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

信学技報
IEICE Technical Report
SS2007-48(2007-12)

AGMアルゴリズムを用いたギャップを含むコードクローン情報の生成

肥後　芳樹†　　植田　泰士††　　楠本　真二†　　井上　克郎†

† 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　〒560-8531 豊中市待兼山町 1-3
†† 宇宙航空研究開発機構 情報・計算工学センター　〒305-8505 茨城県つくば市千現 2-1-1
E-mail: †{higo,kusumoto,inoue}@ist.osaka-u.ac.jp, ††ueda.yasushi@jaxa.jp

あらまし　これまでに多くのコードクローン検出手法が提案されているが，ギャップ (不一致部分) を含むコードクローンを検出できる手法は少ない．本稿では，ギャップを含むコードクローンを検出できないコードクローン検出手法の出力結果に対する後処理を行うことで，ギャップを含むコードクロー情報を生成する手法を提案する．提案手法は，グラフマイニングアルゴリズムの１つである AGM アルゴリズムを用いており，効率的にギャップを含むコードクローン情報を生成することができる．さらに，提案手法をコードクローン検出ツール CCFinder のポストプロセッサとして実装し，オープンソースソフトウェアに対して適用した．適用の結果，多数の興味深いギャップを含むコードクローン情報を得ることができた．

キーワード　コードクローン，ソフトウェア保守

Generating Gapped Code Clone Information using AGM Algorithm

Yoshiki HIGO†, Yasushi UEDA††, Shinji KUSUMOTO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
†† Jaxa's Engineering Digital Innovation Center, Japan Aerospace Exploration Agency
2-1-1, Sengen, Tsukuba, Ibaraki 305-8505, Japan
E-mail: †{higo,kusumoto,inoue}@ist.osaka-u.ac.jp, ††ueda.yasushi@jaxa.jp

Abstract　At present, there are many code clone detection methods and tools. However, only some of them can detect gapped code clones. A gapped code clone is a code clone including non-duplicated segments to its correspondant code clones. In this paper, we propose a method generating gapped code clone information from a detection result of existing code clone detection tools. The method adopts AGM (Apriori-baded Graph Mining) algorithm, and it can extract gapped code clone information efficiently. The method has already been implemented as a post-prosessor of CCFinder, which is a popular code clone detection tool. We applied the post-prosessor to four open source software systems, and got interesting gapped code clones.

Key words　Code Clone, Software Maintenance

1. まえがき

近年，ソフトウェア開発および保守支援の１つとしてコードクローン検出技術が注目を集めている．コードクローンとはソースコード中に存在する，他のコード片と同一または類似したコード片のことである [6]．コードクローンは，コピーアンドペーストや定型処理，パフォーマンス改善などのさまざまな理由によりソースコード中に作りこまれる．

コードクローンの存在により，ソフトウェア開発および保守が困難になるといわれている．例えば，あるコード片にバグが含まれていた場合，そのコードクローン全てに対して同様の修正の是非を検討する必要がある．このような作業は，特に大規模ソフトウェアでは非常に大きな手間を必要とする．したがって自動的にコードクローンを検出し，その情報を用いることは，ソフトウェア開発および保守を行うにあたり，非常に有効であるといえる．

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が，それらはどれも異なったコードクローンの定義をもつ．つまり，コードクローンの厳密で普遍的な定義は存在しない．Bellon は，コードクローン間の違いの度合いに基づき，それらを３つに分類している [2]．

