情報処理学会論文誌 Vol.54 No.2 845–856 (Feb. 2013)

ソースコード中の繰返し部分に着目したコードクローン検出ツールの実装と評価

村上 寛明[1,a)] 堀田 圭佑[1,b)] 肥後 芳樹[1,c)] 井垣 宏[1,d)] 楠本 真二[1,e)]

受付日 2012年5月14日, 採録日 2012年11月2日

概要：これまでにさまざまなコードクローン検出手法が提案されている．その中で行単位や字句単位の検出手法が広く利用されている．しかし，行単位や字句単位の検出手法には，命令の繰返しを含むソースコードにおけるコードクローン検出能力が低いという課題がある．そこで，本論文では，ソースコード中の繰返し部分に着目したコードクローン検出手法を提案する．提案する検出手法は，既存手法が検出してしまうオーバラップするコードクローンの検出を抑えることができる．提案手法を検出ツール FRISC として実装し，オープンソースソフトウェアに対して実験を行った．その結果，提案手法を用いることにより，検出結果の適合率が約 50%上昇したこと，および再現率が約 3%低下したことを確認した．また，提案手法は既存手法が検出できないコードクローンを検出できていることも確認できた．

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

Implementation and Evaluation of Code Clone Detection Method Designed for Information of Repetition in Source Code

Hiroaki Murakami[1,a)] Keisuke Hotta[1,b)] Yoshiki Higo[1,c)] Hiroshi Igaki[1,d)] Shinji Kusumoto[1,e)]

Received: May 14, 2012, Accepted: November 2, 2012

Abstract: A variety of code clone detection methods have been proposed. Among these methods, line-based and token-based ones are widely used. However, line/token-based detection methods have some issues that they have low performance of detection in repeated instructions. In this paper, we propose a new code clone detection method that is free from the influence of repeated instructions. The proposed method prevents overlapped code clones from being detected in repeated instructions. The proposed method has already been developed as a software tool, FRISC. We have conducted an experiment with some open source software systems to evaluate the usefulness of FRISC. As a result, we confirmed that the precision increased by about 50% and the recall decreased by about 3%. We also confirmed that the proposed method could detect some code clones not detected by the existing methods.

Keywords: code clone, program analysis, software maintenance

1. はじめに

コードクローンはソースコード中に存在する同一，あるいは類似するコード片である．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 つとして指摘されている．コードクローンを自動的に検出するため，これまでにさまざまな手法が提案され，ツールが実装されている [1], [2]．しかし，その中でも広く利用されているのは，行単位や字

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] h-murakm@ist.osaka-u.ac.jp
[b)] k-hotta@ist.osaka-u.ac.jp
[c)] higo@ist.osaka-u.ac.jp
[d)] igaki@ist.osaka-u.ac.jp
[e)] kusumoto@ist.osaka-u.ac.jp

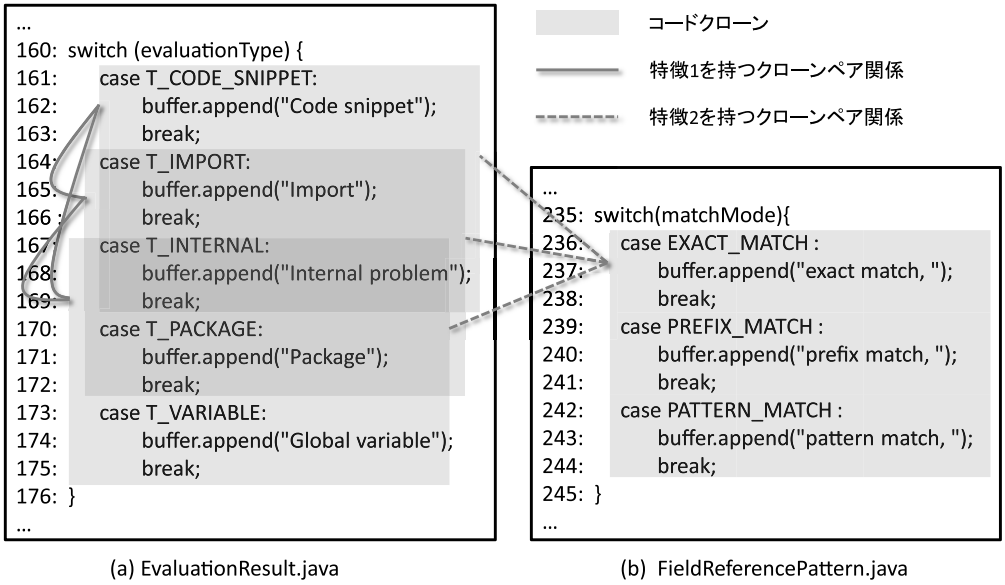


図 1 繰返し部分における既存手法の検出例

Fig. 1 An example of code clone detected by existing methods in repated instructions.

句単位の検出手法である．その理由を以下に示す．

- 行単位や字句単位の検出手法は，ソースコードの字句解析と簡単な構文解析を行うのみであり，抽象構文木やプログラム依存グラフなどを生成しない．よって高速に検出処理を行うことが可能であり，大規模ソフトウェア [3], [4]，多数のソフトウェア間 [5], [6], [7]，ソフトウェアのリビジョン群 [8], [9], [10], [11] に対しても適用可能である．
- ソースコードの深い解析を必要としないため，他の検出手法に比べて複数のプログラミング言語に対応させることが難しくない．たとえば，プログラム依存グラフを用いた検出手法は，プログラム依存グラフを構築するための解析を行わなければならない．実際に，代表的な行単位や字句単位の検出ツールである Simian [12] や CCFinder [3] は，C/C++，Java，COBOL などの社会で広く利用されている複数のプログラミング言語に対応している．

一方で，既存の行単位や字句単位の検出手法には，ソースコード中の同じ命令が繰り返し記述された部分（以降，**繰返し部分**と呼ぶ）においてオーバラップするコードクローンを検出してしまう，という課題がある．ここで，あるコードクローンが他のコードクローンと一部でも重なり合うとき，それらをそれぞれオーバラップするコードクローンと呼ぶ．オーバラップするコードクローンについては 2 章で説明する．本論文では，繰返し部分における課題を改善した検出手法を提案する．具体的には，検出の前処理として，繰返し部分を折りたたむことにより，ソースコード中から繰返し部分を取り除く．その後に，検出処理を行うことにより，繰返し部分が検出に与える悪影響を避ける．本論文の貢献は次のとおりである．

- ソースコード中の繰返し部分に着目し，既存手法の課題点を改善する手法を提案した．さらに，提案手法をコードクローン検出ツール FRISC として実装した．
- 繰返し部分の折りたたみによって，オーバラップするコードクローンの検出が大幅に抑えられることを複数のオープンソースソフトウェアに対して確認した．
- FRISC と既存の複数のコードクローン検出ツールに対して精度比較実験を行った．その結果，FRISC は既存の行単位や字句単位の検出ツールに比べてオーバラップするコードクローンの検出数や検出漏れが少ないことを確認した．

