A New Method to Detect Gapped Code Clones

Yasushi Ueda[†], Toshihiro Kamiya[††], Shinji Kusumoto[†], and Katsuro Inoue[†]

[†] Graduate School of Information Science and Technology, Osaka University, Toyonaka-shi, 560-8531, Japan
[††] PRESTO, Japan Science and Technology Corporation, JAPAN

Abstract

It is generally said that code clone is a factor to make software maintenance difficult. A code clone is the set of code portions in source files that are identical or similar to another. If we make some changes in such a code portion, it might be necessary to apply the same changes in its clones. A code clone introduced by 'copy-and-paste 'reusing tends to have gaps because the developer often modifies the code portion after pasting. This paper proposes a new method to detect clones including gaps. In a case study, we confirm the efficiency of the method.

keyword

Software maintenance, Code clone, Code visualization

論　文

不一致部分を含むコードクローンの抽出手法

植田　泰士[†]　　神谷　年洋[††]　　楠本　真二[†]　　井上　克郎[†]

A New Method to Detect Gapped Code Clones

Yasushi UEDA[†], Toshihiro KAMIYA[††], Shinji KUSUMOTO[†], and Katsuro INOUE[†]

あらまし　ソフトウェアの保守性を阻害する一つの要因として，コードクローンが指摘されている．コードクローンとは，ソースコード中の同一，または類似した部分を指す．例えば，あるコード片に対して何らかの変更（機能追加，バグの修正等）を行う際は，そのコード片のコードクローン全てについて変更の是非を考慮する必要がある．大規模なソフトウェアの場合，それら全ての箇所を手作業で発見し，変更の是非を考慮することは非常に困難である．また，既存コードのコピーとペーストによる再利用によってコードクローンが生成される際には，部分的な変更が行われ，ペーストされたコード片はそのまま利用されることは少なく，コピー元との部分的な不一致が生じることが多い．本研究では，そのような不一致部分を含んだコードクローンを抽出する手法の提案を行う．また，本手法を，コードクローン分析環境 Gemini に実装し，大学におけるプログラミング演習で開発されたプログラムに適用するケーススタディにより，その有効性を確認した．

キーワード　ソフトウェア保守，コードクローン，コード視覚化

1. まえがき

近年，ソフトウェアシステムの大規模化，複雑化に伴い，ソフトウェアの保守・デバッグ作業に要するコストは増大しており，大企業では既存システムの保守に多くのコストを費やすようになってきている．

保守作業を困難にする要因の一つとしてコードクローンが指摘されている [6] [7]．コードクローンとは，ソースコード中の同一，または類似した部分のことであり，「重複コード」とも呼ばれる．コードクローンがソフトウェア中に作り込まれる原因として，既存コードのコピーとペーストによる再利用，頻繁に用いられる定型処理，パフォーマンス改善のための意図的な繰り返し，コード生成ツールによって生成されたコードなどがある [4] [8]．コードクローンの存在が保守作業を困難にするのは，修正されるコード片のコードクローンが存在すれば，その全てのコードクローンに対して修正の是非を検討しなければならないためである．そこで，これまでに様々なコードクローン検出手法やツールが提案されている [1] [2] [3] [4] [5] [9] [10] [11]．

我々も文献 [8] において，新しいコードクローン検出手法を提案している．提案手法に基づいて開発したコードクローン検出ツール CCFinder は，大規模なソフトウェアから実用的な時間でコードクローンを検出できるところに特長がある(注1)．

我々はこれまでに CCFinder を様々なソフトウェア開発組織で運用してきた．その結果，実利用を目指す上での課題が幾つか顕在化した．その課題の一つが，不一致部分を含んだコードクローンを直接的に発見できないことである．実際に，既存コードのコピーとペーストによる再利用の際など，ペーストされたコード片はそのまま利用されることは少なく，ペースト先の文脈に合わせた部分的な修正や変更が行われる可能性が高い．そのような場合，コピー元との不一致が生じるため，CCFinder を用いたクローン検出では，幾つかの相対的に短いクローンに分割され検出されてしまう．一般に，短いコードクローンはソフトウェア中に膨大に存在するため，不一致部分のために分割され

