サーベイ論文

コードクローン検出とその関連技術

肥後　芳樹[†]　　楠本　真二[†]　　井上　克郎[†]

A Survey of Code Clone Detection and Its Related Techniques

Yoshiki HIGO[†], Shinji KUSUMOTO[†], and Katsuro INOUE[†]

あらまし　近年，コードクローンを対象とした研究が活発に行われている．コードクローンとは，ソースコード中に存在する互いに一致または類似したコード片を指す．コードクローン情報を適切に利用することによって，ソフトウェア保守を効果的にかつ効率的に行うことが可能である．本論文では，現在注目を集めているコードクローンの検出とその関連技術に関して，これまでの研究成果を紹介する．

キーワード　コードクローン，ソフトウェア保守，可視化，リファクタリング

1. まえがき

コードクローンとは，ソースコード中に存在する互いに一致または類似したコード片を指す．一般的に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といわれている．例えば，あるコード片中にバグが存在した場合，そのコード片のすべてのコードクローンに対して，同様のバグの存在が疑われる．対象ソフトウェアが大きい場合，チェックすべき箇所が膨大な数になってしまうこと，及び人間がすべての重複部分を認識しておくことは現実的ではないため，ツールを用いた自動的なコードクローン検出が行われる．

近年，ソフトウェアの分析手法としてコードクローン検出技術が注目を集めている．ソフトウェア工学の著名な会議である ICSE（International Conference on Software Engineering）の 2007 年度大会では，ソフトウェア保守やテスト，アスペクト指向プログラミングなどと並んで，コードクローン専門のセッションが設けられていたことからも，その関心の高さをうかがうことができる [1].

現在，コードクローンに関する様々な研究が行われており，その情報を効率的に収集することは難しい．そこで，本論文では，現在までのコードクローンに関する研究成果を体系的にまとめる．本論文を参考にして，コードクローン検出技術を用いてソフトウェア保守プロセスを改善しようとしている方，これからコードクローンに関する研究を始めようとしている方などが，コードクローンに関する研究成果を学んでいただければ幸いである．

以降，**2.** では，コードクローンの発生理由とその分類について言及し，**3.** では，コードクローンの検出技術とその比較について述べる．**4.** では，コードクローン情報の可視化について研究成果を紹介する．**5.** では，コードクローンを一つの関数などにまとめる（以降，コードクローンの集約と呼ぶ）方法について触れ，**6.** では，集約に向いていないコードクローンの管理方法について述べる．最後に **7.** で本論文をまとめる．

2. コードクローン

コードクローンは，ソフトウェアの保守性を悪化させる一つの要因であると考えられている．門田らは，COBOL で記述されたソフトウェアに対してコードクローンとソースファイルの改版数の関係を調査している [2]．その調査では，全体の 80%以上がコードクローンになっているソースファイルや，200 行以上のコードクローンを含むソースファイルは他のソースファイルに比べ，改版数が多くなる傾向であることが報告されている．

その一方で，コードクローンを用いた開発が望ましい場合もあるとの報告もされている．例えば，Kapser らは，ハードウェアドライバを作成する場合は，既存のドライバからのコピーアンドペーストが有効である

[†] 大阪大学大学院情報科学研究科，豊中市
Graduate School of Information Science and Technology, Osaka University, 1–3 Machikaneyama-cho, Toyonaka-shi, 560–8531 Japan

と述べている [3].

コードクローンを把握するための方法としては,

- コードクローン情報の文書化を行うことで変更の一貫性を保つ,
- コードクローンを自動で検出する,

の二つがある [4]. しかし，コードクローン情報の文書化には，すべてのコードクローンに対する情報を常に最新に保つことに非常に手間がかかるため，現実的には困難である．そこで，これまでに様々なコードクローン検出手法が提案されている．

2.1 発生理由

コードクローンがソースコード中に生成される原因として，下記の項目が挙げられる [5]〜[7].

[既存コードのコピーアンドペーストによる再利用]

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，一からコードを書くよりも既存コードを流用して部分的な変更を加える方が信頼性が高いということもあり，実際にはコピーアンドペーストによる既存コードの再利用が多く存在する．

[定型処理]

定義上簡単で頻繁に用いられる処理はコードクローンになる傾向がある．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などがそうである．

[プログラミング言語における適切な機能の欠如]

抽象データ型や，ローカル変数を用いることができない場合には，同じようなアルゴリズムをもつ処理を繰り返し書かなくてはならない場合がある．

[パフォーマンス改善]

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し記述することにより，パフォーマンスの改善を図ることがある．

[コード生成ツールの生成コード]

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

[複数のプラットホームに対応したコード]

複数の OS（Linux，FreeBSD，HP-UX や AIX など）や CPU（i386 系，amd64 系，alpha や sparc64 など）に対応したソフトウェアは，各プラットホーム用のコード部分に重複した処理が存在する傾向が強い．

[偶　　然]

偶然に，開発者が同一のコードを書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.2 定　　義

これまでに，様々なコードクローン検出手法が提案されており，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に基づき，それらを三つに分類している [8], [9]．以下に各分類の概要を示す．

（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）

タイプ 2 における変更に加えて，文の挿入や削除，変更が行われたコードクローンを指す．

3. コードクローン検出技術

本章では，これまでに提案されているコードクローン検出技術を紹介する．既存の検出技術は，コードクローンをどの単位で検出するのかによって，大まかに以下の五つに分類することができる．

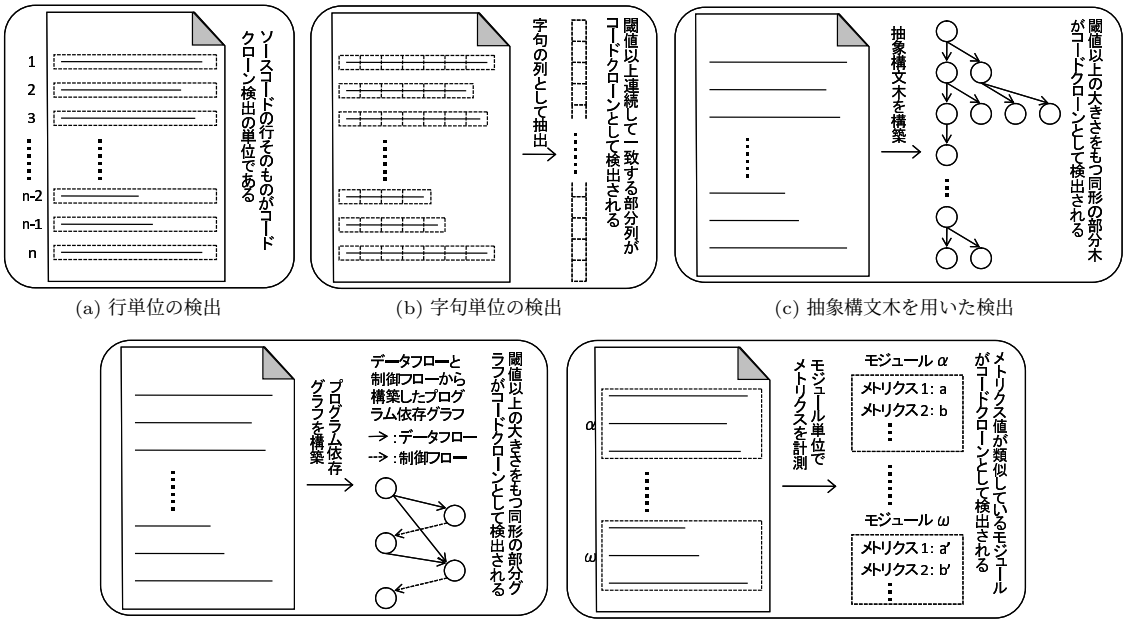
- 行単位の検出
- 字句単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリックスやフィンガープリントなど，その他の技術を用いた検出

各分類に属する検出技術がどの単位でコードクローンを検出するのかを図 1 に表す．また，Bellon の分類 [8], [9] を用いて，各検出技術がどのようなコードクローンを検出できるかを表 1 にまとめる．

以降，**3.1**〜**3.5** では各分類に属する検出技術を紹介し，**3.6** では検出技術の比較について述べ，**3.7** では検出に影響を与える要因について論ずる．

3.1 行単位の検出

ソースコードを行単位で比較することにより重複コー



(a) 行単位の検出　(b) 字句単位の検出　(c) 抽象構文木を用いた検出

(d) プログラム依存グラフを用いた検出　(d) メトリックスを用いた検出

図 1　コードクローン検出の前処理
Fig. 1　preprocesses of code clone detection.

表 1　コードクローン検出技術とコードクローンタイプ
Table 1　Code clone detection techniques and code clone types.

