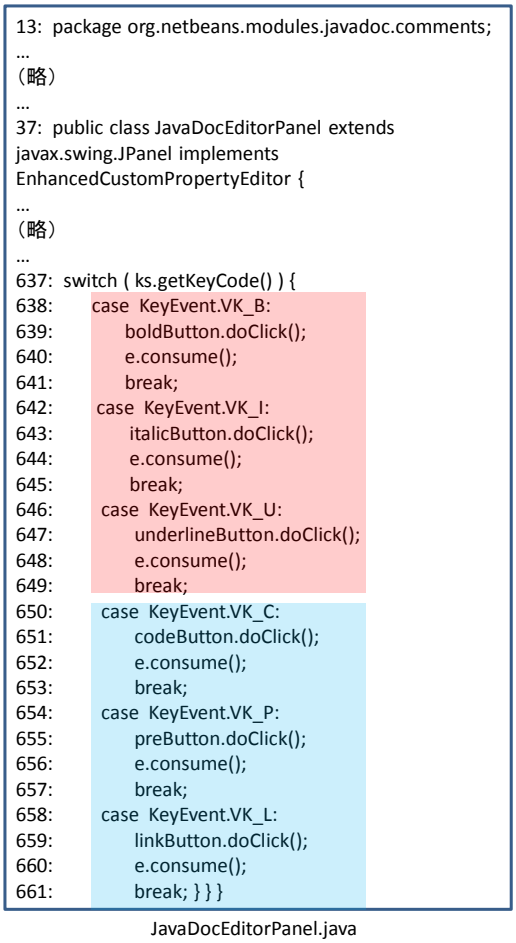
コードクローン検出ツールは字句や行単位のものだけでなく，抽象構文木やプログラム依存グラフを用いたものなど様々である．各コードクローン検出ツールによってコードクローンの定



```
11: package org.eclipse.jdt.core.dom;
...
（略）
...
29: class ASTConverter {
...
（略）
...
449:   if (expression instanceof ArrayAllocationExpression) {
450:       return convert((ArrayAllocationExpression) expression); }
451:   if (expression instanceof QualifiedAllocationExpression) {
452:       return convert((QualifiedAllocationExpression) expression); }
453:   if (expression instanceof AllocationExpression) {
454:       return convert((AllocationExpression) expression); }
455:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.ArrayInitializer) {
456:       return convert((org.eclipse.jdt.internal.compiler.ast.ArrayInitializer) expression); }
457:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.PrefixExpression) {
458:       return convert((org.eclipse.jdt.internal.compiler.ast.PrefixExpression) expression); }
459:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.PostfixExpression) {
460:       return convert((org.eclipse.jdt.internal.compiler.ast.PostfixExpression) expression); }
461:   if (expression instanceof CompoundAssignment) {
462:       return convert((CompoundAssignment) expression); }
463:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.Assignment) {
464:       return convert((org.eclipse.jdt.internal.compiler.ast.Assignment) expression); }
465:   if (expression instanceof org.eclipse.jdt.internal.compiler.ast.CastExpression) {
466:       return convert((org.eclipse.jdt.internal.compiler.ast.CastExpression) expression); }
467:   if (expression instanceof ClassLiteralAccess) {
468:       return convert((ClassLiteralAccess) expression); }
469:   if (expression instanceof FalseLiteral) {
470:       return convert((FalseLiteral) expression); }
471:    if (expression instanceof TrueLiteral) {
472:       return convert((TrueLiteral) expression); }
```

ASTConverter.java

図 9　jdt のプログラム中の if 文の繰り返し



```
13:  package org.netbeans.modules.javadoc.comments;
...
（略）
...
37:  public class JavaDocEditorPanel extends
javax.swing.JPanel implements
EnhancedCustomPropertyEditor {
...
（略）
...
637:  switch ( ks.getKeyCode() ) {
638:       case  KeyEvent.VK_B:
639:            boldButton.doClick();
640:            e.consume();
641:            break;
642:       case  KeyEvent.VK_I:
643:            italicButton.doClick();
644:            e.consume();
645:            break;
646:       case  KeyEvent.VK_U:
647:            underlineButton.doClick();
648:            e.consume();
649:            break;
650:       case  KeyEvent.VK_C:
651:            codeButton.doClick();
652:            e.consume();
653:            break;
654:       case  KeyEvent.VK_P:
655:            preButton.doClick();
656:            e.consume();
657:            break;
658:       case  KeyEvent.VK_L:
659:            linkButton.doClick();
660:            e.consume();
661:            break; } } }
```