[†] 大阪大学 大学院情報科学研究科，豊中市
Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560-8531 Japan
[††] 科学技術振興事業団 若手個人研究推進事業
PRESTO, Japan Science and Technology Corporation, JAPAN

(注1)：例えば，100 万行規模のソースコードに対して，PC/AT 互換機で，数分から数時間でコードクローンを検出できる

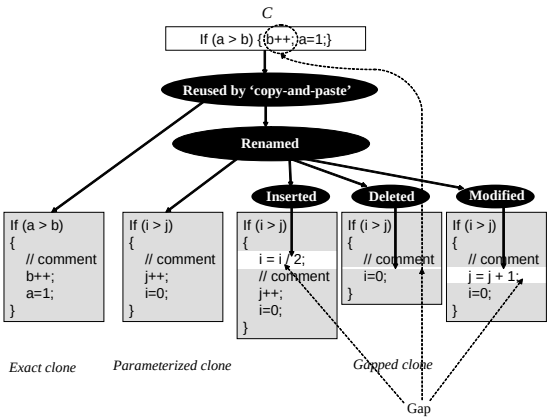

図 1　コードクローンの分類
Fig. 1　Classification of code clones

た短いコードクローンを検出しようとすると，不必要な情報を多く検出してしまうことで分析者の手間が増え，また検出のための計算コストも高くついてしまう．

そこで，本論文では，不一致部分を含んだコードクローンの抽出手法の提案を行う．本手法は，一致箇所を検出した後，連結されるべき不一致箇所を特定し，さらに，マンマシンの共同作業を採り入れることで，低い計算コストでそれらのコードクローンを抽出する．また，本手法を，コードクローン分析環境 Gemini [12] に実装し，大学におけるプログラミング演習で開発されたプログラムに適用しその有用性を評価した．

以降，2. 節では，コードクローンの分類等について，3. 節では，提案手法について述べる．4. 節では，適用実験の結果とその分析，考察について述べ，最後に 5. 節で，まとめと今後の課題を述べる．

2. 準　　備

2.1　コードクローンの分類

一般に，ある与えられたコード片 C に対するコードクローンは，以下の 3 種類に分類される．

Exact クローン (Exact clone):
C に対し，プログラムテキストとして完全に一致したコード片．但し，文字列としての一致とは異なり，空白，改行，コメントなどの違いは考慮しない．

Parameterized クローン (Parametarized clone):
C に対し，変数名，定数名，クラス名，メソッド名等のユーザ定義名の違いを除き一致しているコード片．つまり言語依存の予約語等の構文的な一致を指す．

Gapped クローン (Gapped clone):
C に対し部分的に類似しているコード片．つまり，構文的にも一致しない不一致コード (ギャップ (Gap)) を，部分的に含む．

我々が検出対象にしているクローンの不一致部分 (ギャップ (Gap)) とは，コピーとペーストによるプログラミングにおいて，ペーストされたコード片に含まれる，新規追加されたコード，削除されたコード，または構文が変更されたようなコードを意味する．図 1 に，Exact クローン，Parameterized クローン，Gapped クローンの例を示す．

2.2　CCFinder の検出するコードクローン

CCFinder は単一または複数のソースファイル中から，極大クローンのクローンペアを検出し，その位置情報を出力する．処理は以下の 4 ステップからなる．

ステップ 1(字句解析): ソースファイルを字句解析 (lexical analysis) によってトークン列に変換する．入力ファイルが複数の場合には，それらのファイルの連結した単一のトークン列を生成する．

ステップ 2(変換処理): 実用上意味を持たないコードクローンを取り除くこと，および，些細な表記上の違いを吸収することを目的とした変形ルールによって，トークン列を変換する．例えば，この変換によって変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

ステップ 3(検出処理): トークン列の中から，指定された長さ（最小一致トークン数）以上一致しているような部分クローン列を全て検出する．

ステップ 4(出力整形処理): 検出されたそれぞれのクローンペアについて，元のソースコード上での位置情報を出力する．

以上の検出手順により，CCFinder は Exact クローン，および Parameterized クローン (ステップ 2 の変換処理より) を検出する．以降，これらを併せて，Ng クローン (Ng-clone (Non-gapped clone)) と呼ぶ．