タイプ 1:　空白やタブの有無，括弧の位置などのコーディング

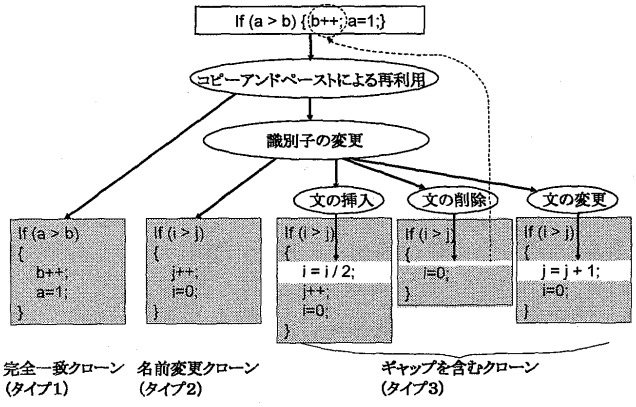


図 1　コピーアンドペーストによる再利用の流れ

スタイルを除いて，完全に一致するコードクローンを指す．

タイプ 2:　変数名や関数名などのユーザ定義名，または変数の型などの一部の予約語のみが異なるコードクローンを指す．

タイプ 3:　コピーアンドペースト後に文の追加や削除，変更が行われたなどの結果により，ギャップ (不一致部分) を含むコードクローンを指す．

図 1 は，コピーアンドペーストによる再利用の流れを表している．単にコピーアンドペーストを行っただけの再利用であれば，コピー元とコピー先は完全一致クローンになる．コピーアンドペースト後に変数名などの付け替えを行えば，名前変更クローンとなる．さらに，コピーしたコード片を文単位で変更した場合は，ギャップを含むコードクローンとなる．

これまでに提案されているコードクローン検出手法のうち，CP-Miner など一部の手法はタイプ 3 のコードクローンを検出することができるが [7], [10]，その他の多くの手法はタイプ 3 のコードクローンを検出することができない．検出するコードクローンの大きさの閾値を下げることにより，1 つのギャップを含むコードクローンを複数の完全一致クローンまたは名前変更クローンとしては検出することは可能であるが，分析者自らがそれら複数のコードクローンを 1 つのギャップを含むコードクローンであると認識しなければならず，効率的に分析作業ができるとはいえない．

本稿では，タイプ 1 およびタイプ 2 のコードクローン情報から，タイプ 3 のコードクローン情報を生成する手法を提案する．提案手法は，グラフマイニングアルゴリズムの 1 つである AGM アルゴリズムを用いてギャップを含むコードクローンを生成する．この手法は特定のコードクローン検出手法に依存しておらず，任意の既存の手法に対して適用可能である．提案手法をコードクローン検出の後処理として用いることにより，分析者はより有益なコードクローン情報を得ることができる．

2. AGM アルゴリズム

AGM(Apriori-based Graph Mining) アルゴリズムは，複数のラベル付きグラフに含まれる多頻度グラフパターンを効率的に抽出するアルゴリズムである [5]．AGM アルゴリズムの概要を図 2 に示す．ここで G はグラフの集合，F_k は大きさ k の多頻度グラフの隣接行列の集合，C_k は大きさ k の多頻度グラフの候補隣接行列の集合を指す．大きさ k のグラフとは，k 個のノードから構成されているグラフを意味する．また，$minsup$ は，多頻度とするために最低限必要な出現回数を表す．

```
1)  Main(G,minsup){
2)    C_1←{ 大きさ 1 の多頻度グラフの候補の隣接行列 };
3)    k←1;
4)    while(C_k≠∅){
5)      Count(G,C_k);
6)      F_k ← {c_k ∈ C_k | sup(G(c_k)) ≧ minsup};
7)      C_{k+1} ← Generate − Candidate(F_k);
8)      k ← k + 1;
9)    }
10)   return ⋃_k {f_k ∈ F_k | f_k is canonical};
11) }
```

図 2　AGM アルゴリズムの概要

AGM アルゴリズムは，大きさが 1 の多頻度グラフパターンから，順にサイズの大きなグラフパターンを抽出していく．以下に AGM アルゴリズム (図 2) の大まかな流れを示す．

2 行目:　1×1 の隣接行列が頂点ラベルの数だけ生成され，C_1 に代入される．

5-6 行目:　大きさが k である多頻度グラフパターンの候補の支持度を求め(注1)，支持度が $minsup$ 以上であれば(注2)，F_k に加える．

7 行目:　大きさ k の多頻度グラフパターンを組み合わせて，大きさ $k+1$ の多頻度グラフの候補を生成する．この操作は C_k が空集合になるまで続けられる．