検出技術の分類	検出手法	検出可能なコードクローン			対応言語
		タイプ 1	タイプ 2	タイプ 3	
行単位の検出	Johnson の手法 [10]	○	-	-	不特定多数
	Ducasse らの手法 [11], [12]	○	-	-	不特定多数
	Baker の手法 [13]〜[15]	○	○	-	不明
	Wettel ら手法 [16]	○	-	○	不特定多数
字句単位の検出	Prechelt らの手法 [17]〜[19]	○	○	-	C/C++，Java，C#,Scheme
	Kamiya らの手法 [6]	○	○	△(注1)	C/C++，Java，COBOL など
	Li らの手法 [20]	○	○	○	C/C++
	Basit らの手法 [21]	○	○	○	Java
抽象構文木を用いた検出	Baxter らの手法 [5]	○	○	-	C/C++，Java，Ada，C#など
	Koschke らの手法 [22]	○	○	-	C
	Jiang らの手法 [23]	○	○	○	C,Java
	Wahler らの手法 [24]	○	○	-	XML で表現されたソースコード
プログラム依存グラフを用いた検出	Komondoor らの手法 [25]	○	○	○	C/C++，Scheme
	Krinke の手法 [26]	○	○	○	C
メトリックスを用いた検出	Mayrand らの手法 [27]	○	○	○	C/C++
	Kontogiannis らの手法 [28], [29]	○	○	○	C

ドを特定するのが行単位の検出手法である（図 1 (a)）．行単位の検出手法では，しきい値以上連続して重複している行がコードクローンとして検出される．この検出技術は，他の検出技術に比べ検出速度が高速であるという特徴をもつが，同じ処理を行っているコードであってもコーディングスタイルの違う場合はコードクローンとして検出できないという弱点をもつ．

Johnson や Ducasse らは，比較的単純な検出手法を提案している [10]〜[12]．この手法では，各行に含まれる空白やタブが取り除かれた後，すべての行を比較して重複した行を検出している．各行に対して行われる処理は，空白とタブの削除のみであるため，不特定多数のプログラミング言語に対して適用することが

(注1)：Ueda らの手法 [30] を併用することによってタイプ 3 のコードクローンも検出することが可能．Kamiya らの手法のみを用いる場合は，タイプ 1 とタイプ 2 のコードクローンのみを検出できる．

可能である．

Baker は Parameterized-Matching を用いた手法を提案している [13]〜[15]．この手法では，比較が行われる前に，各行に含まれるユーザ定義名部分が特殊文字に置き換えられるため，ユーザ定義名が異なっていてもコードクローンとして検出することができる．すべてのユーザ定義名が同一の特殊文字に置き換えられるのではなく，同一変数を使用している部分が，同一特殊文字に置換される．これにより，誤検出を減らすことに成功している．行単位での比較には，接尾辞木(注2)検索アルゴリズム [31] が用いられているため，線形時間でコードクローンを検出することができる．

Wettel らは，行単位でコードクローンを検出した後，近接するコードクローンを一つにまとめる手法を提案している [16]．この処理を行うことにより，小さいコードクローンを大きな一つのコードクローンとしてとらえることができる．つまり，コピーアンドペースト後に文の挿入や削除が行われた部分を一つのコードクローンとして検出することができる．

3.2 字句単位の検出

字句単位のコードクローン検出では，検出の前処理としてソースコードは字句の列に変換される（図 1 (b)）．しきい値以上連続して一致している字句の部分列がコードクローンとして検出される．字句単位のコードクローン検出は，行単位の検出のように検出結果がコーディングスタイルに依存することはない．また，ソースコードを検出用の中間表現に変換する必要もないため，高速にコードクローンを検出できるという利点もある．

Prechelt らは，剽窃の特定を目的としたコードクローン検出ツール JPlag を開発している [17]〜[19]．JPlag は，C/C++，C#，Java，Scheme に対応しており，入力されたソースコードは JPlag 独自のトークン列に変換される．図 2 は，JPlag によるソースコード変換の例である(注3)．メソッドの開始は BEGINMETHOD，変数の宣言は VARDEF などのように，ソースコードが JPlag 独自のトークン列に変換されているのが分かる．コードクローンは，貪欲法を用いることによって変換後のトークン列から検出される．

Kamiya らは，接尾辞木検索アルゴリズム [31] を用いた検出手法を提案し，検出ツール CCFinder を開発している [6]．検出処理の前に，ユーザ定義名を特殊文字に置き換えるという言語依存の処理を必要とするにもかかわらず，C/C++，Java，COBOL，FORTRAN など広く用いられている複数のプログラミング言語に対応している．現在，同氏による後継機 CCFinderX の開発も進められている [32]．

```
 1  public class Count {
 2    public static void main(String[] args)
 3          throws java.io.IOException {
 4      int count = 0;
 5
 6      while (System.in.read() != -1)
 7        count++;
 8      System.out.println(count+" chars.");
 9    }
10  }
```

(a) 変換前のソースコード（Java）

```
 1  BEGINCLASS
 2  VARDEF,BEGENMETHOD
 3
 4  VARDEF,ASSIGN
 5
 6  APPLY,BEGINWHILE
 7  ASSIGN,ENDWHILE
 8  APPLY
 9  ENDMETHOD
10  ENDCLASS
```

(b) JPlag が変換したトークン列

図 2 JPlag のトークン変換
Fig. 2 Tokens transformation on JPlag.

Ueda らは，検出ツール CCFinder のフロントエンドである Gemini を改良し，CCFinder が検出したタイプ 1 及びタイプ 2 のコードクローンからタイプ 3 のコードクローン情報を生成している [30]．しきい値よりも近くに存在するタイプ 1 及びタイプ 2 のコードクローンをつなぎ合わせ，タイプ 3 のコードクローン情報としている．生成されたタイプ 3 のコードクローン情報は，Gemini の散布図(注4)などに反映される．

Li らは，頻出系列マイニング(注5)アルゴリズム [33] を用いた検出手法を提案し，検出ツール CP-Miner を開発している [20]．この手法では，まず，ソースコードに対して字句解析及び構文解析が行われ，ユーザ定義名は特殊文字に置換される．その後，文ごとにハッシュ値に置き換えられる．つまり，用いている変数名が異なっていても同一構造をもつ文であれば，同一ハッシュ値をもつことになる．この操作を行うことにより，文の並びはハッシュ値の列に変換される．この

(注2)：*Suffix Tree*
(注3)：図 2 は文献 [18] で用いられている例である．
(注4)：Gemini の散布図については，**4.** を参照されたい．
(注5)：*Frequent Sequence Mining*

列に対して，頻出系列マイニングを適用することにより，コードクローンを検出している．頻出系列マイニングでは，頻出系列は必ずしも連続している必要はない．例えば，系列 *abdec* には，部分系列 *abc* が一つ含まれているとして検出される．このため，タイプ 3 のコードクローンも検出することができる．

Basit らは，接尾辞配列(注6)アルゴリズム [34] を用いた検出手法を提案し，検出ツール Clone Miner を開発している [21]．このツールは，CCFinder と同様，検出処理の前にユーザ定義名を特殊文字に置き換える処理を行う．Clone Miner は接尾辞配列アルゴリズムを用いてコードクローンを検出した後，頻出集合マイニング(注7)アルゴリズム [35] を用いて，同時に出現するコードクローンの集合を特定している．現在の実装では，コードクローンの集合は一つのファイル内に含まれなければならないという制限があるが，拡張することにより，例えば，複数のクラスにまたがる類似部分を検出することができるであろう．

3.3 抽象構文木を用いた検出

抽象構文木を用いた検出では，検出の前処理としてソースコードに対して構文解析を行い，抽象構文木が構築される（図 1 (c)）．抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を用いたコードクローン検出は，字句単位の検出と同様に，検出結果がコーディングスタイルに依存することはない．コードクローンの検出処理を行う前に，抽象構文木を構築する必要があるため，行単位や字句単位の検出に比べて，検出に必要な時間的及び空間的なコストは高くなるが，実用的な検出法として知られている．更に，プログラムの構造から構築した抽象構文木上での一致部分がコードクローンになるため，ある関数定義の終わりから次の関数定義の先頭までの類似部分や，文の途中からの類似部分など，プログラムの構造を無視した類似部分は検出されないこともこの手法の特徴である．このような類似部分がコードクローンとして検出されないことは一概に良い悪いとはいえない．例えば，プログラム全体におけるコードクローンの割合を算出したい場合は，このようなコードクローンも見つけるべきであるかもしれない．しかし，検出したコードクローンを一つの関数や手続きにまとめることを目的としている場合では，プログラムの構造を無視したコードクローンは一つにまとめることは難しいので，検出する意味がないと思われる．