一般に，ステップ 3 における最小一致トークン数を比較的小さな値に定めれば，偶然の一致等により非常に多くのコードクローンが検出されてくる．逆に，大きくすると，分析粒度が粗くなる．既存コードのコピーとペースト，およびその修正によって発生したコードクローンなどは，個々の一致箇所が小さくなるため検出されない場合がある．一方，ある一定の長さのギャップをコードクローンの中に含むことを許容すると，個々の一致箇所が連結され大きなコードクロー

ンとみなすことができ，検出の際，偶然の一致等による単純に小さなコードクローンとは区別可能となる．したがって，Gapped クローンを探すことにより，単純に小さなコードクローンの情報を検出せず，かつ取りこぼしの少ない分析を行うことができる．

3. Gapped クローン抽出手法

仮に Ng クローンの位置情報が存在した場合，本研究でいう Gapped クローンの個々の発見問題は，Ng クローンの組み合わせ決定問題に帰着する．

しかし，複数の Ng クローンが互いに重なりあい密集した際には，あるひとつの Ng クローンからの次の Ng クローンへの結合選択肢が多数存在し，全体としての Gapped クローン検出は，組合せ爆発を起こし，計算コストは膨大になる．経験上，そのように Ng クローンが密集していることは多く，個々の Gapped クローンを全て検出するということは，大規模ソフトウェアには向かない．

一方，実用的な応用としては，個々の Gapped クローンの正確な一致箇所，不一致箇所まで知る必要性はなく，密集範囲全体としてどの辺りが類似しているかが識別できれば十分なことも多い．例えばメソッド等の類似性を判断する場合などである．

そこで我々は Gapped クローンの検出に際し，個々の Gapped クローンを自動的に検出するのではなく，互いに素な Ng クローンの連結集合を検出することによるマンマシン共同作業としてのアプローチをとる．

3.1 抽 出 手 順

図 2 に Gapped クローン抽出プロセスを示す．本プロセスは次の 4 つのステップで構成されている．

ステップ 1: Ng クローン検出 (Ng-clone detection),
ステップ 2: ギャップ検出 (Gap identification),
ステップ 3: 視覚化 (Visualization),
ステップ 4: ソースコード分析 (Source code investigation).

ステップ 1 においては，入力されたソースコードから Ng クローンを検出する．次にステップ 2 で，ステップ 1 で検出された Ng クローンの位置情報からギャップの位置情報を生成する．そしてステップ 3 において，それら Ng クローン位置情報とギャップ位置情報を視覚化する．最後にステップ 4 で分析者が，視覚化された情報を利用して，Gapped クローンのソースコードを分析するという手順である．

以下，例を用いて検出プロセス手順を説明する．例

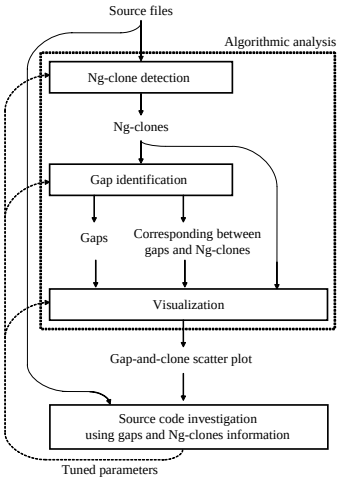

図 2 Gapped クローン抽出プロセス
Fig. 2 Extraction process of gapped clone

として，

コード列 X: “ABCDCDEFBCDG”,
コード列 Y: “ABCEFBCDEBCD”

の 2 つのコード列の比較を考える．ここで，“A”,“B”,“C”,... の記号は，文字，トークン，行，文，関数などプログラムテキスト中の任意の要素を表すことができる．つまり，本節における検出プロセスは，特定の Ng クローン検出アルゴリズムの検出粒度に束縛されることはなく，各記号は，任意の Ng クローン検出ツール毎の検出粒度に対応するものとする．また，我々は複数ファイル間での比較，もしくは単一ファイル内での比較を想定している．以下の説明において，コード列 X, Y が同じ場合も検出プロセスは同様である．

ステップ 1 (Ng クローン検出)
入力ソースコードから Ng クローンを検出し，効率よくギャップ検出を行うための準備として Ng クローンをソーティングする．

