チュートリアル

コードクローン検出法

井上 克郎　神谷 年洋　楠本 真二

1 はじめに

コードクローン (code clone) とは，プログラムテキスト中の同一，あるいは，類似したコードの断片を意味する [5]．コードクローンは，「コピーとペースト」によるプログラミングや意図的に同一処理を繰り返して書くことにより，プログラムテキスト中に作りこまれる．レガシーシステムに対する変更や拡張においても，コードクローンは作りこまれることが多い．実際に，約 20 年間保守しながら利用されている，ある大規模ソフトウェアシステム (約 100 万行のサイズであり，2000 個のモジュールから構成されている) では，約半数のモジュールに何らかのコードクローンが存在していることが確認されている [14]．

一般に，コードクローンはプログラムテキストに対する一貫した変更を難しくすると指摘されている [8]．例えば，複数のサブシステムから構成されるソフトウェアシステムを考え，複数のコードクローンがサブシステム上に点在すると仮定する．このとき，フォールトが，あるコードクローン上に発見された場合に，開発者は他の全てのコードクローンを確認して，必要があれば全てのコードクローンに同様の修正を行う必要がある．大規模なシステムは，チームによる開発が通常行われており，上述したように 1 人の開発者がサブシステムを確認し，全てのコードクローンに対して一貫した修正を行うことは極めて困難な作業となる．

また，コードクローンの存在はメトリクスを用いた計測結果にも影響を及ぼす．例えば，保守の際に機能の追加とともにコードクローンの除去を行った場合，追加されたコードよりも除去されたコードのほうが多ければ，機能が増えてコードが小さくなるという，一見矛盾した事態が生じてしまう．

このようなコードクローンによる問題に対処する方法として以下の 2 つが考えられる．

- コードクローン情報の文書化: コードクローンに関する情報が文書化され，継続的に保守されている場合には，コードクローンに対する変更は幾分やさしくなる．しかし，現実的には全てのコードクローンに対する情報を最新のものに保つ作業は非常に面倒な作業で手間のかかる作業であるため，実際の開発現場に導入することは困難である場合が多い．
- コードクローンの自動検出: コードクローンを形式的に定義し，コードクローンをプログラムテキスト中から自動的に検出するための様々な手法が提案され，検出システムが開発されてきている [1] [2] [3] [4] [5] [7] [10] [11] [12] [13]．提案者によってコードクローンの定義が微妙に異なっており，コードクローンの検出手法についても様々なアプローチが提案されている．

本稿では主にコードクローンの自動検出に関する研究

Code-Clone Detection Methods.

Katsuro Inoue, 大阪大学大学院基礎工学研究科, Graduate School of Engineering Science, Osaka University.

Toshihiro Kamiya, 科学技術振興事業団さきがけ研究21, PRESTO, Japan Science and Technology Corporation.

Shinji Kusumoto, 大阪大学大学院基礎工学研究科, Graduate School of Engineering Science, Osaka University

コンピュータソフトウェア, Vol.18, No.5(2001), pp.47–54.

[チュートリアル] 2001 年 5 月 31 日受付.

の現状について述べる．実用的な立場からは，コードクローン検出ツールには以下のような特性が求められており，それらとの関係も議論する．

- 大規模プログラムへの適用可能性: 例えば数百万ステップのプログラムに対して，現実的な時間とメモリの範囲で検出できる．
- コードクローンの識別: 大規模プログラム中には数多くのコードクローンが検出される．従って，何らかの特徴付けを行い，フィルタリングすることが必要である．
- コードクローンの有意性: 現実的に意味を持たないコードクローン，例えば，複数のモジュールにまたがるものや長大なテーブルの初期化などは，取り除くことができる．
- 拡張性: コードクローン検出対象プログラムの言語に深く依存しない．

以降，2章では，文献 [11] に基づいてコードクローンの定義をまとめる．3章ではこれまでに提案されてきている代表的なコードクローン検出手法を紹介する．4章では，実際に実装されているコードクローン検出システムとその適用事例について述べる．5章では，コードクローン検出システムを用いたレガシーシステムの品質評価やその他の研究課題についてまとめる．

2 コードクローン

ある系列中に存在する2つの部分系列 α，β が「等価」であるとき，$C(\alpha,\beta)$ と書き，α は β のクローンであると言う．また，α，β の組をクローンペアと呼ぶ．通常，C は，反射，推移，対称律を満たし，同値関係である．α を含む同値類を α のクローンクラスと言う．