最初に抽象構文木を用いたコードクローン検出手法を提案したのは Baxter らである [5]．抽象構文木を用いた検出では，部分木の対が同一構造になっているかを調査するのであるが，すべての部分木の対を調査することは実用的ではない．そのため，各部分木からハッシュ値を生成し，ハッシュ値が同一の部分木の対のみを比較している．部分木の構造を細部まで忠実に考慮したハッシュ関数ではなく，細部の構造を無視するハッシュ関数を用いているため，ある程度構造が異なる部分木も同一ハッシュ値をもつことになる．この工夫により，大規模ソフトウェアからでも高速にタイプ 1 とタイプ 2 のコードクローンを検出することが可能である．また，Baxter らは，提案手法に基づき，商用のコードクローン検出ツール CloneDR を開発している [36]．このツールは，非常に多くのプログラミング言語に対応しており，コードクローンの位置情報だけでなく，それらを一つの関数としてまとめる場合のひな形も出力する．

Koschke らは，抽象構文木を直列表現に変換した後，接尾辞木アルゴリズムを用いてコードクローン検出を行う手法を提案している [22]．接尾辞木アルゴリズムを使用しているため，高速にコードクローン検出を行うことが可能である．

Jiang らは，抽象構文木の各部分木を配列表現に変換し，局所感度ハッシュ(注8)アルゴリズム [37] を用いて，類似配列を求めることによりコードクローンを検出する手法を提案している [23]．局所感度ハッシュアルゴリズムでは，ある程度配列に違いがあっても同じハッシュ値を割り当てることができる，そのため，タイプ 3 のコードクローンも検出することが可能である．

Wahler らは，XML で表現された抽象構文木からコードクローンを検出する手法を提案している [24]．ソースコードを XML に変換する技術は既に多くのプログラミング言語で開発されており [38]〜[42]，XML からコードクローン検出を行うことは，個々のプログラミング言語のソースコードをそのまま検出対象にするよりは，言語依存の処理を行う部分が少なくて済むため，効率的にツールの開発を行うことができるであろう．Wahler の手法では，文単位で頻出集合マイニングアルゴリズムを用いることによって，コードクローン検出を行っている．

(注6)：*Suffix Array*
(注7)：*Frequent Itemsets Mining*
(注8)：Locality Sensitive Hashing

```
    fp3 = lookaheadset + tokensetsize;
    for (i = lookaheas(state) ; i < k ; i++){
♭♭      fp1 = LA + i * tokensetsize;
♭♭      fp2 = lookaheadset;
♭♭      while (fp2 < fp3)
♭♭          *fp2++ |= fp1++;
    }
```

(a) コード片 1

```
    fp3 = base + tokensetsize;
    ...
    if (rp) {
        while ((j = *rp++) >= 0) {
            ...
♯♯          fp1 = lookaheadset;
♯♯          fp2 = LA + j * tokensetsize;
♯♯          while (fp1 < fp3)
♯♯              *fp1++ |= *fp2++;
        }
    }
```

(b) コード片 2

図 3　順序入れ換わりコードクローン
Fig. 3　Reordered code clone.

3.4　プログラム依存グラフを用いた検出

プログラム依存グラフを用いた検出では，検出の前処理としてソースコードに対して意味解析を行い，ソースコードの要素間（文や式など）の依存関係が抽出され，要素を頂点，依存関係を有向辺とするグラフ（プログラム依存グラフ）が構築される（図 1 (d)）．グラフ上の同形の部分グラフがコードクローンとして検出される．プログラム依存グラフを用いた検出の長所は，順序入れ換わりコードクローン(注9)や巻き付きコードクローン(注10)など，他の技術では検出することのできないコードクローンを検出できることである．図 3 は，順序入れ換わりコードクローンの例を表している(注11)．この例では，図 3 (a) の ♭♭ で始まる行と図 3 (b) の ♯♯ で始まる行がコードクローンになっている．図 3 (a) の一つ目の代入文に対応するのは，図 3 (b) では，二つ目の代入文である．このようにソースコード上で順序が入れ換わっていても意味的に同一であるコード片はコードクローンとして検出することが可能である．また，図 4 は，巻き付きコードクローンを表している．この図でも，♭♭ で始まる行と ♯♯ で始まる行がコードクローンになっている．このように互いに巻き付き合っているコードクローンは，意味的な処理を考慮しなければ検出することはできない．しかし，プログラム依存グラフの構築には高い計算コストを必要とするため，この技術を大規模ソフトウェアに対して適用することは現実的ではない．

```
    ...
♭♭  tmpa = UCHAR(*a),
♯♯  tmpb = UCHAR(*b);
♭♭  while (blanks[tmpa])
♭♭      tmpa = UCHAR(*++a);
♯♯  while (blanks([tmpb])
♯♯      tmpb = UCHAR(*++b);
♭♭  if (tmpa == '-') {
        tmpa = UCHAR(*++a);
        ...
    }
♯♯  else if (tmpb == '-') {
        if (...UCHAR(*++b)...) ...
```

図 4　巻き付きコードクローン
Fig. 4　Intertwined code clone.

Komondoor らは，ソースコード中の文をプログラム依存グラフのノードとする検出手法を提案している [25]．この手法では，まず，ソースコード中の文をその種類に応じて分類する．その後，同種の文の対に対して，フォワードプログラムスライスとバックワードプログラムスライスの両方を用いて，同一のグラフ構造が作成されるかを検査する．しきい値以上の大きさの同一構造のグラフが構築された場合は，そのグラフに含まれる文がコードクローンであるとしている．

Krinke は，変数や演算子などの字句をノードとするプログラム依存グラフからコードクローンを検出する手法を提案している [26]．字句を単位とした場合，文を単位とする場合よりもノードの数が増えるため，すべての部分グラフを調査するには非常に高い計算コストを必要とする．そのため，しきい値を用いて部分グラフを構築する範囲を制限している．初期ノードからしきい値以上離れているノードについては，部分グラフの調査を行わない．

3.5　メトリックスやフィンガープリントなどその他の技術を用いた検出

プログラムのモジュール（ファイルや，クラス，メソッドなど）に対してメトリックスを計測し，その値の一致または近似の度合を検査することによって，そのモジュール単位でのコードクローンを検出する手法が，メトリックスを用いたコードクローン検出である（図 1 (e)）．メトリックス値が近似していればコードクローンとして判定されるため，タイプ 3 のコードク

(注9)：reordered code clone
(注10)：intertwined code clone
(注11)：図 3 と図 4 は文献 [25] で用いられている例である．

ローンも検出することが可能である．しかし，サイズの小さいモジュールの場合はメトリックス値に差がでないため誤検出の可能性が高くなり，及びサイズの大きいモジュールの場合はその内部の一部分が類似していてもコードクローンとして判定されない，という問題点がある．

Mayrand らは関数に対して 21 種類のメトリックスを計測することによってコードクローンを検出する手法を提案している [27]．この手法では，ユーザ定義名やコーディングスタイル，制御文などに関するメトリックスを用いて，関数単位で 8 段階の類似度を定義している．

Kontogiannis らは抽象構文木の部分木（ソースコード中の文や式，またはそれらを含むブロック）に対して 5 種類のメトリックスを計測することによってコードクローンを検出する手法を提案している [28], [29]．計測された各部分木のメトリックスは，ユークリッド距離を用いて評価される．しきい値よりも短い距離に位置する部分木がコードクローンとして検出される．

また，プログラムの剽窃を検出することやプログラムの作者を特定することなどを目的とした，フィンガープリントやバースマークを用いた手法も多数存在する．フィンガープリントとは，プログラムに埋め込まれる著作権などに関する情報のことであり，バースマークとは，プログラムにおける固有の特徴を意味する．ソフトウェアが改変されてしまっても，これらの値自体は変わらないため，プログラムの剽窃を発見することができる．これらの技術は，コードクローン検出とは独立に研究されてきている．

Aiken は，プログラムの剽窃を検出するインターネットサービスを行っている [43]．その Web ページには，フィンガープリント技術 [44] を用いているとの記述があるが，詳しい検出手法は公表されていない．

Ottenstein は，プログラム中で用いられている識別子と演算子の種類及び出現回数をバースマークとして用いる手法を提案している [45]．非常に単純な手法であるが，大学生が作成した小規模のプログラムから，学生間のコピーを検出することに成功している．

Wise は，貪欲法を用いたプログラムの剽窃を検出する手法を提案している [46]．この手法は検出の前処理として，対象プログラムを特殊な字句の列に変換する．変換を行うことにより，for 文と while 文間の類似のような，コピーアンドペースト後の簡単な修正により発生してしまったと思われる重複部分を特定することができる．