例における検出結果を，表 1 に示す．本表には，長さ 2 以上の Ng クローンが含まれている．

Ng クローンの最小一致長を定めなければ，これら全てと，記号単一で一致する部分が検出されることになる．しかし，この例において，たかだか長さ 12 のコード列の中に長さ 2 以上の Ng クローンが 7 つも検出されているように，代入文，変数宣言などの単純な構文により，比較的短いクローンの中には単なる偶然の一致が多い．よって以降，最小一致長 (以降，**閾値**

1) が指定可能であることを前提とする．

この検出の際に決められる閾値 1 は Gapped クローンの中に含まれる，それぞれ一致部分の最小長となる．つまり，最終的に分析を行いたい Gapped クローンの最小長よりも小さい値が指定されなければならない．

また，ステップ 2 においてギャップを効率よく検出するための前準備として，Ng クローンの検出結果に対してソーティングを行う．Ng クローンの 2 つのコード片対それぞれ (以下，コード片 X,Y) は，単一ファイル内で閉じており，複数ファイルをまたぐことがない (Gapped クローンとしても同様である) ので，まずファイルの組み合わせで Ng クローン集合を分割することができる．そして，分割されたそれぞれの集合の中でソーティングを行う．ソーティングに用いられるキーは，優先順位の高いものから，以下の通りとなる．

コード片 X のファイル内での出現位置，

コード片 Y のファイル内での出現位置．

ここでいうファイルは，例においてコード列 X，Y を指す．表 1 では，既にソーティングが完了している．

ステップ 2 (ギャップ検出)

ソーティング済みの Ng クローン情報を元に，ギャップの位置情報を生成する．

図 3 にアルゴリズムの詳細を擬似コードとして示す．各ギャップ毎に生成すべき情報は，ギャップのソースコード上での位置情報，ギャップの両端に連結される 2 つの Ng クローン情報であり，これらはステップ 3 で利用される．表 2 は，表 1 の Ng クローンに対して生成されたギャップ位置を示している (`g1`, ..., `g7`)．

図 3 の中で現れている `threshold2`(以降，**閾値 2**) は Gapped クローン中に含まれることが許容されるそれぞれのギャップの上限長を表している．

また，図 3 の中の最適化では，Ng クローン情報がソート済みであるという事実を利用している．内側

表 1　Ng クローン検出結果 (長さ 2 以上)
Table 1　Detected Ng-clones that are longer than 2

Ng クローン ID	コード列		一致文字列
	X	Y	
c1	1 - 3	1 - 3	"ABC"
c2	2 - 4	6 - 8	"BCD"
c3	2 - 4	10 - 12	"BCD"
c4	5 - 6	11 - 12	"CD"
c5	5 - 7	7 - 9	"CDE"
c6	7 - 11	4 - 8	"EFBCD"
c7	9 - 11	10 - 12	"BCD"

(コード片の位置は記号の開始位置と終了位置を表す．例えば "6 - 8" は，6 番目の記号から 8 番目の記号までを意味する．)

```
// for each NG-clone
for (i = 0; i < ngCloneTotalCount; i++){
  NgClone ci = sortedNgCloneDB.get(i);
  // search connection targets
  for (j = i; j < ngCloneTotalCount; j++){
    NgClone cj = sortedNgCloneDB.get(j);
    // if NG-clone ci and cj are near
    dx = distance(ci.codeInX, cj.codeInX);
    dy = distance(ci.codeInY, cj.codeInY);
    if (((0 <= dx) && (dx < threshold2)) &&
        ((0 <= dy) && (dy < threshold2))){
      // create new gap information
      Gap newGap = new Gap(ci, cj);
      gapDB.add(newGap);
    }
    else{
      // optimization
      if (distance(c.codeInX, d.codeInY) >= threshold2)
        break; // goto next i's loop
    }
  }
}
// Function "distance(x, y)" computes the
// difference from end position of code portion x
// to start position of code portion y.
// The result may be zero or minus when x and y
// are overlapping or when y appears ahead of x.
```

図 3　ギャップ検出アルゴリズム
Fig. 3　Gap identification algorithm

の `for` ループでは，いったん Ng クローン `ci` からの距離が閾値 2 以上の Ng クローンが見つかれば，それ以降は閾値 2 以内の距離にある Ng クローンが見つかることは有り得ない．ここで注意すべきは，もし閾値 2 が入力ソースコード長に対して十分小さく，全 Ng クローンの密集度合いに偏りがなければ，ある Ng クローンから閾値 2 以内の距離にある Ng クローンは，高々定数個以内であると考えられる．したがって，内側の `for` ループは定数時間で終了し，アルゴリズム全体の計算複雑さは，$O(n)$(n: Ng クローン数) である．