任意の α，β に対して $C(\alpha,\beta)$ ならば，それぞれの部分系列 α', β'(α'$< \bullet\alpha$, β'$< \bullet\beta$ と書く) に対して $C(\alpha$',β') が成り立つ．また，任意の α", β"($\alpha < \bullet\alpha$", $\beta < \bullet\beta$") に対して $C(\alpha$",β") でなく，かつ $C(\alpha,\beta)$ ならば，α，β を極大クローンペアと呼ぶ．本稿では，ある部分系列 α に対し，別の部分系列 β が α と極大クローンペアを構成するとき，α を単に「クローン」と呼ぶ．

系列 S が与えられたとき，S 中の極大クローンペア α，β(ただし $\alpha \neq \beta$) を全て発見することを，クローン発見問題と言う．通常，「クローン検出」，「重複コード発見」と呼ばれるツールは，このクローン発見問題を解くことを目的としている．ただし，ある一定の長さ以上の極大クローンペアのみを出力するようにしているのが普通である．短いクローンは，多数発見されることが多いが，その意味や存在には，興味のない場合が多いからである．

クローン発見問題は，パターンマッチング問題 (pattern matching problem [9]) や局所アラインメント問題 (local alignment problem [9]) とは異なる．パターンマッチング問題は，系列 S の中に，指定したパターン P と等価な部分系列が存在するかどうか，存在するならばその位置を示す問題である．また，局所アラインメント問題は，与えられた2つの系列 S_1，S_2 の中のそれぞれの部分系列 t_1，t_2 のうち等価で最長のものを1つ求める問題である．クローン発見問題では，対象の系列は1つで，その中のクローンを全て求める必要がある．異なる2つの系列の間の極大クローンペアを求める問題は，それらを連接した1つの系列を作ることで，クローン発見問題に帰着される．

ここで議論しているクローン発見問題で対象とする系列は，プログラムテキストである．プログラムテキストの中のクローンを特に「コードクローン」と呼ぶ．しかし，本稿で紹介する各種の手法は，基本的には他の種類の系列にも適用可能である．例えば，HTML や XML などの構造化テキスト，日本語や英語などの自然語文章，DNA や蛋白質配列なども扱うことができよう．ただし，それぞれの対象に応じたチューニングや最適化は必須である．

上述の「等価」に関して，様々な定義を与えることができる．まず，文字列として完全に同じものを等価とすることが考えられる．しかし，そうした場合，空白やコメントの削除，挿入，識別子の名前の変更などが行われた場合，クローンとして検出できなくなる．そこで，多少の違いのある文字列の対も等価とみなすような工夫が必要になってくる．このような等価の判定には，通常，効率の点から文字列レベルの操作ではなく，字句，行，文レベルでの変換操作を用いることが多い．次章でこれらについて詳しく述べる．

```
341:    dwFrameGroupLength = 1;
342:    for(dwCnt = 2; dwCnt <= 64; dwCnt *= 2)
343:    {
344:            if(((ulOutRate / dwCnt) * dwCnt) !=
345:               ulOutRate)
346:            {
347:                    dwFrameGroupLength *= 2;
348:            }
349:    }
```

```
784:    frameGroupLength = 1;
785:    for (cnt = 2; cnt <= 64; cnt *= 2) {
786:            if (((rate / cnt) * cnt) != rate)
787:                    frameGroupLength *= 2;
788:    }
```

図1 変数名等が変更されているコードクローンの例

3 コードクローン検出手法

3.1 検出のための粒度

- 文字

  前述のように，プログラムテキストの構成要素を文字列とみなして，文字列のパターンマッチングを行い，コードクローンを発見することができる．この方法は，対象とするプログラムテキストの記述言語に依存しない．しかし，この方法では，コメント，空白などを含めて，厳密に同形のプログラム断片しかコードクローンとして検出されない．通常，コピーとペースト作業を用いて，コード断片を再利用する場合，そのまま用いることはまれである．現れる変数や関数名を変えたり，パラメータを変更したり，定数値を変えたりすることが多い．また，コーディングスタイルを合わせるために，コメントを追加・削除したり，インデンテーションや改行場所を変えることもある．さらに，文字列のレベルでの等価判定は，手間がかかり，大規模なプログラムの解析には向かない．このような理由で，文字列のレベルでの比較を行ってコードクローンの発見を行うツールはない．