Krsul らは，メトリックスを用いてプログラムの作者を特定する手法(注12)を提案している [47]．用いているメトリックスは，大まかに以下の 3 種類に分類することができる．

- インデントやコメントなど，プログラムのレイアウトに関するメトリックス．
- 識別子名の長さや大文字の使用頻度など，プログラミングスタイルに関するメトリックス．
- 関数の長さやマクロの使用頻度など，プログラムの構造に関するメトリックス．

この手法の適用実験は，大学の職員や学生など 29 人が作成した 88 個のプログラムを対象として行われ，そのうちの約 7 割について正しく作者を特定することができた．

3.6 手法の比較

本節では，これまでに行われている検出手法の比較について，その概要と結果を述べる．

3.6.1 Bellon らによる比較

これまでに最も大規模な検出手法の比較実験を行っているのは，Bellon らであろう．Bellon らは，Baker の手法 [15]，Baxter らの手法 [5]，Kamiya らの手法 [6]，Krinke の手法 [26]，Mayrand らの手法 [27]，そして Ducasse らの手法 [11] の六つを比較している [9]．

この実験では，ツールを用いてコードクローン検出を行ったのはそのツールの開発者若しくはそのツールを熟知している人物であり，その検出結果が Bellon に集められた．Bellon は集まった約 32 万のコード片のペアのうち，2%のソースコードを閲覧し，それが本当にコードクローンであるかを判断した．Bellon がコードクローンであると判断したコード片のペアの集合(注13)が，コードクローンの正解集合として用いられた(注14)．

コードクローンの検出対象は，C 言語または Java 言語で記述された，約 11,000 行〜235,000 行の八つのオープンソースソフトウェアである．また，対象プログラムに対して各タイプのコードクローンを埋め込み，

(注12)：*Authorship analysis*

(注13)：Bellon は，ツールが検出したコード片そのものをコードクローンとしたわけではない．必要に応じて，その中の一部分のみをコードクローンとしたり，前後のコードと合わせてコードクローンとするなどの処理を行った．

(注14)：Bellon はコードクローン検出ツールの開発者ではないため，Bellon が作成したコードクローンの正解集合はすべての検出ツールに対して中立である，と文献 [9] は主張している．

各ツールが検出することができるかの調査も行った．

実験の結果，Baker の手法，Ducasse らの手法，Kamiya らの手法は，検出結果が類似しており，また，再現率が高いことが分かった．一方，Baxter らの手法，Mayrand らの手法は適合率が高かった．Krinke の手法は，タイプ 3 のコードクローン検出以外では，うまく機能していないことが分かった．

ツールによって検出されたかなり数のコードクローンが，Bellon によって棄却された．例えば，Baxter らの手法によって検出されたコードクローンの 24%，Krinke の手法によって検出されたコードクローンの 77%が，Bellon によって棄却された．

埋め込まれたコードクローンの多くは，ツールによって検出されることはなかった．各ツールは 24〜46%の埋め込まれたコードクローンの検出にとどまっていた．

また，Bellon らは，各ツールの性能面についても言及している(注15)．Baker の手法と Kamiya らの手法は効率的にコードクローンを検出していたと述べている．例えば，postgresql（約 23,5000 行）から，Baker の手法は約 12 秒（メモリ使用量は約 62 MByte），Kamiya らの手法は約 40 秒（メモリ使用量は約 47 MByte）で検出を完了した．最も短時間でコードクローン検出を行っていたのは，Mayrand らの手法である．この手法は各対象の検出処理をすべて 4 秒未満で完了した．一方，Baxter らの手法と Krinke の手法はコードクローン検出に長い時間を必要とした．Baxter らの手法は，snns（約 115,000 行）からのコードクローン検出に約 3 時間（メモリ使用量は 380 MByte）を要した．Krinke の手法 [26] は postgresql（約 235,000 行）からのコードクローン検出に失敗し，cook（約 80,000 行）からのコードクローン検出に 245 時間（メモリ使用量は 12 MByte），snns（約 115,000）からのコードクローン検出に 63 時間（メモリ使用量は 64 MByte）を要した．

3. 6. 2 Burd らによる比較

Burd らは，Kamiya らの手法 [6]，Baxter らの手法 [5]，Mayrand らの手法 [27]，Prechelt らの手法 [18]，そして Aiken の手法 [43] の五つを比較している [48]．すべての手法により検出された各コードクローン(注16)を，手作業で本当にコードクローンであるか調査を行い(注17)，コードクローンであると判定された集合をコードクローンの正解集合として，各検出手法の再現率，適合率を求めている．

表 2 Burd らの比較：各手法により検出されたコードクローンの数，適合率，再現率
Table 2 Comparison by Burd et al.: the numbers of code clones, precisions, and recalls of each detection technique.

検出手法	検出数	適合率	再現率	F 値
Kamiya らの手法 [6]	1,128	72	72	72.0
Baxter らの手法 [5]	84	100	9	16.5
Mayrand らの手法 [27]	278	63	19	29.2
Prechelt らの手法 [18]	131	82	12	20.9
Aiken の手法 [43]	120	73	10	17.6

表 2 は，その結果を表している．この表から分かるように，Kamiya らの手法 [6] は他の手法に比べ，多くのコードクローンを検出しており，検出したコードクローンの約 2/3 は正しくコードクローンであった．その一方で，Baxter らの手法 [5] は，他の手法に比べあまりコードクローンを検出できてはいないが，検出されたコードクローンは，すべて正しくコードクローンであるという結果であった．

検出されるコードクローンは検出ツールの設定によっても異なる．例えば，Kamiya らの手法であれば，検出するコードクローンの最小の長さや，プログラム中の変数名や関数名の違いを考慮するかどうかを，利用者が設定することができる．つまり，同じ手法を用いて検出を行った場合でも，その適合率と再現率は検出の設定によって異なる．Burd らの調査はこの点を考慮していない．より正確なツールの比較を行うためには，様々な設定でコードクローンを検出し分析を行う調査が必要である．

3. 6. 3 Rysselberghe らによる比較

Rysselberghe らは，行単位の検出技術，字句単位の検出技術及びメトリックスを用いた検出技術を比較している [49]．ツールではなく，検出手法の比較を行うために，既存ツールを用いるのではなく，手法ごとに新たにツールを作成して，そのツールを用いて実験を行っている．行単位の検出ツールは，単に空白とタブを取り除いて比較をするツール（以降，行単位の単純検出）と，ユーザ定義名を特殊文字に置換して比較するツール（以降，行単位のパラメータ化検出）を実装している．

(注15)：文献 [9] の実験では，各ツールは異なったハードウェア環境でコードクローン検出を行っていることに注意されたい．
(注16)：コードクローンの検出対象は，約 16,000 行の Java 言語で記述されたソフトウェアである．
(注17)：Bellon らの比較（文献 [9]）と同様に，コードクローンであるかどうかの判断は，筆者らの主観により行われた．

行単位の単純比較は，全くプログラミング言語に依存しないため，コードクローン検出を行う準備が容易であるという利点があると主張している．行単位のパラメータ化検出，及び字句単位の検出は，各プログラミング言語用に解析器を作成する必要があるが，単純な処理しか行っていないため，それほどコストは高くないと述べている．その一方，メトリックスを用いた検出を適用するためには，メトリックスを計測するために様々な情報をソースコードから取得しなければならないため，解析器の作成に高いコストを要すると述べている．

検出結果については，行単位の単純検出によって検出されるコードクローンは，行単位のパラメータ化検出によって検出されるコードクローンの部分集合になる場合が多い，つまりコードクローンには変数名の違いなどがしばしば存在する，と指摘している．また，行単位のパラメータ化検出と字句単位の検出には，あまり検出されるコードクローンに違いはなく，メトリックスを用いた検出は，誤検出が多いと述べている．

また，Rysselberghe らは，検出したコードクローンを集約するという視点から，Ducasse らの手法 [11]，Baker の手法 [14]，Mayrand らの手法 [27]，Kontogiannis らの手法 [28] を比較している [50]．Mayrand らの手法及び Kontogiannis らの手法は，一致または類似しているメソッドやメソッド内のブロックをコードクローンとして検出するため，Ducasse らの手法や Baker の手法に比べ，検出されたコードクローンは集約に適していると述べている．Ducasse らの手法及び Baker の手法は，プログラムの構造は考慮せず，単純に類似した行をコードクローンとして検出するため，検出されたコードクローンは集約には向いていない場合が多いと述べている．しかし，Mayrand らの手法及び Kontogiannis らの手法は，メトリックスを用いた検出技術であるために，**3. 5** の冒頭で述べた問題点があり，検出されたすべてのコードクローンが集約可能というわけではない．