10 行目:　全ての多頻度グラフパターンが出力される．

大きさが k の多頻度グラフパターンから，大きさが $k+1$ の多頻度グラフパターンの候補を生成する際の条件を変えることにより，連結グラフパターン，順序木パターン，グラフのパスなど，さまざまなグラフパターンを取り出すことが可能である．この手法を B-AGM(Biased Apriori-based Graph Mining) と呼ぶ．詳細については文献 [5] を参照されたい．

提案手法では，グラフのパスを抽出する条件を用いて，グラフのノードをタイプ 1 やタイプ 2 のコードクローン，グラフのエッジをコードクローン間のギャップとして，多頻度グラフパターンを抽出する．

3. 提案手法

提案手法は，以下の手順でギャップを含むコードクローン情報を生成する．

手順 1:　各ソースファイルのグラフを構築

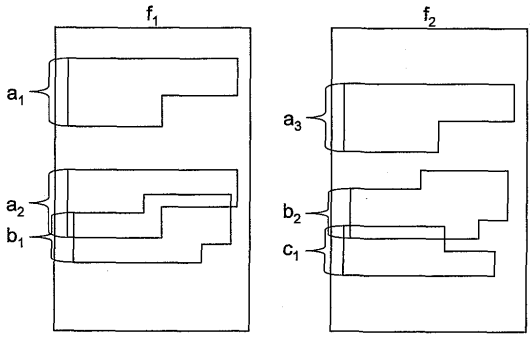
手順 2:　多頻度サブグラフ (=ギャップを含むコードクローン) の検出

手順 3:　生成したギャップを含むコードクローン情報の出力

手順 1 では，入力として与えられたタイプ 1 およびタイプ 2

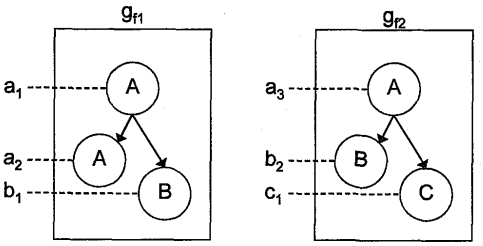
(注1)：$Count(G, C_k)$ は，グラフ集合 G 内でのグラフパターンの集合 C_k に含まれる各パターンの出現数 (支持度) を計測する関数である．

(注2)：$sup(G(c_k))$ は，パターン c_k の出現数 (支持度) を返す関数である．



コードクローン検出対象ファイル: {f_1, f_2,・・・,f_n}
クローンセットA: {a_1, a_2,・・・,a_x}
クローンセットB: {b_1, b_2,・・・,b_y}
クローンセットC: {c_1, c_2,・・・,c_z}

(a) 対象ファイル中のコードクローン



対象ファイルから生成されたグラフ: {g_{f1}, g_{f2},・・・,g_{fn}}
ノードのラベル: {A, B, C}

(b) 生成されたグラフ

図 3　グラフ構造の生成

のコードクローンの情報を元に，AGM アルゴリズムの入力となるグラフ集合を構築する．手順 2 では，AGM アルゴリズムを用いて，手順 1 で構築したグラフ集合に現れる多頻度サブグラフを検出する．最後に手順 3 では，手順 2 で検出した多頻度サブグラフをギャップを含むコードクローンの情報として出力する．以下，各手順の詳細を述べる．

手順 1: 各ソースファイルのグラフを構築

入力として与えられたタイプ 1 およびタイプ 2 のコードクローンの情報を元に，コードクローンをノードとするグラフをソースファイル毎に作成する．同一クローンセット(注3)に含まれるコードクローンから生成されたノードは同一ラベルをもつ．ある 2 つのコードクローンがあり，それらが離れた位置に存在する場合は，それらに対応するノードにエッジを引く．ファイル内における位置が上位のコードクローンから下位のコードクローンに向けてエッジが引かれる．この操作を全てのコードクローンの組み合わせに対して実行し，ソースファイル毎にグラフを構築する．