- 字句 (token)

  プログラムの字句解析を行った後，その字句を要素とした系列に対して，クローン発見問題を解く方法を筆者らは提案している [11]．字句解析を行うことで，空白，改行やコメントが削除できる．また，識別子や定数など特定の種類の字句を特殊な1つの字句に固定化することで，変数名や関数名の変更されたプログラム断片もコードクローンとして認識することができる．このように，識別子の種類に応じて字句を固定化することを，「パラメータ化する (parameterize)」と言う．

  図1は，変数名 dwCnt が cnt に変更されているコード断片の例である．この他，後者は改行位置の変更や if 文の後のブロックの消去が行われているが，フィルタリングなど簡単な工夫で両コード断片をクローンとして認識させることができるようになる．

  字句解析の手間は，コンパイラ等とは違い，各種テーブル登録などの必要がないため，軽量なプロセスで実現できる．字句系列に対するクローン発見問題も，後のマッチングアルゴリズムを工夫することで，巨大なプログラムに対しても適用できる．字句解析は対象となるプログラミング言語に依存するが，比較的軽微な手間で構築することができる．

- 行

  さらに粗い粒度にして，プログラムテキストの各行をハッシュ関数を用いて一定の長さの文字列 (や一定の桁数の整数，これらは一般に「ハッシュ値」と呼ばれる) に圧縮し，そのハッシュ値の列を対象として，クローン発見問題を解く方法がある [7]．この方法は，大きなプログラムテキストを比較的小さな要素列に圧縮することができるので，効率良くコードクローンを見つけることができる．ただし，空行の削除や挿入，改行位置の変更によって，コードク

ローンとして検出できなくなる場合があるので，あらかじめ空行の除去，改行位置の整形などをし，検出精度の向上を図るのが普通である．また，行の中の識別子を認識しそれをパラメータ化したうえで，クローン発見を行う手法も実現されている [1]．しかし，字句レベルの抽象化に比べて細かなチューニングは困難である．

- 文
  プログラムテキスト中の文 (statement) を取りだし，それをハッシュ関数で 1 つの要素にし，その系列に対してクローン発見を行う方法もある [15]．この方法では，文の認識のため，簡単な字句解析，構文解析が必要であるが，空白や改行，コメントの挿入などには影響を受けない．名前の付け替えなどに対応するためには識別子などをパラメータ化する必要がある．
- 関数，手続き，クラス定義
  プログラムテキストの関数や手続き，クラス定義全体をそれぞれ 1 つの要素とし，等価な要素対を見つけることでクローン発見を行うこともできる [3] [6]．この方式では，関数，手続き，クラスなどの全体ではなく，一部がコードクローンになっているものを発見することはできない．要素化するためは，通常，プログラムの性質を計測したメトリクス値 (特徴メトリクス) を用いる．関数，手続き，クラスから特徴メトリクス値への変換もハッシュ関数をそれぞれに適用したものと見ることもできる．この方式では，大規模なプログラムもコンパクトな系列に変換でき，効率的であるが，どのような特徴メトリクスを選ぶかの選択は一般に容易ではない．

3.2 クローン発見マッチングアルゴリズム

得られた要素列の中から等価な部分系列を求めるアルゴリズム (マッチングアルゴリズム) には次のようなものがある．これらのマッチングアルゴリズムは自由に選べるものではなく，上記の抽象化レベルによって，適当なものを選択する必要がある．

- 表検索
  対象となる系列の各要素を縦，横に並べた表を作り，縦と横の要素が等しい場合に * を，等しくない場合には空白を埋めていく．図 2 は，あるプログラム断片をトークンレベルに分解し，表にしたものである．この表で右下に下る対角線は，必ず * が現れる．また対角線に関して線対称な表になる．この表のことをクローンのスキャッタプロット (scatter plot) とも言う．クローンは右下に下る線分として現れるので，ある一定の長さ以上のものを発見して出力すればよい．この方法は，系列の長さを n とした場合，時間計算量，空間計算量ともに $O(n^2)$ を要する．系列の要素の種類は有限である必要はない．
- サフィックス木 (suffix tree) 探索
  サフィックス木は，特定の系列 S の全クローン情報を含む構造であり，クローンの発見を効率よく行える [9]．サフィックス木の葉は，系列 $S = (s_1 s_2 s_3 ... s_N)$ の各添え字 $(1...N)$ に対して 1 つずつ存在する．内部頂点はクローンを表現しており，2 つの葉 (から根に至るまでのパス) が共通の内部頂点を持つとき，2 つの葉 (が表現する添え字から始まる部分系列) がクローンであることを意味する．要素の種類が有限であるという制約の下では，サフィックス木を $O(n)$ の時間で構築することができる [9]．また，構築されたサフィックス木からクローンを取り出すには，木の全探索が必要であるため，k を (極大クローンだけではなくすべての) クローンの数として，$O(n+k)$ の時間を要する．
- ソーティング
  メトリクスを用いた抽象化を行った場合，メトリクス値の要素をソートして等価なものを探すことができる．通常，バケツソートを用いて高速に検出するように工夫する．