3. 6. 4 Bruntink らによる比較

Bruntink らは，横断的関心事を実装したコードを検出する，という視点から，Kamiya らの手法 [6]，Baxter らの手法 [5]，そして Komondoor らの手法 [25] を比較している [51]．まず，対象システムを熟知している開発者が，手作業により 5 種類の横断的関心事を特定する．その後，各ツールによるコードクローン検出結果と手作業による特定結果の比較を行っている．手法の違いによる検出精度の差はあまりなかったが，横断的関心事による検出精度の差ははっきりと出ていた．具体的には，コードの長さが数個の字句のみから構成されている短い横断的関心事は，誤検出が非常に多いため，検出精度が悪かった．

3. 7 検出結果に影響を与える要因の調査

Baker は，自身が開発したツール Dup を用いて，再現率の視点からコードクローンの検出結果に影響を与える要因について調査を行っている [52]．Baker はコードクローンの正解集合として，**3. 6. 1** で述べた Bellon が作成したコード片のペアの集合を用いている．

調査項目 1：識別子の変更

識別子を特殊文字に置き換えてコードクローン検出を行うことにより，変数名や関数名が異なっていてもコードクローンとして検出をすることができるが，その反面誤検出が増えてしまう．Baker の実験では，識別子の 50%以上が異なる場合はコードクローンとして検出しないフィルタリングについて調査を行っている．

調査項目 2：Parameterized-Matching

3. 1 で述べているように，Baker の手法は，同一変数を使用している部分が，同一特殊文字に置換される．実験では，すべての識別子を同一の特殊文字に置換した場合に再現率がどのように変化するかを調査している．

調査項目 3：ネストの深さ

コードクローンになっている部分の最終行は，先頭行とネストの深さが同じか若しくはより深くなければならない，というのが Bellon らの主張である [9]．Baker はこのフィルタリングの有無についても調査を行った．

調査項目 4：繰返し領域

繰返し領域(注18)のコードクローン(注19)は，ツールにって検出されたコードクローンと一致しない傾向が強い．なぜなら，繰り返し領域のコードクローンの出力形式は，各ツールで大きく異なるためである．ツールの出力方式の違いのために，Bellon が作成した正解集合と一致しないと思われるコードクローンについて調査を行った．

調査項目 5：その他の調査

Bellon が作成したコード片のペアの集合を，Dup が

(注18)：Repetitive Regions：連続した代入文や，連続した if-else 文などを指す．

(注19)：このコードクローンは，Bellon がツールの検出結果をもとに作成したコード片のペアの集合を指す．

表 3 Baker による再現率に関する調査の結果
Table 3 Results of Baker's investigation on recall factors.

調査内容	C 言語	Java 言語
識別子の変更	48% → 56%	52% → 57%
P-Match	56% → 63%	57%で変化なし
ネストの深さ	63% → 69%	57% → 65%
繰り返し領域	69% → 不明	65%で変化なし
その他	不明 → 70%	65%で変化なし

検出できなかったその他の原因について調査を行っている．例えば，Bellon は型名や予約語 super を特殊文字への置換対象としているが Dup はしていないことや，Bellon はコメント行を含めて 6 行以上がコードクローンとしているが，Dup はコメント行はカウントしていないことなどである．その他の原因については，文献 [52] を参照されたい．

表 3 は，上記の調査項目を順に適用したときの再現率の変化を表している．すべての項目を適用した場合，つまり Dup により検出されたコードクローンのフィルタリングを緩くした場合は，C 言語で 70%，Java 言語で 65%まで再現率が上昇している．

Baker は，ツール Dup が誤検出したコードクローン(注20)についてもその原因を調査している．その大きな原因として，Bellon は Java 言語の import 文や C 言語のプリプロセッサ命令はコードクローンとして扱っていないが，Dup はそれらをコードクローンとして検出したことや，Bellon は最終行のネストが先頭行のネストよりも浅いコード片はコードクローンにしていないが，Dup はそのようなコード片もコードクローンとして検出したことを挙げている．

本章では，これまでに提案されているコードクローン検出技術，それらの比較報告，及び検出に影響を与える要因の調査結果について述べたが，すべての面において他の検出技術に勝っているものはない．例えば，プログラム全体におけるコードクローンの割合を算出したい場合は，他のツールよりも漏れなくコードクローンを検出することのできる Kamiya らの手法 [6] を用いる，集約を目的としたコードクローン検出を行う場合は，検出されたコードクローンに対して集約を行いやすい Baxter らの手法 [5] を用いる，などのように，コードクローン情報を扱う状況に応じて，適切な検出技術を選択することが重要である．

4. コードクローンの分析及び可視化

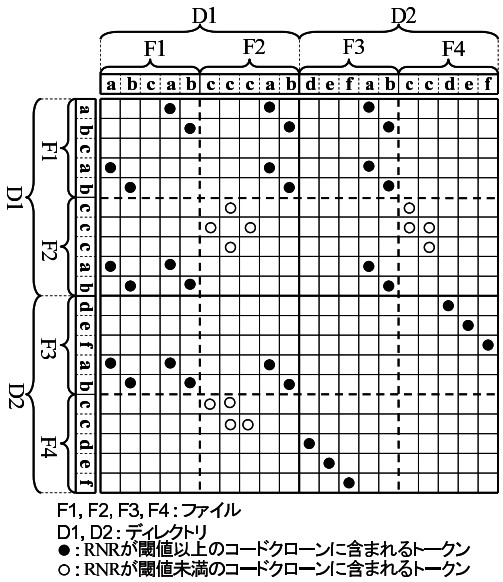
コードクローンの分析及び可視化を行う上で最も障害になるのは，調査の必要がないコードクローンの存在である [53]〜[56]．調査の必要がないコードクローンとは，ソフトウェア開発・保守を行う視点でコードクローン情報を扱う場合に特に対象とする必要がないものである．このようなコードクローンの存在は，調査を必要とするコードクローン情報を隠ぺいし，分析作業の非効率化を招いてしまう．実用的な可視化手法であるためには，調査の必要がないコードクローンのフィルタリングは必須であるといえる．

肥後らは，コードクローンの構造に着目したフィルタリング手法を提案している [53], [54]．彼らは，検出される調査の必要がないコードクローンの多くが，連続した変数宣言やメソッド呼出し，switch 文の連続した case エントリなど，繰り返し構造をもっていることに着目し，コードクローンの非繰り返し度を表すメトリックス RNR を提案している．実験の結果，RNR のしきい値として 0.5 が有効(注21)であることが確認されている．

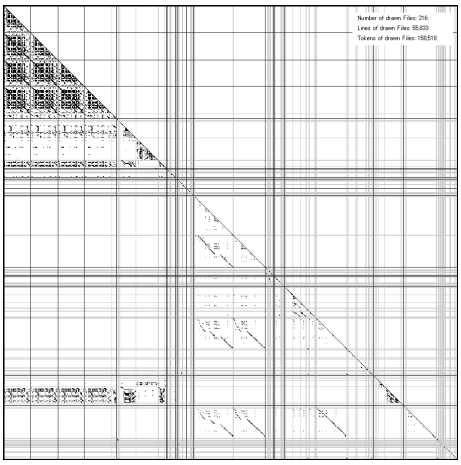
Kapser らは，コードクローンと関数や構造体などのプログラムの要素の対応付けを行い，コードクローン含有率が 6 割以下の要素については，そのコードクローンのフィルタリングを行うことが有効であると報告している [55]．また，対応付けにより，フィルタリングだけでなく，より理解しやすいコードクローン情報が得られると主張している．

最も広く用いられている可視化手法は，散布図であろう．詳細は異なるものの，これまでにいくつかの散布図が提案されている [11], [53], [56], [57]．図 5 (a) は Gemini (注22) [53] の散布図のモデルを表している．散布図では，水平軸と垂直軸に，対象プログラムの要素が，出現順に配置される．そして，水平方向と垂直方向の要素が等しい場合に点が描画される．このモデルでは，連続して 2 字句以上一致している部分をコードクローンとして描画している．また，このモデルは，上述したメトリックス RNR によるフィルタリングの結果を反映している．◯ で示された部分は，それを含むコードクローンが RNR のしきい値以下，つまり，調査の必要がない，と判断されたことを表している．