また，閾値 2 の値の決定方法については議論の必要があるが，本研究においては定数値として扱う．よっ

表 2　ギャップ位置
Table 2　Gap location

ギャップ ID	コード列		長さ
	X	Y	
g1	4	4 - 6	3
g2	4	4 - 10	7
g3	4 - 6	-	3
g4	4- 8	4 - 9	6
g5	-	9 - 10	2
g6	5 - 8	9	4
g7	8	-	1

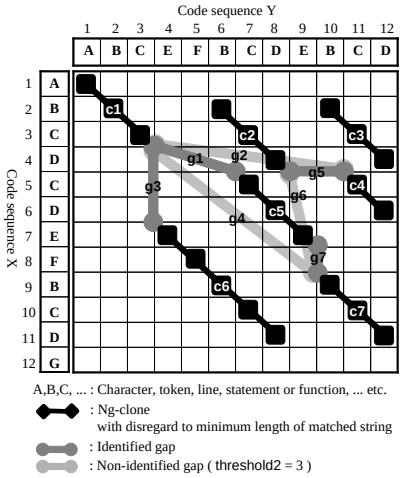

図 4　散布図 (フィルタリング前)
Fig. 4　Scatter plot without filtering

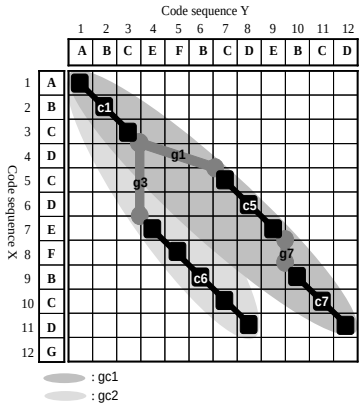

図 5　散布図 (フィルタリング後)
Fig. 5　Scatter plot with filtering

て，閾値 2 は認識するギャップの最大長となっている．

本例においては，閾値 2 は 3 と定める．したがって実際は，g2，g4，g6 は検出されてこない．

ステップ 3　(視覚化)

ステップ 3-1　(散布図)

Ng クローンとギャップを散布図に描画する．

この時点で，Ng クローンとギャップの位置情報を得ることができている．従って，図 4 のように Ng クローンとそれらのギャップをそれぞれ独立に散布図上に描画すれば，擬似的に Gapped クローンの視覚化を行うことができる (以降，本散布図を**Gap-and-clone 散布図**，あるいは，混乱の恐れがない場合は単に散布図と呼ぶ)．

散布図の両座標軸は，各コード列を表しており，左上角が原点になっている．図 4 では，垂直軸がコード片 X，水平軸がコード列 Y を表しており，各 Ng クローンは，両座標軸上で対応した 2 つの要素が一致していることを表している黒い四角形を連結した線分として表されている．また，各ギャップは，両端の Ng クローンを結ぶ灰色の線分として表現されている．つまり，黒い線分と灰色の線分を順に辿ったものが，Gapped クローンを表していることになる．表 3 に，そのパス (辿り方) とその対応コード列を示す．但し，g2，g4，g6 は認識されなかったので，それらを通る辿り方は示さない．

ステップ 3-2　(フィルタリング)

Gapped クローンに関係のない Ng クローンとギャップを取り除く

本ステップは随意的なものであるが，散布図の視認性を改善することができる．

図 4 には，コード片 X，Y から検出され得る全 Ng クローン，全ギャップが描かれている．しかし，先に述べたように大規模ソフトウェアを対象とした場合，短い Ng クローンは非常に数多くの存在するので，散布図上には多くの短い Ng クローンやギャップが存在し，分析が煩雑になる．

閾値 1(Ng クローン最小一致長) と閾値 2(最大ギャップ長) を用いれば，Ng クローンとギャップの数を制御することができる．しかし，これらの閾値では，Gapped クローンとしての長さを制御することはできない．例えば，閾値 1 を 3 にすれば，c4 , g2，g5 を除いた全ての Ng クローンとギャップが現れる．逆にもう少し大きな値，5 にすれば，Ng クローン c6 しか残らない．