4 手法の実装と実例

本章では，これまでに開発されてきた様々なコードクローン検出ツールを紹介する．

Dup [1] [2]は Baker により 1992 年に開発されたツールであり，C 言語のプログラムテキストを入力し，行単位の比較によって検出したコードクローンを出力する．プログラムテキストの内部表現として行の並びを用いている．前処理として，ユーザ定義の識別子をパラメータ化する．マッチングアルゴリズムにはサフィックス木探

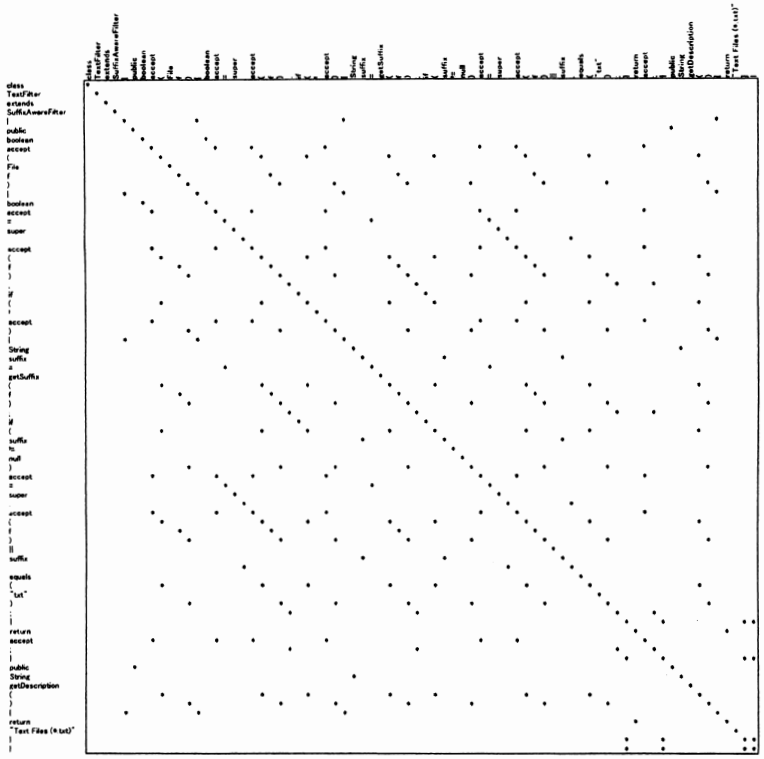

図2 表検索によるクローン検出の例

索を用いているため，計算複雑さは $O(n)$ である．

Duploc [7]は Ducasse らにより 1999 年に開発されたツールであり，検出アルゴリズムとして表検索を用い，計算複雑さは $O(n^2)$ である．特徴は，言語に依存する処理をほとんど行わないため，非常に多くのプログラミング言語に対応することである．実際，Duploc が行う前処理は，空白 (スペース，タブ，改行など) およびコメントを取り除くこと，プログラムテキスト中で非常に多く出現する文字列 (C 言語の場合は compound block の「{}」など) を取り除く処理だけである．Duploc はまた，GUI(graphical user interface) を備えたツールであり，コードクローンのスキャッタプロットを表示し，スキャッタプロットで表示されているコードクローンに対応するプログラムテキストを表示し，編集する機能を持つ．Ducasse らのグループは，プログラミング言語から独立した意味表現 CDIF [16]を用いることにより，多くのプログラミング言語に対応した，クローン検出も含めてより一般的なリファクタリングを目的とするシステム MOOSE の開発を計画している．