2. 研究動機

図 1 に既存の検出ツールによって繰返し部分から検出されたコードクローンの例を示す．この例においては，case エントリが左のソースファイルでは 5 回，右のソースファイルでは 3 回繰り返されている．既存の行単位や字句単位の検出ツールを用いると，図 1 に示すように 6 つのクローンペア（互いに同一，あるいは類似したコード片の対）が得られる．繰返し部分から得られるクローンペアは次に示す 2 つの特徴のうち，いずれかを持つことがある．

特徴 1 クローンペアを構成する 2 つのコード片が一部重なり合っている．

特徴 2 クローンペアを構成する 2 つのコード片が，別のクローンペアを構成する 2 つのコード片と一部重なり合っている．

これらの 2 つの特徴のうち，いずれかを持つクローンペアを構成するコード片をオーバラップするコードクローンと定義する．既存手法ではオーバラップするコードクローンを検出しているため，検出結果が膨大になってしまう．

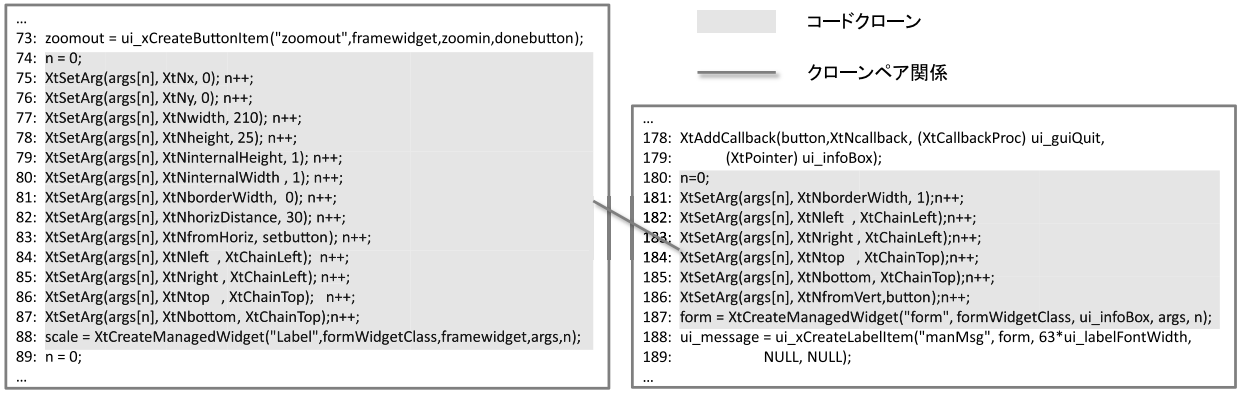


図 2 人間が繰返し部分を一括してコードクローンであると判断した例

Fig. 2 An example of code clone that human regards whole the repeated instructions as a code clone.

そのため，検出対象における他の部分のコードクローンを見つけ難くなるという課題がある．この課題は，繰返し部分の折りたたみによって改善が可能であると考えられる．繰返し部分の折りたたみを端的に述べると，繰り返し記述された同じ命令を 1 つの命令に変換するソースコード上の操作である．「繰返し部分」と「折りたたみ」の定義は，それぞれ 3 章と 4 章で行っている．図 1 の例に対して，繰返し部分の折りたたみを行った後に検出処理を行うと，左のソースファイルの 161–175 行目と右のソースファイルの 236–244 行目が 1 つのクローンペアとして検出される．つまり繰返し部分の折りたたみを行うことで，オーバラップするコードクローンの検出を抑制できる．

また**図 2** に，既存研究において人間がコードクローンであると判断した例 [13] を示す．この例では，コードクローン中に異なる回数の繰返し部分が含まれている．結果として左右のソースファイルに含まれるコード片の長さが異なるため，既存手法では検出が困難である．提案手法による繰返し部分の折りたたみにより，このような異なる回数の繰返し部分を含むコードクローンも検出が可能であると考えられる．

我々は，繰返し部分の折りたたみを行った後に検出処理を行う手法を提案する．本研究では以下の Research Questions を調査する．

RQ1 繰返し部分の折りたたみによって検出結果の再現率・適合率は向上するのか．

RQ2 提案手法は既存手法と比べて高い精度で検出できるのか．

再現率は検出漏れが少ないほど高い値を示す指標であり，適合率は冗長な検出が少ないほど高い値を示す指標である．

3. 用語の定義

本章では，ソースコード中の繰返し部分の定義を行う．はじめに，この定義を行ううえで必要となる諸用語の定義を行う．まず列および部分列を以下に定義する．

Definition3.1（列）n 以下のすべての自然数からなる集合を I とする．このとき，ある集合 X に対して I を定義域とする写像 $\phi : I \to X$ を列と呼び，$\phi = \langle x_1, x_2, \ldots, x_n \rangle (\forall i \in 1..n[i \in I \wedge x_i \in X])$ と表記する．

Definition3.2（部分列）ある集合 X と n 以下のすべての自然数からなる集合 I に対して，列 $\phi = \langle x_1, x_2, \ldots, x_n \rangle (\phi : I \to X)$ を考える．また k 以上 $k+j$ $(\leq n)$ 以下のすべての自然数からなる集合を J $(J \subset I)$ とする．このとき，式 (1) を満たす列 $\phi' = \langle x_k, x_{k+1}, \ldots, x_{k+j} \rangle (\phi' : J \to X' \subset X)$ を列 ϕ の部分列であると定義する．

$$\forall i \in J[\phi'(i) = \phi(i) \wedge x_i \in X'] \tag{1}$$

また，ある列 $\phi = \langle x_1, x_2, \ldots, x_n \rangle$ に含まれる要素の数 n を列 ϕ の長さと呼び，$|\phi|$ と表記する．

さらに，2 つの列 ϕ, ψ $(\phi, \psi : I \to X)$ の連結を以下に定義する．

Definition3.3（列の連結）2 つの列 $\phi = \langle x_a, x_{a+1}, \ldots, x_{a+l} \rangle$，$\psi = \langle x_b, x_{b+1}, \ldots, x_{b+m} \rangle$ について，以下のようにして連結して得られる列を $\phi \cdot \psi$ と表記する．

$$\phi \cdot \psi = \langle x_a, x_{a+1}, \ldots, x_{a+l}, x_b, x_{b+1}, \ldots, x_{b+m} \rangle \tag{2}$$

次に，あるソースファイル S を考える．一般的に，ソースファイルはプログラムを構成する要素（文，条件式など．以降，**プログラム要素**とする）が連結されて構成される列であると見なすことができる．すなわち，n 以下のすべての自然数からなる集合を I，s_i（$i \in I$）を任意のプログラム要素とするとき，ソースファイル S を以下のように表記できる．

$$S = \langle s_1, s_2, \ldots, s_n \rangle \tag{3}$$

以降，式 (3) を満たす列 S を**プログラム要素列**と呼ぶ．次に，2 つのプログラム要素列 ϕ，ψ の同値を定義する．

図 3 繰返し要素と繰返し部分列
Fig. 3 Repeated elements and repeated subsequence.