(注20)：ここで誤検出されたコードクローンとは，ツール Dup によって検出されたが，Bellon の作成したコードクローンの集合には含まれていないコードクローンを指す．
(注21)：繰り返し構造がコードクローン全体の半分以下であることを意味する．
(注22)：Gemini は，文献 [6] で提案及び実装されているコードクローン検出ツール CCFinder のフロントエンドである．

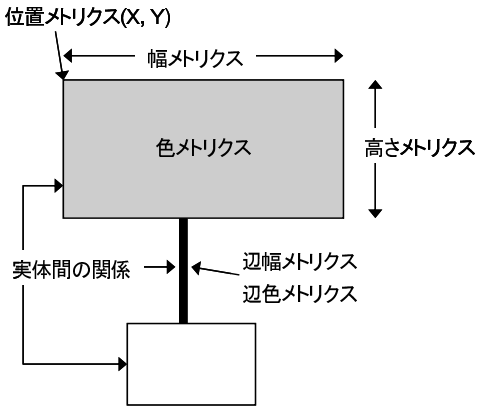


(a) 散布図のモデル

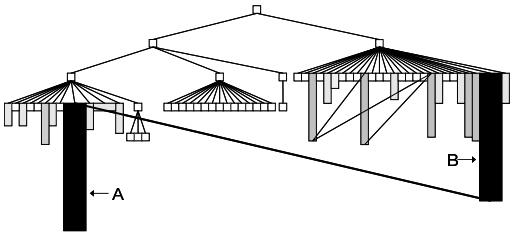


(b) 散布図を用いた可視化の例

図 5　Gemini [53] の散布図
Fig. 5　Scatter plot of Gemini [53].



(a) Polymetric View のモデル



(b) Polymetric View を用いた可視化の例

図 6　Polymetric View
Fig. 6　Polymetric View.

調査の必要がないコードクローンを他のコードクローンと異なって表示することにより，それらを除外して，効率的にコードクローンの分析作業を行うことができる．また，図 5 (b) は，散布図を用いた可視化の例である．ひと目で，左上の部分に非常に多くのコードクローンが存在していることが分かる．

Kapser らは，理解支援を目的としたツール CLICS を開発している [58]．CLICS はソースファイルの構造やシステムのアーキテクチャをコードクローン情報を付与して表示する．CLICS はクエリ処理を実装しており，ユーザは興味のある特徴をクエリとして与えることにより，その条件を満たすコードクローン情報を得ることができる．Kapser らは，散布図はスケーラビリティが高くないため実用的ではない，と述べており，CLICS は散布図を実装していない．また，このツールは上述したコードクローンとプログラムの要素の対応付けによるフィルタリング手法 [55] を実装している．

Rieger らは，Polymetric View [59] を用いたコードクローン情報の可視化手法を提案している [56]．この表現方法は，様々な粒度で抽象化されたコードクローン情報を提供する．図 6 (a) は，Polymetric View のモデルを表している．Polymetric View は，要素の位置や幅，高さなどと，要素のつなぐ辺の幅や色などがそれぞれメトリックス値を表現しており，注目すべき箇所を一見して判断することができる．また，図 6 (b) は，Polymetric View を用いた可視化の一例である．

- 要素は，ソースファイルを表しており，要素の幅メトリックスはその要素内で閉じたコードクローンの量，要素の高さメトリックスは，その要素が他の要素の共有しているコードクローンの量を表している．
- 辺は，その両端のソースファイル間で共有されているコードクローンの量を表している．

この可視化により，図 6 (b) 中のソースファイル A と B は，それぞれ内部で閉じたコードクローンと外部と共有しているコードクローンを多く含んでおり，また両者は多くのコードクローンを共有していることが分かる．Rieger らも，コードクローン検出対象システムが大きい場合には，非常に多くのコードクローンが検出されてしまうため，フィルタリング機構が必要不可欠であると述べている．

Adar らは，複数のバージョンからコードクローンを検出して，その遷移の様子を表示するツール GoftGUESS を開発している [60]．このツールを使うことによって，バージョンを通じて安定しているコードクローンや，異なる修正が加わった結果，コードクローンではなくなったものなど，コードクローンの状態の変化を容易に知ることができる．

Livieri らは，複数のコンピュータ上でコードクローン検出を分散して行うことによって，大規模ソースコード集合からの効率的なコードクローン検出及び可視化を行っている [61], [62]．このようなスケーラビリティの高い手法により，ソフトウェア内ではなくソフトウェア間でのコードクローンを調査することが可能となる．実際，Livieri らは，約 6,700 個（約 4 億行）のソフトウェア間での類似部分を特定することに成功している．また，Livieri らは，このような大規模ソースコード集合からのコードクローン検出が，近年問題になっている著作権違反に応用できる，と主張している．例えば，ソフトウェアライセンスの一つである GPL でライセンスされた著作物は，その派生著作物に対しても GPL でライセンスされなければならない．コードクローン検出を行うことにより，GPL でライセンスされたソフトウェアと他のライセンスをもつソフトウェアが高い類似度であることが判明した場合は，著作権違反の疑いを指摘することができる．

Johnson は HTML を用いたコードクローン情報の巡回手法を提案している [63]．HTML のハイパーリンクを用いることによって，コードクローンを共有しているソースファイル間を自由に巡回することを可能にしている．

5. コードクローンの集約

コードクローンの集約(注23)とは，互いにコードクローンになっているコード片群を一つのモジュール（関数やメソッド，アスペクトなど）にまとめることである．コードクローンを集約することにより，コードクローンによって引き起こされる問題を軽減することができる．リファクタリングの先駆者 Fowler も，“重複コードは最も優先して取り除くべき不吉な匂いの一つ”，と述べている [64]．また，Fowler は，文献 [64] の中で，コードクローンの集約方法についても言及している．例えば，ある一つのクラス内にコードクローンが存在している場合は，重複部分を新たな内部メソッドとして抽出すればよく，同一クラスから派生している複数のクラス間にコードクローンが存在している場合は，重複部分を共通の基底クラスに引き上げることによって，コードクローンを除去することが可能である，と述べている．

Baxter らのツール CloneDr は，コードクローンを検出すると同時に，そのコードクローンを集約するためのひな形も出力する [5]．このひな形を用いることによって，集約を行うにはどのようにソースコードを修正すればよいのかを知ることができる．表 2 に示すように，Baxter らの手法は他の手法に比べて適合率が高いため，この支援はリファクタリングに非常に有効であると思われる．

肥後らは，オブジェクト指向言語で記述されたソフトウェアにおいて，コードクローン間の位置関係をメトリックスとして表現することにより，各コードクローンがどのように集約できるか予測する手法を提案している [65]．肥後らは，まず Kamiya らの手法 [6] を用いてコードクローンを検出し，それに含まれるプログラムの構造的なまとまり（クラスやメソッド，ループなど）を抽出する．そして，抽出した構造的なコードクローン間のクラス階層内における位置関係をメトリックスを用いて表現する．このメトリックスを用いることによって，各コードクローンが，文献 [64] の中で紹介されているどの方法を用いて集約可能であるかを予測する．また，各コードクローンがどの程度その周囲のコードと強く結び付いているのかを計測する．結合が低い場合は，容易にコードクローンを他の部分に移動させることができるが，結合が強い場合は，移動させることは困難である．このような結合の度合もメトリックスとして提供することにより，どの程度容易に集約が行えるかを予測している．

Balazinska らは，メソッド単位のコードクローンに対する集約支援手法を提案している [66]．この手法では，集約の際に重要な情報となるコードクローン間の

(注23)：*merging code clones*，*unifying code clones*

違いを提供する．この違いを用いることによって，各コードクローンを集約するのか容易かどうかを判断することができる．また，肥後らの手法 [65] と同様にコードクローンとその周囲との結合度も解析する．

Jarzabek は，従来のプログラミング言語の抽象化機構ではまとめることが難しい，複数のクラスにまたがるような，大きい単位での類似部分を集約するためのフレームワーク XVCL を提案及び開発している [67], [68]．XVCL を用いることによって，類似クラス群は，メタコンポーネントと呼ばれるモジュールに集約される．メタコンポーネントは，使用されているプログラミング言語による記述と，そのメタコンポーネントがどのようにコンパイル可能なクラスに展開されるかを記述した XVCL の命令文を含んでいる．実際に，XVCL を用いることによって，多くのコードクローンを集約できたとの報告もされている [69], [70]．

6. コードクローンの修正支援

前章では，コードクローンの集約支援手法を紹介したが，すべてのコードクローンが集約可能あるいはすべきというわけではない．川口らや Kim らは，オープンソースソフトウェアの複数のバージョンに対して，コードクローンの出現と消失を調査し，次のことを報告している [71], [72]．