CloneDR [5]は Baxter らにより 1998 年に開発された商用のツールであり，C 言語や COBOL で記述されたプログラムテキストから，AST(abstract syntax tree) の節点を比較することによりクローンを検出する．計算複雑さは，入力プログラムテキストから作られる AST の節点のペアそれぞれを比較するため，$O(n^2)$ となる．構文解析によって，プログラミング言語の意味に基づいた処理を行える．例えば，(1)2 つの変数の等価性を，2 つの変数の型が同じかどうかで判断する，(2)C 言語の場合は，マクロ展開を行った後のソースで比較を行う，などである．意味に基づいた処理を行う必要上，プログラミング言語の文法規則に深く依存した意味解析処理ルーチンを用意している．

CloneDR はさらに，発見されたコードクローンを共通ルーチンとして書き換えるための機能を持つ．この機能もやはり，プログラミング言語の言語機能に応じた処理を行う．例えば，C 言語のプログラムテキストに対しては，マクロを生成し，個々のコードクローンをマクロ呼び出しに置き換える．COBOL で記述されたプログラムテキストに対しては，共通ルーチンを生成し，個々のコードクローンをルーチン呼び出しに置き換える．

SMC [3] [4]は Merlo らのグループにおいて 1999 年に開発されたツールであり，まず特徴メトリクスによって，コードクローンと思しきメソッドにあたりをつけ，その後，ローカルな表探索によって，それら疑わしいメ

表 1 コードクローン検出ツールの比較

ツール名	検出方式 (粒度, マッチングアルゴリズム, 計算複雑さ)	スケーラビリティ (プログラムテキストの規模, CPU, メモリ, 検出時間)	プログラミング言語への依存性 { 適用可能な言語 }
Dup [1] [2]	行, サフィックス木, $O(n)$	1.1MSLOC R3000 40MHz, 256MB RAM, 7 分	小 {C}
Duploc [7]	行, 表検索, $O(n^2)$	460KSLOC G3 230MHz, 100MB RAM, 6.5 時間	小 {C, COBOL, Python, Smalltalk}
CCFinder [11]	字句, サフィックス木, $O(n)$	10MSLOC PIII 650MHz, 1GB RAM, 68 分	中 {C, C++, Java, COBOL}
CloneDR [5]	AST の節点, 独自 (本文参照) $O(n^2)$	100KSLOC, (不明), 600MB RAM, 2 時間	大 {C, COBOL, Java}
SMC [3] [4]	メソッド ソーティング及び表検索 $O(n)$	215KSLOC, Pentium-Pro 180MHz, 64MB RAM, (不明)	大 {Java}

ソッドのペアを比較するという, 2 段階の処理を行うツールである. 表探索の計算複雑さは $O(n^2)$ であるが, 特徴メトリクスによって絞込みを行うことにより, 実用上は $O(n)$ の計算複雑さとなる. このツールの最大の特徴は, 検出されたクローンペアのメソッドを, その特徴により 18 種類に分類する機能である. 分類は主に, クローンペアのメソッドの何が異なるか (ローカル変数の名前だけが異なる, 引数が異なるなど) に基づく分類であり, それぞれの分類に対して, どのように共通化メソッドとして書き換えるかについての指針も示されている.

CCFinder [11]は筆者らのグループにおいて 2000 年に開発されたツールであり, 検出アルゴリズムとしてサフィックス木を用い, 計算複雑さは $O(n)$ である. プログラムテキストをトークンの系列として表現し, 簡易な変形ルールを用いることにより, インデントや複文が変更されたプログラムテキストからコードクローンを検出することができる. また, (通常コードクローンとして興味のない) モジュールの先頭によくあるテーブルの初期化文の繰返しなどは検出しない機能を持つ. さらに, 複数のプログラミング言語への対応も実現している. これらのツールを比較したまとめを表 1 に示す. 表には, ツールが用いているアルゴリズムと計算複雑さの他に, スケーラビリティ (現在までにどれくらいの規模のプログラムテキストに適用されたか, そのときに用いられた計算機のスペック, 処理時間) も示されている.

5 保守プロセスにおけるコードクローン利用

保守プロセスにおいてコードクローンを検出する目的としては, (1) リファクタリング [8], (2) プログラムテキスト修正時のチェック, (3) プログラムテキストの評価, の 3 つが提案されている.