一部がオーバーラップしている場合や，あるコードクローンが他のコードクローンを完全に包含している場合には，これら 2 つのノードの間にはエッジを引かない．このような部分は，コピーアンドペースト後の修正により発生したコードクローンであるとは考えにくいからである．また，2 つのコードクローンがあまりに離れている場合も，それらのノード間にはエッジを引かない．この場合も，コピーアンドペースト後の修正により発生したコードクローンであるとは考えにくいからである．

図 3 は，グラフ構築の例を表している．この例では，n 個のコードクローン検出対象ファイル $\{f_1,f_2,\cdots,f_n\}$ から，3 つのクローンセット A,B,C が検出されている．クローンセット A には x 個のコード片 $\{a_1,a_2,\cdots,a_x\}$，クローンセット B には y 個のコード片 $\{b_1,b_2,\cdots,b_y\}$，クローンセット C には z 個のコード片 $\{c_1,c_2,\cdots,c_z\}$ が含まれている．

図 3(a) は，ファイル f_1 と f_2 に含まれるコードクローンの様子を表している．ファイル f_1 中のコード片 a_1 と a_2 および a_1 と b_1，ファイル f_2 中のコード片 a_3 と b_2 および a_3 と c_1 は離れて存在している (ギャップが存在する) ので，対応するノード間にエッジを引く．それに対して，ファイル f_1 中のコード片 a_2 および b_1，ファイル f_2 のコード片 b_2 と c_1 は，一部が重複している (ギャップが存在しない) ので，これらのノード間にはエッジを引かない (図 3(b) 参照)．

手順 2: 多頻度サブグラフの検出

手順 1 で構築したグラフ集合に対して AGM アルゴリズムを適用し，そのグラフ集合に現れる多頻度グラフパターンを抽出する．具体的には，グラフ集合に 2 回以上 (AGM アルゴリズムにおける最小支持度以上) 出現するパスをギャップを含むコードクローンのパターンとする．多頻度グラフパターンを求めた後，各パターンのインスタンスがグラフのどの部分に存在するのかを求める．図 3 の場合では，ファイル f_1 のパス a_1b_1 とファイル f_2 のパス a_3b_2 がギャップを含むコードクローンとして検出される．

手順 3: 生成したギャップを含むコードクローン情報の出力

手順 2 で検出したギャップを含むコードクローン情報を，入力として与えられたコードクローン情報と同様のフォーマットで出力する．

4. 実　　装

提案手法をコードクローン検出ツール CCFinder [8] のポストプロセッサとして実装した．CCFinder は，コードクローン検出結果をテキスト形式で出力するツールであり，分析作業は付属の可視化ツール Gemini [11] を用いて行う．

図 4 は，実装したポストプロセッサを用いたコードクローン分析の手順を表している．CCFinder は，タイプ 1 とタイプ 2 のコードクローンを検出できるため，従来は Gemini を用いてそれらの分析を行うことができた．実装したポストプロセッサを適用することにより，タイプ 1 とタイプ 2 に加えてタイプ 3，つまりギャップを含むコードクローンの分析も Gemini を用いて行えるようになる．

5. 適 用 実 験

5.1 概　　要

提案手法を用いることにより，どのようなコードクローンが検出されるかを調査するために適用実験を行った．この実験には，前節で述べたポストプロセッサを用いた．対象ソフトウェ