そこで，Gapped クローンを散布図上でフィルタリングするため，各 Ng クローンの連結集合の長さを新しいパラメータとして導入する．その長さとは，連結集合中に含まれ得る Gapped クローンの上限長である．

表 3　Gapped クローン
Table 3　Gapped clone path list

ID	パス	コード列 X	コード列 Y
gc1	c1 g1 c5 g7 c7	“ABC-CDE-BCD”	“ABC---CDEBCD”
gc2	c1 g3 c6	“ABC---EFBCD”	“ABCEFBCD”
gc3	c2 g5 c4	“BCDCD”	“BCD--CD”

(“-” はギャップを表している．つまり，それぞれの “-” の数がギャップの長さであり，そのギャップの両端の Ng クローンの距離となる．)

連結集合中に含まれ得る Gapped クローンの上限長の算出手順は，まずギャップ検出の際に生成した，ギャップと Ng クローンの結合情報を用いて Ng クローンを連結集合に分類する．そして，各連結集合 s の中で，各コード片がコード列 X(Y) において最も先頭に近いコード片の開始位置を s の開始位置 sStartX(sStartY) とし，同様に最も先頭から遠いものを s の終了位置 sEndX(sEndY) として求め，次式

sSize = max(sSizeX , sSizeY),
sSizeX = sEndX − sStartX,
sSizeY = sEndY − sStartY

で求まる sSize が連結集合中に含まれ得る Gapped クローンの上限長となる．この長さの下限値を**閾値 3** とし，閾値 3 以上の長さを持つ連結集合のみを表示させることでフィルタリングを実現する．本例において，閾値 3 を 8 とするならば，散布図は図 5 のようになる．

ステップ 4　(ソースコード分析)
各パラメータの値を調整しながら，散布図を利用してソースコードの分析を行う．

最後に，Gapped クローンが擬似的に現れている散布図を用いてユーザが実際に分析を行う．ユーザーインターフェイスを通じて，散布図上で特定の Gapped クローンや Ng クローンを指定し対応するソースコードを参照する，あるいは，選択したソースコードだけを散布図で表示したりできるものとする．

ユーザーは，散布図やソースコードを参照することにより，以下のパラメータを調整したり，あるいは検出対象のソースコードを追加あるいは削除することで，分析を進める．

閾値 1　Ng クローン検出における最小一致長,
閾値 2　ギャップ検出におけるギャップ上限長,
閾値 3　Ng クローン連結集合の下限長.

これらのパラメータは計算コストと完全性のトレードオフである．またその閾値毎にその効果は異なる．例えば，閾値 1 を小さくすると解析時間が増大する．数多く修正が施されたような，それぞれの一致部分が短い Gapped クローンを検出しようと思えば，閾値 1 に小さな値を定めなければならない．しかし，メソッドの抽出のようなリファクタリングなどが目的の場合は，一般的にそのような Gapped クローンを 1 つモジュールにまとめるのは困難であると思われるので，さほど小さな値にする必要はないと考えられる．また，閾値 2 は大きくすると，1 つの Ng クローンから連結される Ng クローンの数が増え，解析時間が増大する．ソースコード長に近づくにつれ，ステップ 2 の計算複雑さは $O(n^2)$(n: Ng クローン数) に近づく．ただし，ソースファイルの境界を越えて Ng クローンを連結することはないという経験則を用いることにより，実用上の問題にはなっていない。

3.2 実　　装

本手法をコードクローン分析環境 Gemini [12] に実装した．Gemini は，CCFinder を内部的に利用し，検出されたコードクローンの位置を示す散布図, 対応ソースコードを参照するビューなどその他様々なビューをユーザに対して提供する．本散布図を Gap-and-clone 散布図として実装することでユーザは，Ng クローン連結集合を選択し対応したソースコードを参照できる．

4. 適 用 実 験

4.1 概　　要

本適用実験においては，大阪大学のプログラミング演習で学生が作成したソースコードを対象とし，Gapped クローンの検出手法の適用を行う．

本プログラミング演習課題は,，Pascal 風言語 (Pascal 言語のサブセット) で記述されたプログラムを CASL(アセンブリ言語) に変換するコンパイラを C 言語で作成することである．また，その作成課題は，次の 3 つのステップ (副課題) で構成されている．

ステップ 1(課題 1): 構文解析器 ($Parser$) の作成.
ステップ 2(課題 2): 意味解析器 ($Checker$) の作成.
ステップ 3(課題 3): コンパイラ (SPC) の作成.

