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

信学技報
TECHNICAL REPORT OF IEICE.
SS2001-14(2001-07)

クローン検出ツールを用いたソースコード分析ツールの試作

植田 泰士† 神谷 年洋‡ 楠本 真二† 井上 克郎†

†大阪大学 大学院基礎工学研究科 情報数理系専攻
〒560-8531 大阪府豊中市待兼山町 1-3
Phone: 06-6850-6571 Fax: 06-6850-6574
‡科学技術振興事業団 若手個人研究推進事業
E-mail: y-ueda@ics.es.osaka-u.ac.jp

あらまし 近年，プログラムの保守作業は，開発されるシステムの大規模化に伴い複雑かつ困難な作業となってきている．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，あるいは，類似したコードの断片を意味する．あるクローンにフォールトが含まれていた場合には，それに関連する箇所を全て修正する必要がある．しかし，大規模なプログラムの場合，関連箇所を手作業で全て修正することは困難である．そこで本研究では，クローン検出ツール *CCFinder* からの出力結果を視覚化したソースコード分析システムの試作を行った．本システムは，コードクローンの散布図や，クローンクラス (同値類) に関するメトリクスグラフから，警戒すべきコードクローン集合を特定し，対応したソースコードを対で表示する．

キーワード ソフトウェア保守，コードクローン，ソフトウェアメトリクス

Source Code Analysis System Using Code Clone Detection Tool

Yasushi Ueda†, Toshihiro Kamiya‡,
Shinji Kusumoto† and Katsuro Inoue†

†Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
Phone: +81-6-6850-6571 Fax: +81-6-6850-6574
‡PRESTO, Japan Science and Technology Corporation
E-mail: y-ueda@ics.es.osaka-u.ac.jp

Abstract Maintaining the software system is getting more complex and difficult, as its scale is getting larger. It is generally said that code clone is one of the factors that make software maintenance difficult. A code clone is a code portion in source files that is identical or similar to another. If a code clone contains a fault and is copied and modified many times, it is necessary to correct all the fragments. However, for the large-scale software, it is very difficult to correct them completely. In this paper, we develp a source code analysis system which visualizes the code clone information from Code Clone Detection Tool, *CCFinder*. Using the system, we can specify a set of code clone which should be examined through the scatter plot about clone pair or the softwere metrics graph about clone class(a set of equivalents), and refer the corresponded fragments of source code by the pair.

key words Software Maintenance, Code Clone, Software Metrics

1 はじめに

ソフトウェア保守は，出荷後のソフトウェアに対して施す，欠陥の修正，性能などの属性の改善，または変更された環境に適合させるための変更と定義されている [10]．近年，ソフトウェアシステムの大規模化，複雑化に伴い，プログラムの保守・デバッグ作業に要するコストが増加してきている．例えば，Lam は，ソフトウェア全ライフサイクルのコストの 66%が保守に費やされていると報告している [13]．保守作業の効率を高めることは，ソフトウェア工学における重要な課題の一つとなっている．

保守作業を難しいものとしている要因も幾つか指摘されている．その中の一つに，**Copy-and-Paste Programming(切貼プログラミング)** がある．切貼プログラミングとは，既存のプログラムに機能を追加する時に，そのプログラム中のコードフラグメントをコピーし修正することで，機能追加を行う手法である．この手法は，手軽に機能追加が行えるという利点があるが，コピーされたコード片にフォールトが含まれていた場合，フォールトがプログラム中に分散してしまうという問題点がある．また，重複したロジックを含むプログラムは修正が困難であるとの指摘もある [3]．ソフトウェア保守においては，このようなコードフラグメントに対する修正を行う際には，類似コード片を全て抽出し，漏れなく修正を施すことが必要となってくる．

我々の研究グループでは，一つあるいは複数のプログラムから，類似しているコードフラグメント (コードクローン) を検出するツール**CCFinder**[8] を開発してきている．CCFinder は，大規模プログラム開発の保守支援や教育環境におけるプログラム評価支援を目的として開発されている．しかし，コードクローンの検出結果はコードフラグメント対の位置情報の集合である為，特に大規模プログラム対する検出結果を直観的に理解することは困難である．従って，その利用にはコードクローン情報の視覚化，および即座にソースコードへフィードバックしていく手段が求められていた．

本研究では，CCFinder を利用したソースコード分析システムの試作を行い，実際のプログラム開発に適用することで，その有効性を評価することを目的とする．本システムは，コードクローンの散布図やコードクローンに関する種々のメトリクス値グラフから，警戒すべきコードクローン集合を特定する．更に，そのクローンに該当するコードフラグメント集合のソースコードを表示できるため，コードクローンに基いて，より保守性の高い再構築をすることが可能となる．

以降，2. では，コードクローン検出ツール CCFinder，およびクローンクラスに関するソフトウェアメトリクスについて説明する．3. では，本システムの設計，実装に関して説明する．さらに，4. では，大学におけるプログラミング演習に本システムを適用し，その結果についての分析と考察を行う．最後に 5. では，まとめと今後の課題について述べる．