- あるバージョンでコードクローンになったが，その後異なった修正が加えられたことにより，コードクローンでなくなる場合がある．
- 長期間存在するコードクローンは，プログラミング言語に適切な抽象化機構が存在しないなど，集約を行わない理由が存在する．

Kapser らは，コードクローンを集約すべきではない状況をいくつか紹介している [3]．例えば，新しいハードウェアのドライバを作成する場合，既存のドライバから再利用可能な部分をコピーアンドペーストすることが有益であるとしている．既存の正しく動作しているドライバと，新しく作成しているドライバ間のコードクローン部分を集約することは，その際に不具合を混入してしまう危険があるため，そのような集約は行うべきではないとしている．

Balazinska らは，コードクローン間の差異によって，集約の困難さが異なると報告している [73]．Balazinska らの実験結果では，コーディングスタイルなどの表面的な違いのみを含むコードクローンは，利用している変数の型が違うなどの意味的な違いを含むコードクローンに比べ，リファクタリングを容易に行えるという結論であった．

上述のように，すべてのコードクローンが集約に向いているわけではない．Toomim らは，あるコードクローンを修正すると，それと対応するすべてのコードクローンに対して，同様の修正を自動的に施すエディタを開発している [74]．Duala-Ekoko らも同時修正を行うエディタを Eclipse のプラグインとして開発している [75]．しかし，現段階では，これらのエディタはユーザの入力を，対応する各コードクローンに対してそのまま反映させるため，タイプ 2 やタイプ 3 のコードクローンに対しては適用することができない．

泉田らは，CCFinder [6] のオプションを適切に設定することによって，入力コード片と対象ソースコード間のコードクローンのみを高速に検出できると報告している [76]．適切に入力コード片を与えることによって，同時に修正すべき箇所を得ることができる．また，佐々木らも，CCFinder の検出したコードクローン情報をソースコード中にコメントとして付加することによって，同時に修正すべき箇所を容易に特定することができると報告している [77]．

Mann はコピーアンドペーストの履歴を保存することが有益であると主張している [78]．コピーアンドペーストの履歴をたどることにより，任意のコード片の出自を知ることができ，またこれらは同時に修正すべき候補となり得るとも述べている．開発者は 1 時間に約 4 回のコード片単位でのコピーアンドペーストを行う [79]，及びコピーアンドペースト後に，各コード片には異なった修正がしばしば加えられる [80]，との報告もあることから，コピーアンドペーストの履歴を追うことにより，ソースコードのみを用いたコードクローン検出手法では検出することのできない，同時に修正すべき箇所を検出することができると考えられる．

7. むすび

本論文では，コードクローン検出とその関連技術について，これまでの研究成果をまとめた．コードクローン検出技術については，行単位の検出，字句単位の検出，抽象構文木を用いた検出，プログラム依存グラフを用いた検出，メトリックスを用いた検出に分類し，分類ごとに提案されている手法を紹介した．また，各提案手法につき，どのようなコードクローン(注24)を

(注24)：コードクローンの分類については，**2.2** を参照されたい．

検出することができるのかをまとめた．

関連技術については，コードクローン情報の分析と可視化，コードクローンの集約，コードクローンの修正支援に既存の研究成果を分類して紹介した．本論文で紹介した手法を適切に用いることよって，コードクローン情報を，ソフトウェア開発・保守に有効に用いることが可能であろう．

謝辞 本論文は，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研究，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），及び若手研究（スタートアップ）（課題番号：19800022）の助成を得て行われた研究をもとに作成された．

文　献

[1] “The 29th international conference on software engineering,” http://web4.cs.ucl.ac.uk/icse07/

[2] 門田暁人，佐藤慎一，神谷年洋，松本健一，“コードクローンに基づくレガシーソフトウェアの品質の分析，” 情処学論，vol.44, no.8, pp.2178–2187, Aug. 2003.

[3] C. Kapser and M.W. Godfrey, ““Cloning considered harmful” considered harmful,” Proc. 13th Working Conference on Reverse Engineering, pp.19–28, Oct. 2006.

[4] 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, Sept. 2001.

[5] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proc. International Conference on Software Maintenance 98, pp.368–377, March 1998.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, July 2002.

[7] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo, “Software analysis by code clones in open source software,” J. Computer Information Systems, vol.XLV, no.3, pp.1–11, April 2005.

[8] S. Bellon, “Detection of software clones,” Technical Report, Institute for Software Technology, University of Stuttgart, 2003, available at http://www.bauhaus-stuttgart.de/clones/

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” IEEE Trans. Software Engineering, vol.31, no.10, pp.804–818, Oct. 2007.

[10] J.H. Johnson, “Substring matching for clone detection and change tracking,” Proc. International Conference on Software Maintenance 94, pp.120–126, Sept. 1994.

[11] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” Proc. International Conference on Software Maintenance 99, pp.109–118, Aug. 1999.

[12] S. Ducasse, O. Nierstrasz, and M. Rieger, “On the effectiveness of clone detection by string matching,” J. Software Maintenance and Evolution: Research and Practice, vol.18, no.1, pp.37–58, Jan. 2006.

[13] B.S. Baker, “A program for identifying duplicated code,” Proc. 24th Symposium of Computing Science and Statistics, pp.49–57, March 1992.

[14] B.S. Baker, “On finding duplication and near-duplication in large software systems,” Proc. 2nd Working Conference on Reverse Engineering, pp.86–95, July 1995.

[15] B.S. Baker, “Parameterized duplication in strings: algorithms and an application to software maintenance,” SIAM J. Comput., vol.26, no.5, pp.1343–1362, Oct. 1997.

[16] R. Wettel and R. Marinescu, “Archeology of code duplication: Recovering duplication chanins from small duplication fragments,” Proc. 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp.63–70, Sept. 2005.

[17] “Jplag - detecting software plagiarism,” https://www.ipd.uni-karlsruhe.de/jplag/

[18] L. Prechelt, G. Malpohl, and M. Phlippsen, “JPlag: Finding plagiarisms among a set of programs,” Technical Report, University of Karlsruhe, D-76128 Karlsruhe, Germany, March 2000.

[19] L. Prechelt, G. Malpohl, and M. Philippsen, “Finding plagiarisms among a set of programs with JPlag,” J. Universal Computer Science, vol.8, no.11, pp.1016–1038, Nov. 2002.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: Finding copy-paste and related bugs in large-scale software code,” IEEE Trans. Softw. Eng., vol.32, no.3, pp.176–192, March 2006.

[21] H.A. Basit and S. Jarzabek, “Detecting higher-level similarity patterns in programs,” Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.156–165, Sept. 2005.

[22] R. Koschke, R. Falke, and P. Frenzel, “Clone detection using abstract syntax suffix trees,” Proc. 13th Working Conference on Reverse Engineering, pp.253–262, Oct. 2006.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” Proc. 29th Intenational Conference on Software Engineering, pp.96–105, May 2007.

[24] V. Wahler, D. Seipel, J.W.v. Gudenberg, and G. Fischer, “Clone detection in source code by frequent itemset techniques,” Proc. 4th IEEE International Workshop on Source Code Analysis and Manipula-