5.1 リファクタリング

コードクローンの存在はプログラムテキストの修正をより難しくするため, リファクタリング [8]では, コードクローンを除去することを推奨している. クローンの除去のためには, コードクローンの検出だけではなく, 検出されたクローンを共有ルーチンで書き換える (以下, 「クローンのマージ」) ための手法が必要となる. クローンをマージするための具体的な手法としては, 4

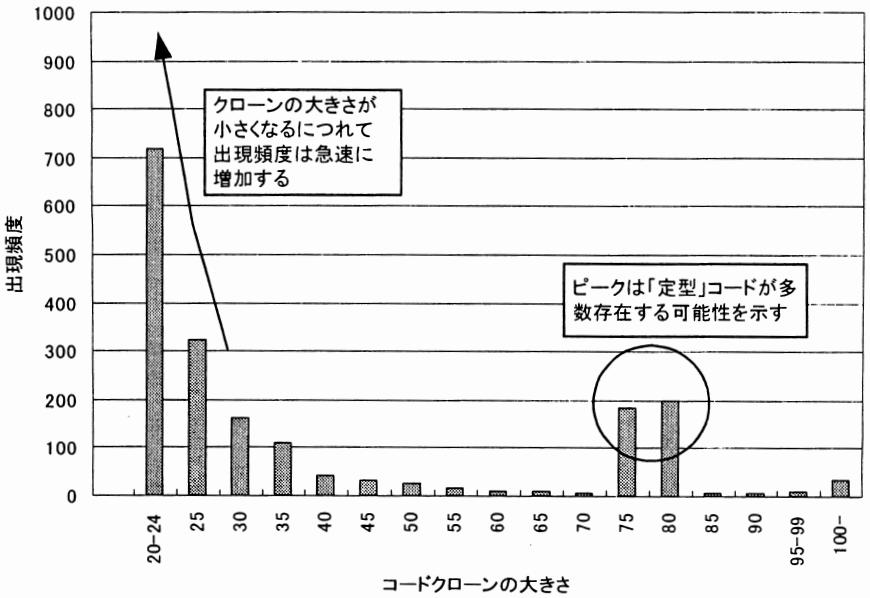

図3 コードクローンの典型的なヒストグラム

章で紹介したCloneDRのようなツールによる共有ルーチンの生成や，SMCのようなクローンの特徴を調べた上での書き換え指針などがある．ただし，いずれの手法においても，マージされたプログラムテキストが書き換え前のプログラムテキストと同じ振る舞いをすることを保証する手法(一種のプログラムの等価性判定問題)は確立されていない．

5.2 プログラムテキスト修正時のチェック

特に大規模なプログラムテキストにおいて何らかの修正を行う際には，コードクローンの存在を確認しておくことが必要である．コードクローンの修正し忘れによって，「直したはずの不具合が直っていない」などという事態が起きる．コードクローンの存在を確認するための手法としては，コードクローン検出ツールを用いる方法以外にも，開発者が事前にコードクローンの存在を文書化しておく方法があるが，文書の更新し忘れや記入漏れなどといった問題に対処する必要がある．

5.3 プログラムテキストの評価用メトリクス

コードクローンの存在はプログラムテキストの修正をより難しくするのであるから，クローンを一種の保守性を表すメトリクスとして用いる立場がある．

中江らは，COBOLで記述されたあるソフトウェアシステムからクローンを検出する実験を行い，そのシステムにおいては，比較的大きなクローンを持つモジュールはフォールトを含む傾向にあることを発見した[14]．

また，プログラムテキストからコードクローンを取り除いた行数(以下，noclone-SLOC)を見積もる手法も予測されている．例えば，コード生成ツールは，一定のテンプレートにパラメータを与えてコードを生成するものがある．そのようなツールによって生成されたコードはコードクローンになる．noclone-SLOCを用いれば，このようなツールによって生成されたコードは取り除かれるため，単なるSLOCよりもよく開発労力を見積もることができるかもしれない．

一般的なプログラムテキストにおけるコードクローンの長さと出現頻度のヒストグラムを図3に示す．コード生成ツールなどで生成されたルーチンなどが多く存在する場合などに，ヒストグラムに特徴のあるピークが現れる．

ヒストグラムにおいて，長さを短くすると非常に多くのコードクローンが発見されるが，そのようなものには単なる偶然の一致が多い．例えば，極端な例ではあるが，検出最小行数を1行に設定すると，ほとんどの代入文がクローンとして検出されてしまう．そのため，