(注3)：クローンセットとは，互いに類似したコード片を全て含んだ集合である．

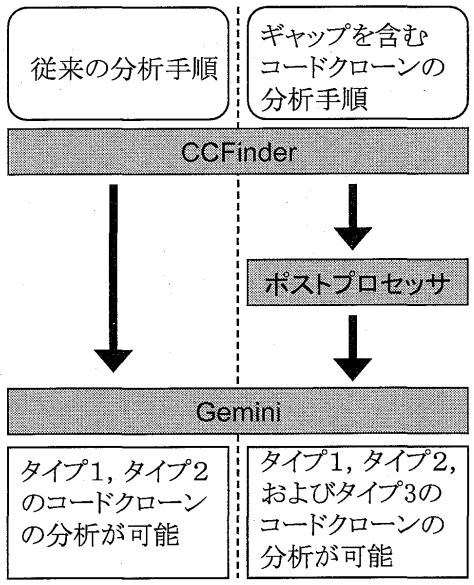


図 4　提案手法を用いたコードクローン分析の手順

アは，C 言語で記述された Canna 3.6 と httpd 2.2.4，および Java 言語で記述された Ant 1.6.0 と JHotDraw 7.0.9 の 4 つである．この実験では，CCFinder は 30 トークン(注4)以上のコードクローンを検出するように設定し，コードクローン間のギャップが 30 トークン以内の場合に，それらに対応するノード間にエッジを引くことにした．表 1 は，対象ソフトウェアの規模を表している．

提案手法を定量的に評価するために，以下の基準を設け，各基準に該当するギャップを含むクローンセットの数を調査した．
(1) 複数関数:　1 つのコードクローンが複数の関数やメソッドにまたがっているクローンセットを指す．関数内部がギャップとなっている場合は，関数内で処理の異なる部分がギャップとなっていることを意味し，そのような処理の違いを含むコードクローンを検出することは有益と考えている．しかし，複数の関数にまたがって存在しているということは，関数の外部がギャップになっていることが多いと考えられ，そのようなコードクローンは検出してもあまり意味がないと思われる．
(2) オーバーラップ:　同一クローンセットに含まれるコードクローンがオーバーラップして存在していることを指す．オーバーラップしているので，コピーアンドペーストにより生成されたとは考えにくい．そのため，このようなコードクローンを検出する意味はないと考えている．

表 1　対象ソフトウェア

ソフトウェア			規模	
名前	バージョン	記述言語	ファイル数	総行数
Canna	3.6	C 言語	96	90,326
httpd	2.2.4	C 言語	760	409,489
Ant	1.6.0	Java 言語	627	180,844
JHotDraw	7.0.9	Java 言語	471	90,214

(注4)：トークンとは，字句解析により分割されたソースコードを構成する要素を表す．

(3) 調査の必要がない:　コードクローンの中には，ソフトウェア開発・保守を行う視点でそれらを扱う場合に特に対象とする必要が無いものが存在する．これらのコードクローンの多くは，連続した変数宣言やメソッド呼出，switch 文の連続した case エントリなど，繰り返し構造をもつことがわかっている [4]．この実験では，文献 [4] で紹介されている，繰り返し構造をもつコードクローンを，調査の必要がないコードクローンとした．
(4) 有益:　この実験では，関数内にギャップが存在し，同一クローンセットに属する他の全てのコードクローンとも重複しておらず，文献 [4] で紹介されている繰り返し構造を持っていない場合に，そのギャップを含むコードクローンは検出を行う価値があると判断し，有益とした．

5.2　調査結果

対象ソフトウェアから検出されたギャップを含むコードクローンのソースコードを全て閲覧し，各基準に該当するクローンセットの数を計測した．表 2 に検出されたギャップを含むクローンセットの総数および各基準に該当した個数を示す．この表から，各ソフトウェアから検出されたソフトウェアのうち，およそ 25%~35%のギャップを含むクローンセットが有益であると判定されたことがわかる．

有益と判定されたギャップを含むコードクローンの例を示す．図 5 は Canna から検出されたコードクローンである(注5)．コード片 1(図 5(a)) は日本語の変換処理対象領域を拡大する処理を実装しており，コード片 2(図 5(b)) は，変換処理対象領域を縮小する処理を実装している．縮小処理のコード片には，さらに縮小化可能かを判定する処理が含まれており，この部分が拡大処理のコード片とのギャップになっている (図 5(b) の行頭が G の部分)．ソースコード中のコメントから，開発者はこの処理の正しさについて確信がないことがわかる．ギャップを含むコードクローンを検出することにより，このような調査を行う必要がある部分を発見することができた．