JavaDocEditorPanel.java

図 10　繰り返し部分内のコードクローンの例

義が異なるので，他のツールを用いて比較をした場合，異なる実験結果が得られる可能性がある．

7. 関 連 研 究

肥後らは，クローンペアの集合に含まれるコード片がどの程度繰り返し要素を含まないか表すメトリクス RNR を提案している [7]．このメトリクスを使うことで関数や代入文などの羅列，switch 文のように同じ構造になりやすい文などを取り除くことができる．RNR は，CCFinder [8] の後続ツールである CCFinderX [9] でも採用されており，閾値を定めることで検出結果のフィルタリングができる．RNR は，繰り返し要素から構成されるコードクローンをある程度自動的にフィルタリングすることは可能だが，把握すべきクローンを検出することができない点においては提案手法に対して劣っている．

今回の実験では，比較に Bellon らの実験を用いた．他にもコードクローン検出手法・ツールの比較実験は行われているが，Bellon らの実験は比較ツールの数，対象プログラムの規模共に最大であることから，本研究における比較実験として Bellon らの実験方法を採用した．Burd らの実験 [4]，Rysselberghe らの実験 [5] を以下に示す．

Burd らは Kamiya らの手法 [8]，Baxter らの手法 [10]，Mayland らの手法 [11]，Prechalt らの手法 [12]，Aiken らの手法 [13] の 5 つの比較をしている．各検出手法で検出されたコードクローンを実際に見て，本当にコードクローンであったものを正解クローンとし，各検出手法の再現率と適合率を求めている．

Rysselberghe らは，行単位，字句単位，メトリクス計測の 3 つの検出技法を比較している．Rysselberghe らはツールを比較するのではなく検出技法をするため，各検出技法を用いたツールを作成し，実験を行った．行単位，字句単位の検出は各プログラミング言語用に解析器を作る必要があるが，それほどコストは高くないと述べている．また，メトリクス計測による検出はソースコードから様々な情報を得なければならないので，解析器を作るコストは高いと述べている．

8. あ と が き

本研究では，ソースコード中の繰り返し部分を折りたたんで検出する手法を提案した．提案手法を用いたコードクローン検出ツールを実装し，オープンソースソフトウェアに対して検出を行った．その結果，折りたたみ有りの場合と折りたたみ無しの場合で検出数が減少したことを確認した．さらに，さらに既存の検出手法との比較を通じて，検出されたコードクローンの評価を行った．その結果，繰り返し部分の折りたたみにより，既存手法の問題点を解消したことを示した．

今後の課題は以下の通りである．

- 検出対象の言語を増やす．
- インクリメンタルな検出に対応する．インクリメンタルな検出とは，検出処理で得た情報をデータベースに登録し，それ以降の検出処理においてデータベースを活用することである．これにより，検出時間を短縮できる．
- 現在は，コードクローンの情報をテキストファイルに出力している．より実用性を高めるためには，表示方法の改良や Eclipse プラグインとして実装するなど，ユーザインターフェースの充実も必要であると考えられる．

謝辞 本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A）（課題番号：21240002）および萌芽研究（課題番号：23650014），文部科学省科学研究費補助金若手研究（B）（課題番号：22700031）の助成を得た．

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, Vol. J91-D, No. 6, pp. 1465–1481, 2008.
[2] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. pp. 319–27, 1990.
[3] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Software Engineering*, Vol. 31, No. 10, pp. 804–818, Oct. 2007.
[4] E.Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. pp. 36–43, 2002.
[5] F. V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. pp. 336–339, 2004.
[6] Detection of Software Clones. `http://bauhaus-stuttgart.de/clones/`.
[7] 肥後芳樹, 吉田則裕, 楠本真二, 井上克郎. 産学連携に基づいたコードクローン可視化手法の改良と実装. 情報処理学会論文誌, Vol. 48, pp. 811–822, 2007.
[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.
[9] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.
[10] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees, 1998.
[11] J. Mayland, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. pp. 244–253, 1996.
[12] L.Prechelt, G.Malpohl, and M.Phlippsen. Jplag: Finding plagiarisms among a set of programs. *Technical report, University of Karlsruhe, Department of Informatics*, 2000.
[13] Moss: A system for detecting software plagiarism. `http://www.cs.berkeley.edu/~aiken/moss.html`.