また，$Checker$ と SPC を作成するにあたり，各課題の前課題のプログラムを再利用する（拡張して開発する）よう要求されている．つまり，$Checker$ は $Parser$ を再利用することで，SPC は $Checker$ を再利用することで作成される．したがって，各課題のプログラム間の関係は，本手法で検出することを想定しているコピーとペーストおよびその修正によって書かれたコードと同じと考えることができる．

4.2 分　　析

本手法の有効性を確認するため，閾値 1(Ng クローンの最小一致長) や，閾値 3(Ng クローン連結集合の下限長) を調整することで短い Ng クローンがある程度の大きさを持った Gapped クローンとしてどの程度 Gap-and-clone 散布図に現れてくるのかを確認する．

本適用実験では 69 人のプログラム (計 360KLOC) を対象にしたが，文献 [12] の課題間再利用率評価実験において，その評価が最も高かった学生 (S) を例に

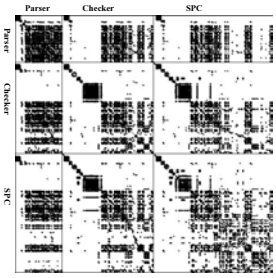

図 6　Ng クローン散布図 (閾値 1 = 10 トークン)
Fig. 6　Ng-clone scatter plot (threshold1 = 10 tokens)

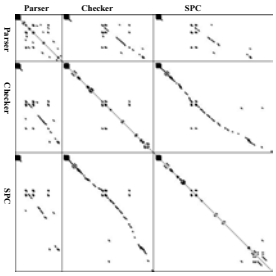

図 7　Ng クローン散布図 (閾値 1 = 30 トークン)
Fig. 7　Ng-clone scatter plot (threshold1 = 30 tokens)

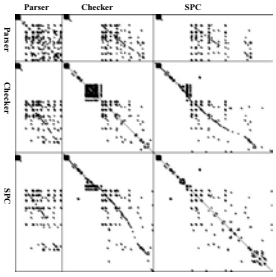

図 8　Gap-and-clone 散布図 (閾値 3 = 30 トークン)
Fig. 8　Gap-and-clone scatter plot (threshold3 = 30 tokens)

とって，その分析結果を示す．

図 6 に S の $Parser$(2267 トークン), $Checker$(4394 トークン)，SPC(5738 トークン) の 3 プログラム間の比較結果の散布図を示す．ここで Ng クローンのみを表示する散布図を，Gap-and-clone 散布図と区別するため，Ng クローン散布図 (Ng-clone scatter plot) と呼ぶことにする．

図 6 の Ng クローン散布図では，10 トークン以上の Ng クローンが全て描かれている (閾値 1=10 トークン)．図中に膨大に存在する黒い点は，非常に多くの短い Ng クローンが存在することを意味しており，これらには，単一の変数宣言が一致しただけといった短いコードクローンも含まれる．また，少し大きな Ng クローンのみを調べるため，閾値 1 を 30 トークンにすると Ng クローン散布図は図 7 のようになる．この図には，30 トークン以上の長めの Ng クローンしか現れていないので，容易に分析が行える．しかし，コピーとペーストとその修正によって生じたコードクローンなどは多く抜け落ちている可能性がある．

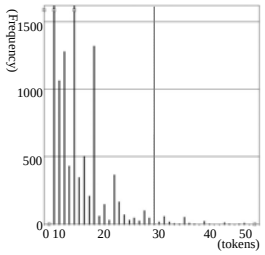

図 9　図 6，7 における Ng クローン出現頻度
Fig. 9　Number of Ng-clones in Figures 6 and 7

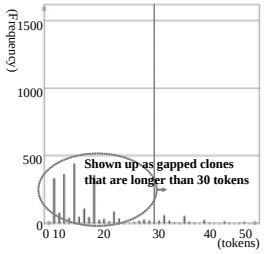

図 10　図 8 における Ng クローン出現頻度
Fig. 10　Number of Ng-clones in Figure 8

それに対し，Gap-and-clone 散布図は，個々の Gapped クローンを検出することに比べて計算コストが小さいまま，きめの細かい分析が容易に行える．図 8 は，閾値 1 を 10 トークン，閾値 2 を 10 トークン，閾値 3 を 30 トークンとした Gap-and-clone 散布図である．つまり，図 8 では，図 7 に現れたすべての Ng クローン，および，10 トークン以上の Ng クローンと 10 トークン未満のギャップで構成された 30 トークン以上の Gapped クローンが現れている．