Definition3.4（プログラム要素列の同値関係）ϕ，ψ が以下の条件を満たすとき，ϕ と ψ は同値であるといい，$\phi \equiv \psi$ と表記する．

- $|\phi| = |\psi|$
- $\phi = \langle s_a, s_{a+1}, \ldots, s_{a+l} \rangle$，$\psi = \langle s_b, s_{b+1}, \ldots, s_{b+l} \rangle$（$l = |\phi| = |\psi|$）とするとき，$\forall i \in 0..l[s_{a+i} \equiv s_{b+i}]$

ここで，2 つのプログラム要素 s，s' が同値である（$s \equiv s'$）であるとは，プログラム要素 s，s' を構成する文字列が等しいことを意味する．

これらの定義を用い，あるプログラム要素列 S 中の**繰返し部分列**と**繰返し要素**を以下に定義する．

Definition3.5（繰返し部分列，繰返し要素）あるプログラム要素列 $S = \langle s_1, s_2, \ldots, s_n \rangle$ およびその部分列 $S_r = \langle s_i, s_{i+1}, \ldots, s_j \rangle$（$i,\ j \in 1..n \wedge i < j$）を考える．このとき，$S_r$ が以下の条件を満たすならば，S_r を S 中の**繰返し部分列**と定義する．

- $S_r = S_{T_1} \cdot S_{T_2} \cdot \ldots \cdot S_{T_u} \wedge \forall j, \forall k \in 1..u[S_{T_j} \equiv S_{T_k}]$ となる部分列 $S_{T_1}, \ldots, S_{T_u}$ が存在する．

またこのとき，繰返し部分列 $S_r = S_{T_1} \cdot S_{T_2} \cdot \ldots \cdot S_{T_u}$ を構成するそれぞれの部分列 $S_{T_1}, \ldots, S_{T_u}$ を**繰返し要素**と呼ぶ．

本論文では，ソースファイル中に出現する繰返し部分列を**ソースコード中の繰返し部分**であると定義する．

図 3 に繰返し要素と繰返し部分列の例を示す．図 3 において，プログラム要素列 “ABCABCABCABCD” が与えられたとき “ABC” が繰返し要素，“ABCABCABCABC” が繰返し部分列である．

4. 提案手法

提案手法は以下の 5 つのステップで構成される．

ステップ 1 字句解析・正規化
ステップ 2 文のハッシュ値の算出
ステップ 3 繰返し部分の折りたたみ
ステップ 4 一致部分ハッシュ値列の検出
ステップ 5 出力整形処理

提案手法の入力を以下に示す．

- 対象ソフトウェアのソースファイル（以降，**対象ファイル**）
- 繰返しと判断する文の数の最大値（以降，**繰返しの最大値**）
- 検出するコードクローンの最小の大きさ（以降，**最小コードクローン長**）

図 4 に提案手法全体の流れを示す．提案手法の出力は対象ファイルに含まれるクローンセット（互いに同一，あるいは類似したコード片の集合）である．以降，各ステップについて詳細に述べる．

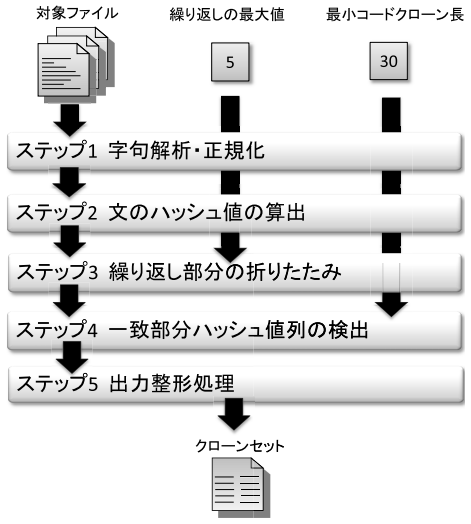


図 4 提案手法全体の流れ
Fig. 4 Overview of proposed method.

ステップ 1 字句解析・正規化

各対象ファイルを字句列に変換する．このステップではコメントと空白と改行記号とタブを削除し，予約語や記号はそのまま残し，ユーザ定義名を特殊文字に置換する．

ステップ 2 文のハッシュ値の算出

まず，ステップ 1 で得られた字句列中に含まれる**文**を特定する．ここで，文とは “;”，“{”，“}” で区切られた字句列である．提案手法では，このようにして特定した文をプログラム要素であると見なす．次に文ごとに字句を連結し，文単位でハッシュ値を算出する．ここで，すべてのハッシュ値は文に含まれる字句の数を**重み**として持っている．提案手法ではプログラム要素の同値判定の際に，文字列を直接比較せずこのハッシュ値を使用する．この理由は，文字列を直接比較するよりもハッシュ値の比較を行った方がより小さいコストで同値判定を行うことができるためである．

ステップ 3 繰返し部分の折りたたみ

まず折りたたみの定義を行う．

Definition4.1（折りたたみ）S_r をプログラム要素列 S 中の繰返し部分列であるとし，$S_r = S_{T_1} \cdot S_{T_2} \cdot \ldots \cdot S_{T_u}$ とする．このとき，繰返し部分列 S_r から先頭の繰返し要素 S_{T_1} 以外のすべての繰返し要素を削除する操作のことを**折りたたみ**と定義する．すなわち，$S_r = S_{T_1} \cdot S_{T_2} \cdot \ldots \cdot S_{T_u}$ を $S_{r_{folded}} = S_{T_1}$ とする操作のことを折りたたみと定義する．ここで，$S_{r_{folded}}$ は折りたたまれた繰返し部分列を表す．