全ての対象ソフトウェアからはかなりの数の，「(1) 複数関数」，「(2) オーバーラップ」，「(3) 調査の必要がない」，の基準に該当するクローンセットが検出されており，(4) 有益に該当するクローンセットの全体に対する割合を下げていることがわかる．なお，1 つのコードクローンには，2 つ以上のギャップが含まれていることがあり，このような場合は複数の基準に該当することがある．例えば，2 つのギャップを含んでいるコードクロー

表 2　検出されたギャップを含むクローンセットの内訳

ソフトウェア	ギャップを含むクローンセットの数				
	合計	(1) 複	(2) オ	(3) 調	(4) 有
Canna	208	130(62.5%)	21(10.1%)	20(9.6%)	75(36.1%)
httpd	275	161(58.5%)	29(10.5%)	21(7.6%)	98(35.6%)
Ant	113	73(64.6%)	23(20.4%)	13(11.5%)	30(26.5%)
JHotDraw	257	155(60.3%)	22(8.6%)	22(8.6%)	87(33.9%)

(注5)：コード片 1 およびコード片 2 の文字列リテラル"文字列 1"および"文字列 2"は，実際にはもっと長い文字列であるが，ソースコードをよりわかりやすく掲載するために置換している．

```
    int i;

    d->nbytes = 0;
    yc->kouhoCount = 0;

    if (yc->ys < yc->kEndp || yc->ye != yc->kEndp) {
        i = yc->curbun; /* とっとく */
        if (chikuji_subst_yomi(d) == -1) {
            makeGLineMessageFromString(d, jrKanjiError);
            return TanMuhenkan(d);
        }
        if (RkwGoTo(yc->context, i) == -1) {
            (void)makeRkError(d, "文字列1", "文字列2");
            /* 文節の移動に失敗しました */
            return TanMuhenkan(d);
        }
        yc->curbun = i;
    }
    if ((yc->nbunsetsu = RkwEnlarge(yc->context)) <= 0) {
        (void)makeRkError(d, "文字列1", "文字列2");
        /* 文節の拡大に失敗しました */
        return TanMuhenkan(d);
    }
    if (chikuji_restore_yomi(d) == NG) {
        return TanMuhenkan(d);
    }
    yc->status |= CHIKUJI_OVERWRAP;
    makeKanjiStatusReturn(d, yc);
    return d->nbytes;
```

(a) コード片 1(変換対象領域の拡大処理)

```
    int i;

    d->nbytes = 0;
    yc->kouhoCount = 0;
    if (yc->ys < yc->kEndp || yc->ye != yc->kEndp) {
        i = yc->curbun;
        if (chikuji_subst_yomi(d) == -1) {
            makeGLineMessageFromString(d, jrKanjiError);
            return TanMuhenkan(d);
        }
        if (RkwGoTo(yc->context, i) == -1) {
            (void)makeRkError(d, "文字列1", "文字列2");
            /* 文節の縮小に失敗しました */
            return TanMuhenkan(d);
        }
        yc->curbun = i;
    }

G   if (RkwGetStat(yc->context, &stat) < 0 || stat.ylen == 1) {
G       /* これ以上短くできるかどうか確認。要る？ */
G       return NothingForGLine(d);
G   }
    yc->nbunsetsu = RkwShorten(yc->context);
    if (yc->nbunsetsu <= 0) { /* 0 ってことあんのかなあ？ */
        (void)makeRkError(d, "文字列1", "文字列2");
        /* 文節の縮小に失敗しました */
        return TanMuhenkan(d);
    }
    if (chikuji_restore_yomi(d) == NG) {
        return TanMuhenkan(d);
    }
    yc->status |= CHIKUJI_OVERWRAP;
    makeKanjiStatusReturn(d, yc);
    return d->nbytes;
```

(b) コード片 2(変換対象領域の縮小処理)

図 5 Canna から検出されたギャップを含むコードクローン

ンがあり，そのうちの 1 つが関数の外側，残りの 1 つが関数の内側にあった場合は，そのコードクローンを含むクローンセットは，「(1) 複数関数」と (4) 有益の両方の基準に該当する．

6. 考　　察

本節では，「(1) 複数関数」，「(2) オーバーラップ」，「(3) 調査の必要がない」に該当するコードクローンが検出された原因を述べ，さらにそれらの検出を抑える対策について考察する．