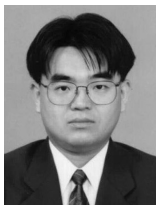
tion, pp.128–135, Sept. 2004.
[25] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” Proc. 8th International Symposium on Static Analysis, pp.40–56, July 2001.
[26] J. Krinke, “Identifying similar code with program dependence graphs,” Proc. 8th Working Conference on Reverse Engineering, pp.301–309, Oct. 2001.
[27] J. Mayrand, C. Leblanc, and E. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” Proc. International Conference on Software Maintenance 96, pp.244–253, Nov. 1996.
[28] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, “Pattern matching for clone and concept detection,” Automated Software Engineering, vol.3, no.1-2, pp.77–108, June 1996.
[29] K. Kontogiannis, “Evaluation experiments on the detection of programming patterns using software metrics,” Proc. 4th Working Conference on Reverse Engineering, pp.44–55, Oct. 1997.
[30] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “On detection of gapped code clones using gap locations,” Proc. 9th Asia-Pacific Software Engineering Conference, pp.327–336, Dec. 2002.
[31] D. Gusfield, Algorithm on Strings, Trees, and Sequences, Campridge University Press, 1997.
[32] “CCFinderX,” http://www.ccfinder.net/
[33] X. Yan, J. Han, and R. Afshar, “CloSpan: Mining closed sequential patterns in large datasets,” Proc. 2003 SIAM International Conference on Data Mining, pp.166–177, May 2003.
[34] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “The enhanced suffix array and its applications to genome analysis,” Proc. 2nd Workshop on Algorithms in Bioinformatics, vol.2542, pp.449–463, Sept. 2002.
[35] G. Grahne and J. Zhu, “Efficiently using prefix-trees in mining frequent itemsets,” Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Nov. 2003.
[36] “CloneDR,” http://www.semdesigns.com/Products/Clone/
[37] M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni, “Locality-sensitive hashing scheme based on p-stable distributions,” Proc. 20th Symposium on Computational Geometry, pp.253–262, June 2004.
[38] G.J. Badros, “JavaML: A markup language for Java source code,” Comput. Netw., vol.33, no.1-6, pp.159–177, 2000.
[39] G. Fischer and J.W.v. Gudenberg, “Simplifying source code analysis by an XML representation,” Softwaretechnik Trends, vol.23, no.2, pp.49–50, May 2003.
[40] “JavaML,” http://www.badros.com/greg/JavaML/
[41] J. Maletic, M. Collard, and A. Marcus, “Source code files as structured documents,” Proc. 10th IEEE International Workshop on Program Comprehension, pp.289–292, June 2002.
[42] D. Seipel, M. Hopfner, and B. Heumesser, “Analyzing and visualizing prolog programs based on XML-representations,” Proc. 13th International Workshop on Logic Programming environments, Dec. 2003.
[43] “Moss — A system for detecting software plagiarism,” http://theory.stanford.edu/~aiken/moss/
[44] S. Schleimer, “Winnowing: Local algorithms for document fingerprinting,” Proc. ACM SIGMOD Conference, pp.76–85, June 2003.
[45] K.J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” ACM SIGCSE Bulletin, vol.8, no.4, pp.30–41, Dec. 1976.
[46] M.J. Wise, “YAP3: Improved detection of similarities in computer program and other texts,” Proc. 27th SIGCSE technical symposium on Computer Science Education, pp.130–134, Feb. 1996.
[47] I. Krsul and E.H. Spafford, “Authorship analysis: Identifying the author of a prgoram,” Comput. Secur., vol.16, no.3, pp.233–257, 1997.
[48] E. Burd and J. Bailey, “Evaluating clone detection tools for use during preventative maintenance,” Proc. 2nd IEEE International Workshop on Source Code Analysis and Manipulation, pp.36–43, Oct. 2002.
[49] F.V. Rysselberghe and S. Demeyer, “Evaluating clone detection techniques,” Proc. International Workshop on Evolution of Large Scale Intdustrial Software Applications, Sept. 2003.
[50] F. Rysselberghe and S. Demeyer, “Evaluating clone detection techniques from a refactoring perspective,” Proc. 19th IEEE International Conference on Automated Software Engineering, pp.336–339, Sept. 2004.
[51] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, “On the use of clone detection for identifying crosscutting concern code,” IEEE Trans. Software Engineering, vol.31, no.10, pp.804–818, Oct. 2005.
[52] B.S. Baker, “Finding clones with dup: Analysis of an experiment,” IEEE Trans. Softw. Eng., vol.33, no.9, pp.608–621, Sept. 2007.
[53] 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，” 情処学論，vol.48, no.2, pp.811–822, Feb. 2007.
[54] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Method and implementation for investigating code clones in a software system,” Information and Software Technology, vol.49, no.9-10, pp.985–998, Sept. 2007.
[55] C. Kapser and M.W. Godfrey, “Aiding comprehension of cloning through categorization,” Proc. 7th International Workshop on Principles of Software Evolution, pp.85–94, Sept. 2004.
[56] M. Rieger, S. Ducasse, and M. Lanza, “Insights into system-wide code duplication,” Proc. 11th Work-

ing Conference on Reverse Engineering, pp.100–109, Nov. 2004.
[57] 植田泰士，神谷年洋，楠本真二，井上克郎，“開発保守支援を目指したコードクローン分析環境,” 信学論（D-I），vol.86-D-I，no.12, pp.863–871, Dec. 2003.
[58] C. Kapser and M.W. Godfrey, “Improved tool support for the investication of duplication in software,” Proc. 21st International Conference on Software Maintenance, pp.305–314, Sept. 2005.
[59] M. Lanza and S. Ducasse, “Polymetric views: A lightweight visual approach to reverse engineering,” IEEE Trans. Softw. Eng., vol.29, no.9, pp.782–795, Sept. 2003.
[60] E. Adar and M. Kim, “Visualization and exploration of code clones in context,” Proc. 29th International Conference on Software Engineering, pp.762–766, May 2007.
[61] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, “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.
[62] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, “Analysis of the linux kernel evolution using code clone coverage,” Proc. 4th Workshop on Mining Software Repositories, pp.22.1–22.4, May 2007.
[63] J.H. Johnson, “Navigating the textual redundancy Web in legacy source,” Proc. 1996 Conference of Centre for Advanced Studies on Collaborative Research, pp.7–16, Nov. 1996.
[64] M. Fowlor, Refactoring: improving the design of existing code, Addison Wesley, 1999.
[65] 肥後芳樹，神谷年洋，楠本真二，井上克郎，“コードクローンを対象としたリファクタリング支援環境,” 信学論（D-I），vol.J88-D-I，no.2, pp.186–195, Feb. 2005.
[66] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, “Advanced clone-analysis to support object-oriented system refactoring,” Proc. 7th IEEE International Working Conference on Reverse Engineering, pp.98–107, Nov. 2000.
[67] “XML-based variant configuration language — Technology for Reuse,” http://xvcl.comp.nus.edu.sg/
[68] S. Jarzabek, Effective Software Maintenance and Evolution: Reused-based Approach, CRC Press Taylor and Francis, 2007.
[69] S. Jarzabek and L. Shubiao, “Eliminating redundancies with a “composition with adaptation” metaprogramming technique,” Proc. ESEC-FSE’03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.237–246, Sept. 2003.
[70] S. Jarzabek and S. Li, “Unifying clones with a generative programming technique: A case study,” J. Software Maintenance and Evolution: Research and Practice, vol.18, no.4, pp.267–292, July 2006.
[71] 川口真司，松下　誠，井上克郎，“版管理システムを用いたクローン履歴分析手法の提案,” 信学論（D），vol.J89-D，no.10, pp.2279–2287, Oct. 2006.
[72] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy, “An empirical study of code clone genealogies,” Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.187–196, Sept. 2005.
[73] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, “Measuring clone based reengineering opportunities,” Proc. 6th IEEE International Symposium on Software Metrics, pp.292–303, Nov. 1999.
[74] M. Toomim, A. Begel, and S. Graham, “Managing duplicated code with linked editing,” Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp.173–180, Sept. 2004.
[75] E. Duala-Ekoko and M.P. Robillard, “Tracking code clones in evolving software,” Proc. 29th International Conference on Software Engineering, pp.158–167, May 2007.
[76] 泉田聡介，植田泰士，神谷年洋，楠本真二，井上克郎，“ソフトウェア保守のための類似コード検索ツール,” 信学論（D-I），vol.86-D-I，no.12, pp.906–908, Dec. 2003.
[77] 佐々木亨，肥後芳樹，神谷年洋，楠本真二，井上克郎，“プログラム変更支援を目的としたコードクローン情報付加ツールの実装と評価,” 信学論（D-I），vol.87-D-I，no.9, pp.868–870, Sept. 2004.
[78] Z.A. Mann, “Three public enemies: Cut, copy, and paste,” Computer, vol.39, no.7, pp.31–35, July 2006.
[79] M. Kim, L. Bergman, T. Lau, and D. Notkin, “An ethnographic study of copy and paste programming practices in OOPL,” Proc. 2004 International Symposium on Empirical Software Engineering, pp.83–92, Aug. 2004.
[80] M. Balint, T. Girba, and R. Marinescu, “How developers copy,” Proc. 14th IEEE International Conference on Program Conprehension, pp.56–68, June 2006.

（平成 19 年 7 月 26 日受付，12 月 23 日再受付）

肥後　芳樹　（正員）

平 14 阪大・基礎工・情報中退．平 18 同大大学院博士後期課程了，平 19 阪大・情報・コンピュータサイエンス・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．情報処理学会，IEEE 各会員．



楠本　真二　（正員）

昭 63 阪大・基礎工・情報卒．平 3 同大学院博士課程中退．同年同大・基礎工・情報・助手．平 8 同大講師．平 11 同大助教授．平 14 阪大・情報・コンピュータサイエンス・助教授．平 17 同学科教授．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．情報処理学会，IEEE，JFPUG 各会員．

井上　克郎　（正員）

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59〜61 ハワイ大マノア校・情報工学科・助教授．平元阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．博士（工学）．平 14 阪大・情報・コンピュータサイエンス・教授．ソフトウェア工学の研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE，ACM，各会員．