この定義を用い，ステップ 2 で得られたハッシュ値列の中

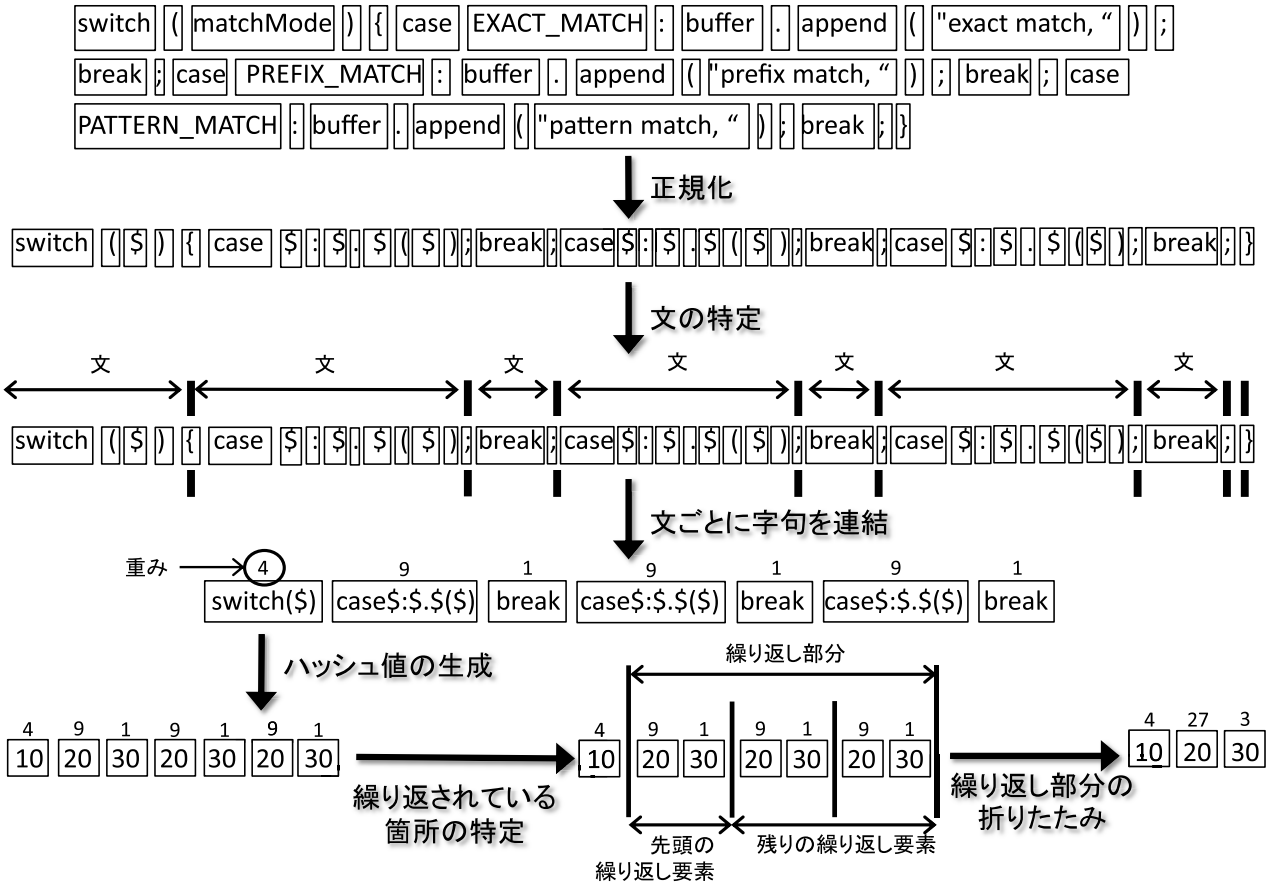


図 5 字句列の正規化からハッシュ値が折りたたまれるまでの動作例

Fig. 5 An example how token sequence is transformed into a folded hash sequence.

で繰り返されている箇所の折りたたみを行う．Algorithm 1 に折りたたみ処理のアルゴリズムを示す．なお Algorithm 1 において，sequence はハッシュ値列，repetition は繰返しの最大値をそれぞれ表している．折りたたみ処理では，まずハッシュ値列中における繰返し部分を特定する．次に繰り返されているハッシュ値列すべてについて折りたたみを行う．このとき削除された繰返し要素の重みは先頭の繰返し要素の重みに加えられる．**図 5** に折りたたみ処理の動作例を示す．この例では 3 つの case エントリが繰返し要素として特定され，折りたたまれている．

ステップ 4 一致部分ハッシュ値列の検出

ステップ 3 で得られたハッシュ値列に対して Suffix Array [14] アルゴリズムを用いて一致する部分ハッシュ値列を検出する．重みの合計が最小コードクローン長以上のハッシュ値列がこのステップの出力となる．

ステップ 5 出力整形処理

ステップ 4 で得られた一致するハッシュ値列をソースコードと対応づけてクローンペアにする．すべてのクローンペアの情報を得た後に，クローンセットを作成し出力する．

5. 実装

提案手法に基づいてコードクローン検出ツール FRISC を実装した．FRISC は Java 言語で記述され，総行数は 783 行である．現在のところ，FRISC は検出可能言語として Java 言語と C 言語に対応している．しかし，FRISC はプログラミング言語に依存する処理としては字句解析しか行っていないため，他のプログラミング言語に対応させることは難しくはない．また，FRISC はコードクローンの検出処理をマルチスレッドで行うことができる．

さらに，FRISC は Java 言語で記述されたコードクローンの検出精度を向上させるために下記の正規化をステップ 1 で行っている．

- “.” でつながれたユーザ定義名をまとめて 1 つの特殊文字に置換する．この正規化を行うことにより，たとえば，クラス名のみの記述と完全限定名の記述が同じ特殊文字になるため，それらの違いがあるコード片もコードクローンとして検出できる．**図 6** に正規化の例を示す．
- 通常，import 文や package 文をコードクローン検出の対象とする必要はない．そこで正規化を行う際に，import 文と package 文を取り除く．

Algorithm 1 Folding Repeated Hash Sequence

Require: *sequence*, *repetition*(≥ 1)
Ensure: folded *sequence*
1: $seqlen \leftarrow length(sequence)$
2: **for** $i = 0$ to *repetition* **do**
3: $left \leftarrow 0$
4: **loop**
5: $flg \leftarrow true$; $index \leftarrow left$; $tmpleft \leftarrow left$; $count \leftarrow 0$;
6: **while** $count \leq i$ *and* $index < seqlen$ **do**
7: **if** $isSentenceEnd(sequence[index])$ **then**
8: **if** flg **then**
9: $k \leftarrow index + 1$; $flg \leftarrow false$
10: **end if**
11: $count \leftarrow count + 1$
12: **end if**
13: $index \leftarrow index + 1$
14: **end while**
15: **if** $index > seqlen$ **then**
16: *break*
17: **end if**
18: $tmp \leftarrow sequence[left..index - 1]$
19: $count \leftarrow 0$
20: $left \leftarrow index$
21: **while** $count \leq i$ and $index < seqlen$ **do**
22: **if** $isSentenceEnd(sequence[index])$ **then**
23: $count \leftarrow count + 1$
24: **end if**
25: $index \leftarrow index + 1$
26: **end while**
27: **if** $index > seqlen$ **then**
28: *break*
29: **end if**
30: $tmp2 \leftarrow sequence[left..index - 1]$
31: **if** $tmp = tmp2$ **then**
32: $sequence \leftarrow sequence[0..left - 1] + sequence[index..seqlen]$
33: $seqlen \leftarrow length(sequence)$
34: $left \leftarrow tmpleft$
35: **else**
36: $left \leftarrow k$
37: **end if**
38: **end loop**
39: **end for**
40: **return** *sequence*

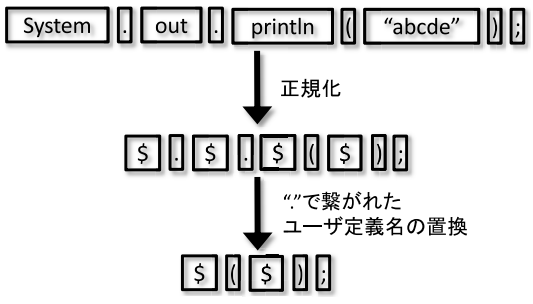


図 6 “.” でつながれたユーザ定義名の正規化の例
Fig. 6 Normalization of user-defined names connected by “.”.