6.1 「(1) 複数関数」への対策

実験では，有益と判定されたクローンセットは，全体の約 25%〜35%であり，高い割合とはいえない．その大きな原因は，「(1) 複数関数」に該当するクローンセットが非常に多く検出されたことである．いずれの対象ソフトウェアも，処理内容が類似した関数が多く定義されており，それらは連続した位置にあった．それらの開始部分と終了部分が共通で (完全一致クローンもしくは名前変更クローンとなっており)，中央のみが異なっているために，ある関数の終了部分から次の関数の開始部分までが 1 つのギャップを含むコードクローンとして検出されていた．

複数の関数にまたがるコードクローンの検出を防ぐには，各関数の開始位置および終了位置を計測し，ギャップを含むコードクローンがそれを跨ぐかどうかを判定しなければならない．しかし，コードクローン検出結果には対象ソースコードに含まれる関数の開始位置および終了位置の情報は含まれていないために，ポストプロセッサ内で再度ソースコード分析を行わなければならない．

ポストプロセッサでソースコード分析を行わない利点は，用いるコードクローン分析ツールが対応しているプログラミング言語であれば，どの言語であってもポストプロセッサを適用できることである．つまり，ポストプロセッサ内でソースコード分析を行うことによって，「(1) 複数関数」に該当するクローンセットの検出を防ぐことは可能であるが，分析対象ソフトウェアのソースコード分析器を構築する必要があり，複数のプログラミング言語を対象とするような場合には，ポストプロセッサの実装に高いコストを必要とする．ctags [1] などの既存のツールを用いて比較的容易に実装できるとも思われる．

6.2 「(2) オーバーラップ」への対策

全ての対象ソフトウェアにおいて，検出されたクローンセットのうち約 10%はオーバーラップしたコードクローンをもっていた．全ての対象ソフトウェアには，独自の頻出する処理が存在した．頻出する処理は，1 つの if 文の各条件分岐先に存在しており，比較的互いに近い位置に存在していた．

提案手法では，1 つのファイル内に同じクローンセットに含まれるコードクローンが 3 つ存在した場合，1 番目と 2 番目をつないだコードクローンと 2 番目と 3 番目をつないだコードクローンが 1 つのギャップを含むクローンセットを構成し，それらのコード片は共に 2 番目のコード片を含んでいるため，オーバーラップになってしまう．このようにして，オーバーラップしたコードクローンをもつクローンセットが多数検出された．

オーバーラップしたコードクローンの検出を防ぐ手段は 2 つ考えられる．

対策 1 ：ギャップを含むコードクローンを生成した後，それらの位置からオーバーラップしているかどうかの判定を行う．ギャップを含むコードクローンを構成している完全一致クローンおよび名前変更クローンの位置情報は，コードクローン検出ツールの出力結果から得られるため，この処理は容易に行うことが可能である．

対策 2 ：提案手法の手順 2 において，同一クローンセットに

含まれるノード間にはエッジを引かないことにする．この処理も容易に行うことか可能である．

対策1は，全てのオーバーラップしたコードクローンの検出を抑える．それに対して，対策2は，開始コード片と終了コード片が異なるクローンセットに含まれている場合は，ギャップを含むコードクローンとして出力を行う．このようなコードクローンは，巻き付きコードクローン [9] の一種であると考えられる．どちらの対策の方が有効であるかを調査するためには，更なる実験を行う必要がある．

6.3 「(3) 調査の必要がない」への対策

オーバーラップしたクローンセットと同様に，調査の必要がないクローンセットも全体の約 10%を占めていた．本稿では，調査の必要がないコードクローンとは，「ソフトウェア開発・保守を行う視点でそれらを扱う場合に特に対象とする必要が無いコードクローン」と定義しており，具体的には，連続した変数宣言やメソッド呼出，switch 文の連続した case エントリなどの処理を行っているコードクローンである．著者らの過去の調査により，調査の必要がないコードクローンは，繰り返し構造をもつ傾向であることがわかっている [4]．このようなコードクローンは，どのソフトウェアにも存在するため，或る程度検出されてしまうのは当然であるといえる．