2 準備

2.1 コードクローン検出ツール CCFinder

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α は β のクローンであるという (その逆もクローンであるという)．また，(α, β) を**クローンペア**と呼ぶ．α，β それぞれを真に包含するどのようなトークン列も等価でないとき α，β を極大クローンという．

CCFinder は，プログラムテキスト中から極大クローンを検出し，それをクローンペアの位置情報として出力する．また，クローンの同値類を**クローンクラス**と呼び [8]，プログラムテキスト中でのクローンを特に**コードクローン**と呼ぶ．

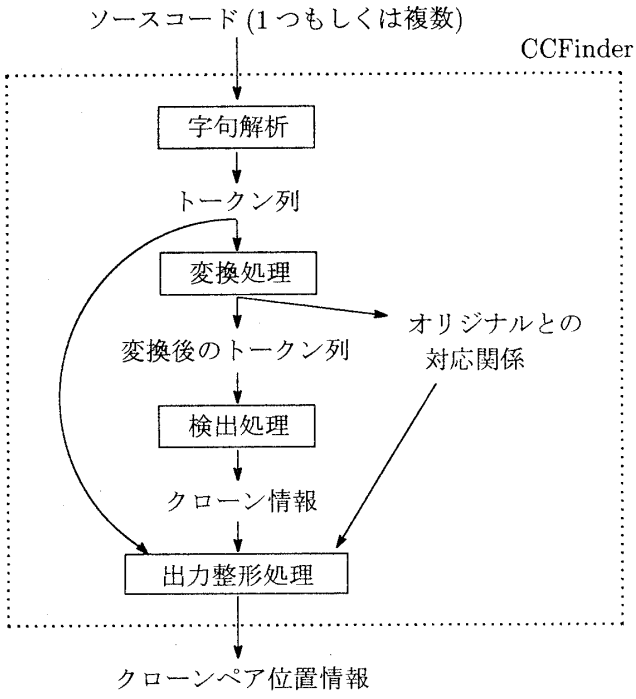


図 1: CCFinder の処理手順

CCFinder の処理概要　図 1 は CCFinder の処理概要を示す．まず最初に，ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．次に，そのトークン列を実用的に意味のあるコードクローンのみを検出するための変換を施す．例えば，パラメータ置き換え (名前が異なっても等価にする) などである．そして，そのトークン列を比較して，一致した部分トークン列をコードクローンとする．トークン列の比較には suffix-tree という木構造を用いたアルゴリズム [5] を採用し，計算複雑さはトークン列の長さを n として $O(n)$ [¶1] である．最後に出力整形処理を行い，検出されたクローンペアについて，元のソースコード上での行番号 (およびカラム番号) が出力される．

[¶1] 構築された suffix-tree からコード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 (極大コードクローンだけではなく全ての) コードクローンの数として $O(n+k)$.

```
#version: ccfinder 3.1
#langspec: JAVA
#option: -b 30,1
#option: -k +
#option: -r abcdfikmnprsv
#option: -c wfg
#begin{file description}
0.0 52  C:\Gemini.java
0.1 94  C:\GeneralManager.java
0.2 237 C:\MDI.java
1.0 7   C:\CCFEventListener.java
1.1 116 C:\CCFinderManager.java
1.2 695 C:\CCFinderOptionFrame.java
              :
              :
#end{file description}
#begin{syntax error}
1.1 56,13
#end{syntax error}
#begin{clone}
0.1 53,9   63,13   1.10 542,9  553,13 35
0.1 53,9   63,13   1.10 624,9  633,13 35
0.2 124,9  152,31  0.2  154,9  216,51 42
0.2 124,9  152,31  1.10 194,9  225,30 42
0.2 126,9  152,31  1.10 185,9  204,34 37
0.2 153,14 211,9   1.10 207,9  242,5  31
0.2 153,14 216,51  1.10 193,9  225,30 44
0.2 172,9  216,51  1.10 185,9  204,34 37
              :
              :
#end{clone}
```

図 2: CCFinder 出力結果例

CCFinder の主な特徴は次の通りである.

- 数百万行規模のシステムにも実用時間 (約 290 万行で 40 分 [8]) で解析可能
- 言語依存部分を取り替えることで，さまざまなプログラミング言語に対応
- 主記憶容量に対し，1 つの suffix-tree で構成するにはソースコードが大きすぎる場合，部分集合に分割して suffix-tree を構成する
- 実用的に意味のあるコードクローンのみを検出するための工夫が施されている
  - 最小一致トークン数が指定できる
  - 定数 (数値定数，文字列定数) を区別しない
  - 共有ルーチンとして再構築するのが困難なクローンは検出しない
    - テーブル初期化部分を取り除く
    - モジュールの区切りを認識する
  - クラススコープや名前空間による複雑な名前の正規化を行う
  - 繰り返しコードが存在した場合には冗長な結果を出力しないようにする
- 解析対象ファイル集合を 1 階層のグループに分類可能