- 修飾子を除去する．ここで修飾子とは，“public”，“protected”，“private”，“static”，“final”，“abstract”，“abstract”，“native”，“synchronized”，“transient”，“volatile”，“strictfp”，“const” を指す．

表 1 対象ソフトウェア
Table 1 Target software.

ソフトウェア名	言語	総行数	正解クローン数
netbeans	Java	14,360	55
ant	Java	34,744	30
jdtcore	Java	147,634	1,345
swing	Java	204,037	777
weltab	C	11,460	275
cook	C	70,008	440
snns	C	93,867	1,036
postgresql	C	201,686	555

6. 実験の概要

本研究では以下の 2 つの実験を行う．

実験 A 折りたたみの有無による精度比較実験

実験 B FRISC と既存ツールの精度比較実験

本研究の実験では以下に示す閾値を用いる．

最小コードクローン長 30

繰返しの最大値 5

本研究の実験では，Bellon らの実験で使用されたデータ [15] を検出すべきコードクローン群とする．このコードクローン群は，8 つの対象ソフトウェアから複数のコードクローン検出ツールを用いて検出し，さらに検出結果から Bellon らの目視確認によって抽出されたコードクローンの集合である[*1]．**表 1** に対象ソフトウェアの概要を示す．また，この実験では以下の用語を用いる．

クローン候補 各ツールが検出したコードクローン

正解クローン Bellon らが抽出したコードクローン

正解クローンは Bellon らにより以下の 3 種類に分類されている．

Type1 空白やタブを除いて同一なコードクローンのペア

Type2 変数名や関数名などユーザ定義名が異なるコードクローンのペア

Type3 Type1 や Type2 が許容する変更に加え，文単位や式単位で異なる部分を含むコードクローンのペア

クローン候補が正解クローンであるか否かは ok 値 [13] を用いて判断する．ok 値とはクローン候補が正解クローンを包含している割合を表す指標であり，包含している部分が大きいほど高い値を示す．本論文では Bellon らの実験と同様，ok 値の閾値として 0.7 を用いる．

クローン候補の集合を R，正解クローンの集合を S_{refs}，R と S_{ref} に共通して含まれているコードクローンの集合を S_R としたとき，R の再現率（$Recall_R$）と適合率（$Precision_R$）は，それぞれ以下の式で定義される．なお，式 (4) と式 (5) において $|A|$ は集合 A の要素数を表す．

[*1] ツールによって検出されたコードクローンをそのまま抽出する場合もあるが，検出されたコードクローンを一部加工（周囲のコード片の追加・削除）して抽出する場合もある．

$$Recall_R = \frac{|S_R|}{|S_{refs}|} \quad (4)$$

$$Precision_R = \frac{|S_R|}{|R|} \quad (5)$$

7. 実験 A 折りたたみの有無による精度比較実験

表 2 に折りたたみの有無によるクローン候補数と減少率を示す．表中の減少率とは折りたたみを行うことで減少したクローン候補数の割合を表す．折りたたみありの場合の検出数を C_{folded}，折りたたみなしの場合の検出数を C_{non_folded} とすると減少率 R_{dec} は以下の式で表される．

表 2 折りたたみの有無によるクローン候補数と減少率
Table 2 Number of clone candidates and decreasing rate of experiment A.

ソフトウェア名	折りたたみなし	折りたたみあり	減少率
netbeans	2,064	1,696	18%
ant	3,879	2,106	46%
jdtcore	57,769	21,494	63%
swing	37,373	20,606	45%
weltab	2,390	1,969	18%
cook	7,028	7,222	−3%
snns	20,335	11,940	41%
postgresql	34,256	15,362	55%

$$R_{dec} = \frac{C_{non\_folded} - C_{folded}}{C_{non\_folded}} \quad (6)$$

折りたたみを行うことで，cook を除くすべてのソフトウェアにおいてクローン候補数が減少した．特に jdtcore に対しては減少率が 63%であり，これはすべての対象ソフトウェアの中で最大である．jdtcore のソースコードを閲覧したところ，多数の連続する if-else 文や case エントリが含まれていた．これらの部分からクローン候補を検出しなくなったことが，クローン候補数の大幅な減少につながった．

一方で cook に対する減少率は −3%であった．これは繰返し部分の折りたたみを行うことでクローン候補数が増えたことを表す．折りたたみにより新たに増えた検出例を**図 7** に示す．折りたたみなしの場合は左のソースファイルの 28–31 行目と右のソースファイルの 118–121 行目が一致しているが，コード片を構成する字句の数が最小コードクローン長に満たないため検出されない．しかし，折りたたみありの場合は右のソースファイルの 121 行目と 122 行目が折りたたまれるため，字句の数が最小コードクローン長を超え，図 7 のように検出される．

図 8 (a) に折りたたみの有無による適合率を示す．最大では 112%の増加，最小では 2%の増加となり，すべてのソフトウェアについて折りたたんだ場合の方が高かった．

図 8 (b) に折りたたみの有無による再現率を示す．最大では 5%の増加，最小では 9%の減少という結果を得た．

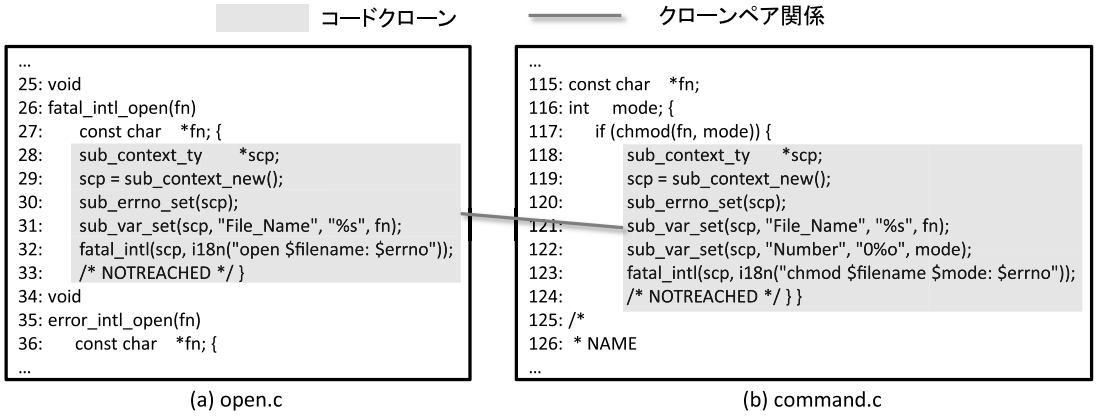
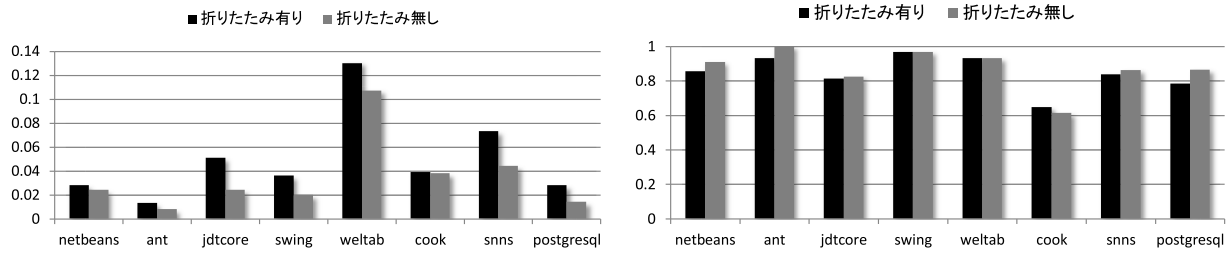