このようなコードクローンをフィルタリングするためには，ソースコードの構造を調査しなければならない．例えば，どの部分が連続した変数の定義であるか，どの部分が switch 文の連続した case エントリであるか，ということがわからなければこのようなコードクローンの検出を抑えることはできない．しかし，コードクローン検出ツールの出力結果には，ソースコードの構造情報は含まれていないため，ポストプロセッサでソースコード解析を行う必要がある．6.1 節でも述べたが，ポストプロセッサ内でソースコード分析を行うことによって，「(1) 複数関数」に該当するクローンセットの検出を防ぐことは可能であるが，分析対象ソフトウェアのソースコード分析器を構築する必要があり，適用にはより高いコストを必要とする．

7. 本研究の位置づけ

CP-Miner などの一部の検出手法はギャップを含むコードクローンを検出することができる [7], [10]．本稿で提案する手法は，その他のギャップを含むコードクローンを検出することができない検出手法を拡張するものである．同じ対象からコードクローンを検出する場合でも，用いる検出手法によって検出されるコードクローンは異なる [3]．ギャップを含むコードクローンを検出できない既存の手法と組み合わせることによって，既存のギャップを含むコードクローンを検出する手法とは，異なるギャップを含むコードクローンを検出できると考えられる．

8. ま と め

本稿では，完全一致クローン (タイプ 1) と名前変更クローン (タイプ 2) のコードクローン情報からギャップを含むコードクローン (タイプ 3) の生成手法について述べた．本稿で提案した手法は，コードクローン分析ツールの出力結果のみを用いるため，コードクローン検出対象ソースファイルを直接解析する必要がない．このため，用いるコードクローン検出ツールが対応しているプログラミング言語であれば，どの言語であっても適用することが可能である．

実際に提案手法をコードクローン検出ツール CCFinder のポストプロセッサとして実装し，どのようなギャップを含むコードクローン情報が生成されるのかを調査した．C 言語あるいは Java 言語で記述された 4 つのソフトウェアについて調査したところ，検出結果全体の約 25%~35%が有益なコードクローン情報であった．残りのコードクローンは，なんらかの理由により，検出しても意味がないコードクローンであると思われた．

また，これらの検出の意味がないと思われるコードクローンのフィルタリングの方法についても述べた．今後は，フィルタリング手法の実装と，CCFinder 以外のコードクローン検出ツールを使って，ギャップを含むコードクローン検出を行うことを予定している．

謝辞 本研究を行うにあたりご助力頂いた藤野陽平氏 (現 JR 東日本) に感謝する．本研究は一部，文部科学省リーディングプロジェクト「e-Society 基盤ソフトウェアの総合開発」の委託に基づいて行われた．また，IJARC(マイクロソフト産学連携機構) 第 3 回 CORE プロジェクト，文部科学省 科学研究費補助金 基盤研究 (A)(課題番号:17200001)，および若手研究 (スタートアップ)(課題番号:19800022) の助成を得た．

文　献

[1] ctags. http://ctags.sourceforge.net/.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct 2007.

[3] E. Burd and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36–43, Oct 2002.

[4] 肥後, 吉田, 楠本, 井上. 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, 48(2):811–822, Feb 2007.

[5] 猪口, 鷲尾, 元田. 多頻度グラフマイニング手法の一般化. 人工知能学会論文誌, 19(5):368–378, May 2004.

[6] 井上. エンピリカルソフトウェア工学の研究と実践 —コードクローンを例に—. EASE プロジェクトニュースレター, 4:1–4, Dec 2005.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proc. of the 29th Intenational Conference on Software Engineering*, pp. 96–105, May 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[9] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, Jul 2001.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transaction on Software Engineering*, 32(3):176–192, Mar 2006.

[11] 植田, 神谷, 楠本, 井上. 開発保守支援を目指したコードクローン分析環境. 電子情報通信学会論文誌, 86-D-I(12):863–871, Dec 2003.