Bakerの研究 [1] ではコードクローンの検出最小行数を30行，Ducasseの研究 [7]では10行としている．マージを目的としてコードクローンを検出する場合には30行程度，比較の目的でコードクローンを検出する場合には10行程度が適当であると思われる．著者らが行った，C言語で記述された3つのOSからクローンを検出する実験においても，検出最小行数を10行程度にしたほうが，OSの類似 (あるいは差異) を数値的により明確に示すことができた．

6 まとめ

以上，コードクローン研究の現状と動向について概観した．コードクローンに関する研究報告は，主に，ソフトウェアメトリクスやソフトウェア保守に関する国際会議 (METRICS (International Symposium on Software Metrics)，ICSM (International Conference on Software Maintenance) 等) や論文誌 (JSME (Journal of Software Maintenance and Evolution) 等) で活発に行われている．より研究が活発化し，理論面・実務面で多くの成果が生まれることを期待したい．

参考文献

[1] Baker, B. S. : A Program for Identifying Duplicated Code, *Proc. of Computing Science and Statistics: 24th Symposium on the Interface*, 24, pp. 49–57, Mar. 1992.

[2] Baker, B. S. : On finding Duplication and Near-Duplication in Large Software System, *Proc. of the Second IEEE Working Conf. on Reverse Eng.*, pp. 86–95, Jul. 1995.

[3] Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K. A. : Measuring Clone Based Reengineering Opportunities, *Proc. of the 6th IEEE Int'l Symposium on Software Metrics (METRICS '99)*, pp. 292–303, Boca Raton, Florida, Nov. 1999.

[4] Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K. A. : Partial Redesign of Java Software Systems Based on Clone Analysis, *Proc. of the 6th IEEE Working Conf. on Reverse Eng. (WCRE '99)*, pp. 326-336, Atlanta, Georgia, Oct. 1999.

[5] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L. : Clone Detection Using Abstract Syntax Trees, *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM '98)*, pp. 368–377, Bethesda, Maryland, Nov. 1998.

[6] Burd, E. and Munro, M. : Evaluating the Evolution of a C Application, *Proc. of ACM SIGSOFT Int'l Workshop on Principles of Software Evolution (IWPSE '99)*, pp. 1–5, Fukuoka, Japan. July 1999.

[7] Ducasse, S., Rieger, M. and Demeyer, S. : A Language Independent Approach for Detecting Duplicated Code, *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '99)*, pp. 109–118, Oxford, England, Aug. 1999.

[8] Fowler, M. : *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (2000) (邦訳: 児玉公信，友野晶夫，平澤章，梅澤真史: リファクタリング：プログラムの体質改善テクニック，株式会社ピアソン・エデュケーション (2000)).

[9] Gusfield, D. : *Algorithms on Strings, Trees, and Sequences*, pp. 89–180, Cambridge University Press, 1997.

[10] Johnson, J. H. : Identifying Redundancy in Source Code Using Fingerprints, *Proc. of IBM Centre for Advanced Studies Conference (CASCON '93)*, pp. 171–183, Toronto, Ontario, Oct. 1993.

[11] Kamiya, T., Kusumoto, S. and Inoue, K. : A Code Clone Detection Technique for Object-Oriented Programming Languages and Its Emprical Evaluation, *Proc. of the 62nd National Convention of IPSJ*, pp. 23–28, 2001.

[12] Kontogiannis, K. A., Mori, R. D., Merlo, E., Galler, M. and Bernstein, M. : Pattern Matching Techniques for Clone Detection and Concept Detection, *Journal of Automated Software Engineering*, Kluwer Academic Publishers, Vol. 3, pp. 770–108, 1996.

[13] Mayland, J., Leblanc, C. and Merlo, E. M. : Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM '96)*, pp. 244–253, Monterey, California, Nov. 1996.

[14] 中江大海，神谷年洋，門田暁人，加藤祐史，佐藤慎一，井上克郎: レガシーソフトウェアを対象とするクローンコードの定量的分析，電子情報通信学会技術研究報告 SS200-49, Vol. 100, No. 570, pp. 57–64, 2001.

[15] Prechelt, L., Malpohl, G. and Phlippsen, M. : JPlag: Finding plagiarisms among a set of programs, Technical Report, Fakultat fur Informatik Universitat Karlsruhe, 2001, http://wwwipd.ira.uka.de/prechelt/Biblio/jplagTR.pdf

[16] The CDIF Standards on the Web, http://www.eigroup.org/cdif/online.html