図 7 折りたたみにより新たに増えたクローン候補
Fig. 7 A clone candidate newly detected by using the folding operation.



(a) 折りたたみの有無による適合率　(b) 折りたたみの有無による再現率

図 8 折りたたみの有無による適合率と再現率
Fig. 8 Precision and recall in experiment A.

cook に対しては，折りたたみを行うことで再現率が増加した．swing と weltab の 2 つのソフトウェアに対しては再現率が変化しなかった．残りの 5 つのソフトウェアに対しては再現率が減少した．

RQ1 について次のように回答する．繰返し部分の折りたたみを行うことで，クローン候補数は最大で 63%の減少，最小で 3%の増加という結果を得た．適合率は最大で 112%の増加，最小で 2%の増加という結果を得た．再現率は最大で 5%の増加，最小で 9%の減少という結果を得た．すなわち，提案手法は繰返し部分におけるオーバラップするクローン候補の検出を抑制しているが，検出しなくなったクローン候補の中にいくつかの正解クローンが含まれていた．

8. 実験 B FRISC と既存ツールの精度比較実験

表 3 に比較対象となるコードクローン検出ツールの概要を示す．Nicad [16] を除くすべてのツールは Bellon らの実験 [13] で使用されたものである．

Nicad は Roy と Cordy が開発したコードクローン検出ツールである．Nicad はその検出の単位として，ブロック単位または関数単位のいずれかを選択できる．今回はブロック単位の検出，および比較を行った．Nicad は，すべてのブロックや関数のペアについて LCS（Longest Common Subsequence）アルゴリズムを用いることで，一致するブロックまたは関数を特定し，それらをコードクローンとして検出する．

表 4 に，対象ソフトウェアにおける FRISC と既存ツールのクローン候補数を示す．表中の “-” はスケーラビリティの問題により，ツールが検出処理を完了できなかったことを表す．行単位や字句単位の検出手法を用いたツールは，他の検出手法と比べて多くのクローン候補を検出していることが分かる．

図 9 に，対象ソフトウェアにおける FRISC と既存の検出ツールの再現率と適合率を示す．FRISC の再現率は 8 つの対象ソフトウェアのうち 5 つのソフトウェアについて最も高かった．ant と snns については，FRISC の再現率は 2 番目に高かった．cook については，FRISC の再現率は 3 番目に高かった．

字句単位の検出ツールは，抽象構文木やプログラム依存グラフなどを用いた他の検出ツールと比べて適合率が低い [13]．FRISC も同様に既存の検出ツールの中では比較的，適合率が低い．しかし，FRISC と同じ字句単位の検出手法を用いたツールである CCFinder に対しては 6 つのソフトウェアにおいて優っていた．同じく字句単位の検出手法を用いたツールである Dup に対しては 4 つのソフトウェアにおいて優っていた．しかし，この実験で用いた正解クローンは，対象ソフトウェアに含まれるすべてのコードクローンから抽出されていない．そのため，適合率の低さがそのまま多くのオーバラップするクローン候補を検出しているとは単純に結論付けることはできない．詳しくは 9 章で述べる．

表 3 比較対象となるツール
Table 3 Tools used for comparison.

作成者	ツール	検出手法
Baker	Dup	字句単位の検出
Baxter	CloneDR	抽象構文木を用いた検出
Kamiya	CCFinder	字句単位の検出
Merlo	CLAN	メトリクスを用いた検出
Rieger	Duploc	行単位の検出
Krinke	duplix	プログラム依存グラフを用いた検出
Roy	Nicad	ブロック単位の検出

既存のツールでは検出できたが，FRISC では検出できなかった正解クローンを調査するため，そのような正解クローンから無作為に 100 個を抽出し，それらのソースコードを閲覧した．調査した 100 個の正解クローンを下記のように分類した．括弧内の数字は各分類における正解クローンの数を表す．

分類 A（70） Type3 の正解クローン
分類 B（21） 最小コードクローン長未満の正解クローン
分類 C（9） 繰返し部分内の正解クローン

以降，各分類について詳細に述べる．

分類 A Type3 の正解クローンは行の挿入や削除が行われているため，コードクローンと見なされる字句列間に不一致部分が存在する．しかし FRISC は字句単位の検出を行っているため，字句列が連続して一致している場合のみ検出可能である．したがって FRISC は Type3 の正解クローンを検出することはできない．