Ng クローン散布図においての分析のきめの細かさと容易さとの間でのトレードオフと，Gap-and-clone 散布図によるそれらのトレードアップを定量的に説明する．図 9，図 10 の棒グラフには，Ng クローン散布図上の Ng クローンの出現頻度と Gap-and-clone 散布図上の Ng クローンの出現頻度が表されている．

図 9 の棒グラフは，図 6 の Ng クローン散布図に含まれる Ng クローンは殆どが 10 から 30 トークン未満

であり，図 7 の Ng クローン散布図には，少ししか Ng クローンが現れていないことを示している．逆に，図 10 の棒グラフでは，約 2000 個近い 10 から 30 トークンの Ng クローンが 30 トークン以上の Gapped クローンを構成するのに寄与していることが分かる．

これらのことは，特に学生 S に限ったことでなく，他の学生についても確認することができた．

5. まとめと今後の課題

本研究では，ギャップの位置情報に基づいた Gapped クローン抽出手法の提案を行った．また，本手法をコードクローン分析環境 Gemini へ実装し，本手法の有効性の確認のため，大学におけるプログラミング演習で作成されたプログラムに適用した．適用実験においては，本手法がコードクローン分析のきめの細かさと容易さとの間でのトレードオフを軽減することを確認できた．今後の課題としては，大規模なソフトウェアや実際の保守作業への適用が挙げられる．

文　献

[1] B.S. Baker, "A Program for Identifying Duplicated Code", *Computing Science and Statistics*, 1992, 24:49-57.

[2] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proceedings the 2nd Working Conference on Reverse Engineering*, 1995, 86-95.

[3] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Measuring Clone Based Reengineering Opportunities", *Proceedings 6th IEEE International Symposium on Software Metrics*, 1999, 292-303.

[4] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings IEEE International Conference on Software Maintenance-1998*, 1998, 368-377.

[5] S. Ducasse, M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code", *Proceedings IEEE International Conference on Software Maintenance-1999*, 1999, 109-118.

[6] M. Fowler, *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.

[7] T. Imai, Y. Kataoka, T. Fukaya, "Evaluating Software Maintenance Cost Using Functional Redundancy Metrics", *Proceedings of 26th Annual International Computer Software and Applications Conference*, 2002, 299-306.

[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*, 2002, 28(7):654-670.

[9] R. Komondoor, and S. Horwitz, "Using slicing to identify duplication in source code", *Proceedings 8th International Symposium on Static Analysis*, 2001.

[10] J. Krinke, "Identifying Similar Code with Program Dependence Graphs", *Proceedings 8th Working Conference on Reverse Engineering*, 2001, 562-584.

[11] J. Mayland, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proceedings IEEE International Conference on Software Maintenance-1996*, 1996, 244-253.

[12] 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", 大阪大学ソフトウェア工学講座技術報告 *SEL-June-3-2003*, (`http://sel.ist.osaka-u.ac.jp/~lab-db/betuzuri/archive/423/423.pdf`), 2003.

（平成年月日受付，月日再受付）

植田　泰士

平 13 阪大・基礎工・情報卒．平 15 同大大学院博士前期課程修了．現在，宇宙開発事業団所属．在学中，コードクローン分析の研究に従事．

神谷　年洋

平 8 阪大・基礎工・情報中退．平 13 同大大学院博士課程了．現在，科学技術振興事業団 PRESTO 研究員．博士 (工学)．オブジェクト指向関連技術，ソフトウェア保守 (メトリクス，コードクローン)，認知科学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．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，IFPUG 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〜昭 61 ハワイ大マノア校・情報工学科・助教授．平 1 阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．工学博士．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．