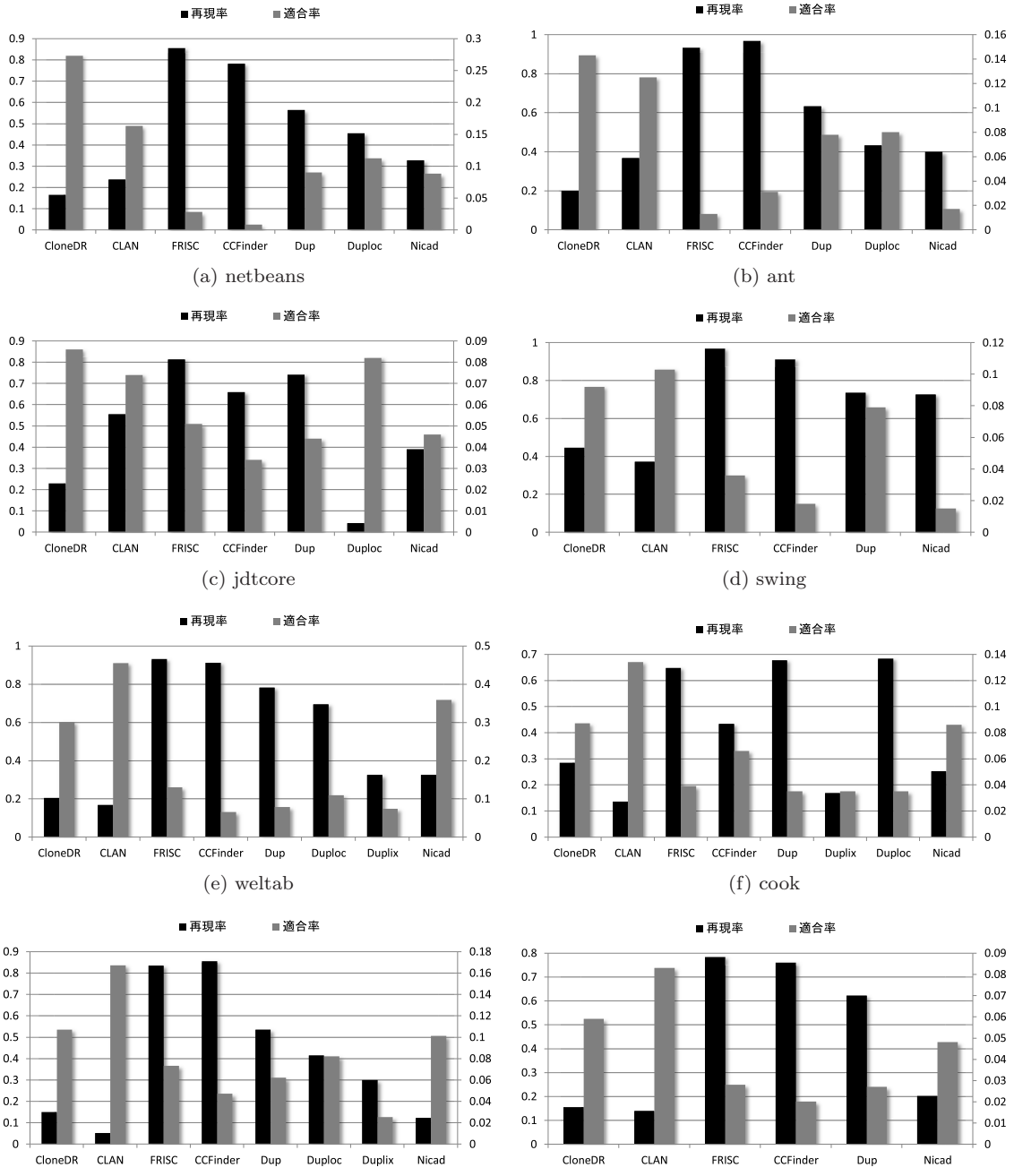
分類 B Bellon らの正解クローンの定義は 6 行以上一致するコード片である．一方で FRISC は，最小コードクローン長以上の字句が一致するコード片をクローン候補として検出する．そのため FRISC は，6 行以上であるが字句の数が最小コードクローン長を超えないコード片をクローン候補として検出することはできない．

分類 C **図 10** に繰返し部分内に存在する正解クローンの例を示す．この図では switch 文の中に 6 つの連続した case エントリが存在し，最初の 3 つのコード片と最後の 3 つのコード片がコードクローンになっている．提案手法は同じ命令が繰り返し記述された箇所を折りたたむため，分類 C の正解クローンは検出されない．しかし，このようなコードクローンを検出することは可能である．もし，折りたたんだ部分の重みの合計値が最小コードクローン長の 2 倍以上であれば，繰返し部分の前半部分と後半部分をクローンペアとすればよい．

RQ2 について次のように回答する．FRISC は大半の対

表 4 対象ソフトウェアにおけるクローン候補数
Table 4 Number of clone candidates detected by tools.

ソフトウェア名	FRISC	Dup	CloneDR	CCFinder	CLAN	Duploc	Duplix	Nicad
netbeans	1,696	344	33	5,552	80	223	-	24
ant	2,106	245	42	950	88	162	-	19
jdtcore	21,494	22,589	3,593	26,049	10,111	710	-	1,142
swing	20,606	7,220	3,766	21,421	2,809	-	-	1,804
weltab	1,969	2,742	186	3,898	101	1,754	1,201	160
cook	7,222	8,593	1,438	2,964	449	8,706	2,135	159
snns	11,940	8,978	1,434	18,961	318	5,212	12,181	352
postgresql	15,362	12,965	1,452	21,383	930	-	-	352



(a) netbeans (b) ant (c) jdtcore (d) swing (e) weltab (f) cook (g) snns (h) postgresql

図 9 対象ソフトウェアにおける FRISC と既存の検出ツールの再現率と適合率
Fig. 9 Precision and recall of all the detection tools for all the target software systems.

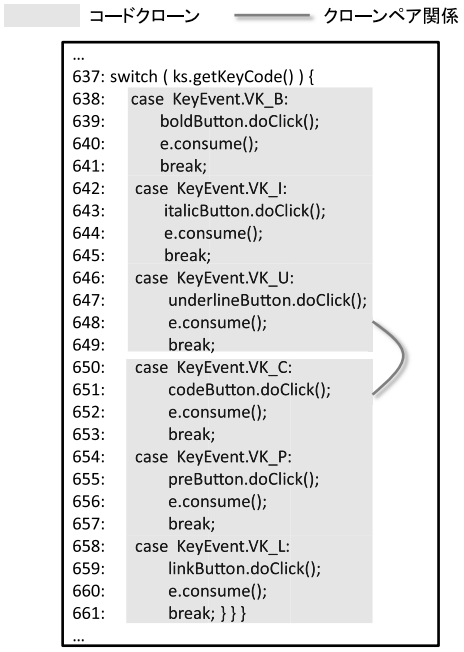


図 10 繰返し部分内の正解クローン
Fig. 10 A clone reference located in a repeated instructions.

象ソフトウェアにおいて，既存ツールよりも多くの正解クローンを検出できている．しかし，他の字句単位の検出ツールと同様に多くのオーバラップするクローン候補を検出している可能性がある．

9. 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，以下であげる点に留意する必要がある．

9.1 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，ツールの性能を調査した．しかし，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から抽出されたものではない．そのため，すべてのコードクローンを対象にして正解クローンの作成を行った場合，本研究で得られた結果と異なる結果が得られる可能性がある．しかし，対象ソフトウェアに含まれるすべてのコードクローンを対象にして正解クローンを作成することは現実的ではない．

9.2 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には，行単位や字句単位の検出手法を用いたツールのみではなく，抽象構文木やプログラム依存グラフを用いたツールなど多数存在する．各ツールによってコードクローンの定義が異なるため，他のツールを用いて比較を行った場合，本研究で得られた結果と異なる結果が得られる可能性がある．

10. 関連研究

肥後らは，クローンセットに含まれるコード片がどの程度繰返し要素を含まないかを表すメトリクス RNR を提案している [17]．このメトリクスを用いることで代入文の羅列や類似する処理を行う case エントリなど，同じ命令が繰り返し記述されたコードクローンを取り除くことができる．CCFinder [3] の後続ツールである CCFinderX [18] は，RNR を用いて検出結果のフィルタリングを行っている．

Koschke は決定木を用いて，同じ命令が繰り返し記述されたコードクローンを取り除く手法を提案している [5]．Koschke の実験においては，次に示す 2 つのメトリクスを決定木学習に用いる．

パラメータ類似度 クローンペアを構成する 2 つのコード片が，同じ変数やリテラルを使用している割合

NR コード片が繰返し要素を含まない割合

決定木を構築するためには学習データが必要である．しかし実験の結果，決定木を用いた分類の精度は非常に高く，その失敗率は 0.1%以下であった．

肥後らの手法および Koschke の手法は，検出されたコードクローンに対して事後的なフィルタリングを行うことによってオーバラップするコードクローンを取り除く手法である．一方，提案手法はソースコードをよりコードクローンの検出に適した形に変形した後に検出処理を行うことによって，オーバラップするコードクローンの検出を抑制する手法である．肥後らの手法および Koschke の手法は検出後の事後処理であるため，それによって新たにコードクローンが検出されることはない．したがって，これらの手法を用いた場合，適合率は向上する可能性があるものの再現率は向上しえない．しかし提案手法は検出前のソースコードに対して変形処理を行うため，変形処理を行わなければ検出することのできないコードクローンを新たに検出できる可能性がある．すなわち，提案手法を用いた場合は適合率および再現率の双方が向上する可能性がある．実際に今回の評価実験において，折りたたみ処理によって適合率および再現率が双方ともに向上した事例の存在が確認された．

11. あとがき

既存のコードクローン検出手法はソースコード中の繰返し部分において，検出する必要がないオーバラップするコードクローンを検出してしまう，という課題があった．そこで，繰返し部分を折りたたむという前処理を行った後に検出処理を行う手法を提案した．提案手法をコードクローン検出ツール FRISC として実装し，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，折りたたみありの場合が折りたたみなしの場合に比べてオーバラップするコードクローンの検出数が減少し，適合率が向上し

たことを確認した．また，既存の検出ツールとの比較を通じて提案手法の有用性を評価した．その結果，繰返し部分において既存ツールが検出していたオーバラップするコードクローンの検出を抑制できたこと，ならびに既存ツールでは検出できなかったコードクローンを検出できたことを確認した．

今後の課題は以下のとおりである．

- 検出対象言語を増やす．
- 増分的なコードクローン検出法を実装する．増分的な検出法では，検出結果はデータベースに保存され，次回以降の検出に利用される．このため，同じファイル群から複数回の検出を行う場合，2 回目以降の検出時間を大幅に短縮できる．

参考文献

[1] Code Clones Literature, available from ⟨http://students.cis.uab.edu/tairasr/clones/literature/⟩.

[2] Roy, C.K., Cordy, J.R. and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, pp.470–495 (May 2009).

[3] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).

[4] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code, *IEEE Trans. Softw. Eng.*, Vol.32, No.3, pp.176–192 (2006).

[5] Koschke, R.: Large-scale inter-system clone detection using suffix trees, *Proc. 16th European Conference on Software Maintenance and Reengineering*, pp.309–318 (Mar. 2012).

[6] Shang, W., Adams, B. and Hassan, E.: An experience report on scaling tools for mining software repositories using mapreduce, *Proc. 25th International Conference on Automated Software Engineering*, pp.275–284 (Sep. 2010).

[7] Livieri, S., Higo, Y., Matushita, M. and Inoue, K.: Very large scale code clone analysis and visualization of open source program using distributed ccfinder: D-ccfinder, *Proc. 29th International Conference on Software Engineering*, pp.106–115 (May 2007).

[8] Göde, N. and Koschke, R.: Frequency and risks of changes to clones, *Proc. 33rd International Conference on Software Engineering*, pp.311–320 (2011).

[9] Hotta, K., Sano, Y., Higo, Y. and Kusumoto, S.: Is duplicate code more frequently modified than non-duplicate code in software evolution?: An emprical study on open source software, *Proc. 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pp.73–82 (Sep. 2010).

[10] Krinke, J.: Is cloned code more stable than non-cloned code? *Proc. 8th IEEE International Workshop on Source Code Analysis and Manipulation*, pp.57–66 (Oct. 2008).

[11] Lozano, A., Wermelinger, M. and Nuseibeh, B.: Evaluating the relation between changeability decay and the characteristics of clones and methods, *Proc. 23rd International Conference on Automated Software Engineering*, pp.100–109 (Sep. 2008).

[12] Simian – Similarity Analyser, available from ⟨http://www.harukizaemon.com/simian/⟩.

[13] Bellon, S., Koschke, R., Antniol, G., Krinke, J. and Merlo, E.: Comparison and evaluation of clone detection tools, *IEEE Trans. Softw. Eng.*, Vol.31, No.10, pp.804–818 (2007).

[14] Manber, U. and Myers, G.: Suffix arrays: A new method for on-line string searches, *1st ACM-SIAM Symposium on Discrete Algorithms*, pp.319–27 (1990).

[15] Detection of Software Clones, available from ⟨http://bauhaus-stuttgart.de/clones/⟩.

[16] Roy, C.K. and Cordy, J.R.: Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, *Proc. 16th IEEE International Conference on Program Comprehension* (June 2008).

[17] 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).

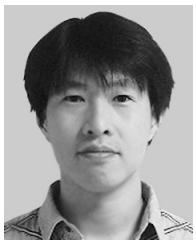
[18] CCFinderX, available from ⟨http://www.ccfinder.net/ccfinderx.html⟩.

村上 寛明

平成 24 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．コードクローン分析に関する研究に従事．

堀田 圭佑

平成 22 年大阪大学基礎工学部情報科学科卒業．平成 24 年同大学大学院情報科学研究科博士前期課程修了．現在，同研究科博士後期課程在学中．コードクローン分析に関する研究に従事．IEEE 会員．



肥後 芳樹 （正会員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院情報科学研究科博士後期課程修了．日本学術振興会特別研究員を経て，平成 19 年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助教．博士（情報科学）．コードクローン分析，リファクタリングに関する研究に従事．IEEE 会員．

井垣 宏 （正会員）

平成 12 年神戸大学工学部電気電子工学科卒業．平成 14 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．平成 17 年同研究科博士後期課程修了．同年同研究科特任助手．平成 18 年南山大学数理情報研究科講師．平成 19 年神戸大学工学部工学研究科情報知能学専攻特命助教．平成 22 年東京工科大学コンピュータサイエンス学部助教．平成 23 年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特任准教授．博士（工学）．ソフトウェア工学教育，サービス指向アーキテクチャ，ホームネットワークシステム，Web サービス，ソフトウェアプロセス等の研究に従事．IEEE，ACM，IEICE 各会員．

楠本 真二 （正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同学科講師．平成 11 年同学科助教授．平成 14 年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助教授．平成 17 年同専攻教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IEEE，JFPUG 各会員．