CCFinder の出力　CCFinder から得られるコードクローン検出結果は図 2 のようなテキストベースの位置情報である．例えば，#begin{clone},#end{clone} 内の 1 行目は，0 グループ 1 番目のファイルの 53 行目 (の 9 文字目) から 63 行目 (の 13 文字目) までと，1 グループ 10 番目のファイルの 542 行目 (の 9 文字目) から 553 行目 (の 13 文字目) までがコードクローンとして検出されている．分析者がこの情報とソースコードだけから実際にコードクローンを調査し，分析や再構築を行うことはかなり困難であることが指摘されていた．

2.2 クローンクラスに関するソフトウェアメトリクス

文献 [8] においてクローンクラスに関する以下のソフトウェアメトリクスが定義されている．本ソースコード分析システムにおいてはそれら 4 つのメトリクスを用いる．

RAD(C)(Radius of clone-class): クローンクラス C 内のコードフラグメントを一つでも含むファイルの集合を F とすると，$RAD(C)$ は F がファイルシステムの中でディレクトリ構造的にどれだけ分散しているかを表す．ディレクトリ構造を表す木構造を考え，F 内の全てのファイルに関して，共通の親ディレクトリの中で最もルートノードから遠いノードまでの距離を求め，それらの距離の最大値を $RAD(C)$ として定義する．

$RAD(C)$ の値が大きいならば，クローンクラス C 内のコードクローンは，システム内において，広範に広がっているため，バグの作り込まれたコードクローンに修正を施すのはより困難であると考えられる．

LEN(C)(Length): クローンクラス C 内の 1 コードフラグメントのサイズを表す．文献 [8] では，クローンクラス C 内の最大 LOC (Lines Of Code:ソースコードの行数) として定義されているが，最大トークン数として再定義する．

POP(C)(Population of clone-class): クローンクラス C に含まれるコードフラグメントの数を表す．$POP(C)$ が高いということは，より多くの箇所にコードクローンが分散していることになる．

DFL(C)(Deflation by clone-class): クローンクラス C を再構築した場合に，どれだけコードサイズが減少するかの予測値を表す．$LEN(C)$ 同様，文献 [8] では，減少分の LOC として定義されているが，トークン数として再定義する．ここでの再構築とは，クローンクラス C 内のコードフラグメント集合を新たな 1 サブルーチンに整理し，そのサブルーチンを各箇所で呼び出すという最も単純な場合を考えている．まず再構築対象となる総トークン数は $LEN(C) \times POP(C)$ で求めることができる．そして，再構築対象が削除された後，新たに追加されるコードはサブルーチン実装コードと，サブルーチンコール用コードであるが，ルーチンサイズは $LEN(C)$ トークン，コール用コードは合計 $5 \times POP(C)$ トークンとなる．呼び出し回数の $POP(C)$ に 5 をかけて

いるのは，1回の呼び出しにかかるトークン数はサブルーチン名，(，引数，)，;の5トークンとみなしているためである (C/C++やJavaの場合).

従って $DFL(C)$ は以下のように定義することができる.

$$\begin{aligned} DFL(C) &= (\text{再構築対象コードサイズ}) \\ &\quad -(\text{再構築後コードサイズ}) \\ &= (LEN(C) \times POP(C)) \\ &\quad -(LEN(C) + 5 \times POP(C)) \\ &= (LEN(C) - 5) \times (POP(C) - 1) - 5 \end{aligned}$$

3 ソースコード分析システム

3.1 コンセプト

これまで既に関連研究として，コードクローンの位置情報を分析や視覚化するツールDUPLOC [11][12] が開発されている．DUPLOCはコードクローンの視覚化手法としてクローン散布図 (3.2.2で述べる) を用いているが，この視覚化手法はコードクローンがどこにあるのかが一目で分かり，ソースコードとのインタラクティブな操作への応用ができる非常に有効な手法であると考えられる．しかしながら，特に大規模プログラムにおいては，膨大なクローン情報の中からどの部分に注目していいのか判断するのは困難な作業である．

従って，警戒すべき，もしくは特徴あるコードクローンを探す際の負担をできる限り減らす工夫が必要となる．また散布図の見た目とソースコードだけで分析を行うのには限界があるため，統計的，多角的な分析を行う必要もある．

そこで本研究では，次のようなコンセプトをもって本システムの試作を行った．

- コンセプト1: **クローン分析を見据えたクローン位置情報の効果的な表示**
- コンセプト2: **クローン位置情報から得られる付加情報により分析に多角性をもたせる**

3.2 システム構成

システム構成の概略を図3に示す．本システムはCCFinderを内部的に実行させ，そこから得られた結果を用いて分析を行うものである．

図上では，処理は上から下へ流れる．ユーザは，まずGUI上から，CCFinderの実行に関する諸設定を行い，CCFinderを起動させ解析を行う．解析が終了すると，クローンペア管理部，クローンクラス管理部のGUIに結果が表示されるので，ユーザはその上で分析を行うことになる．重要なのは，クローンペア管理部，クローンクラス管理部が持つGUIを通してユーザがクローンペア (クラス) を選択できる点であり，その部分に効果的な表示，及び，選択方法を工夫していく必要がある．そして，クローンペア管理部，クローンクラス管理部のGUIによって指定されたクローンがソースコード管理部のGUIによって表示され，利用者は実際にソースコードを参照できる．

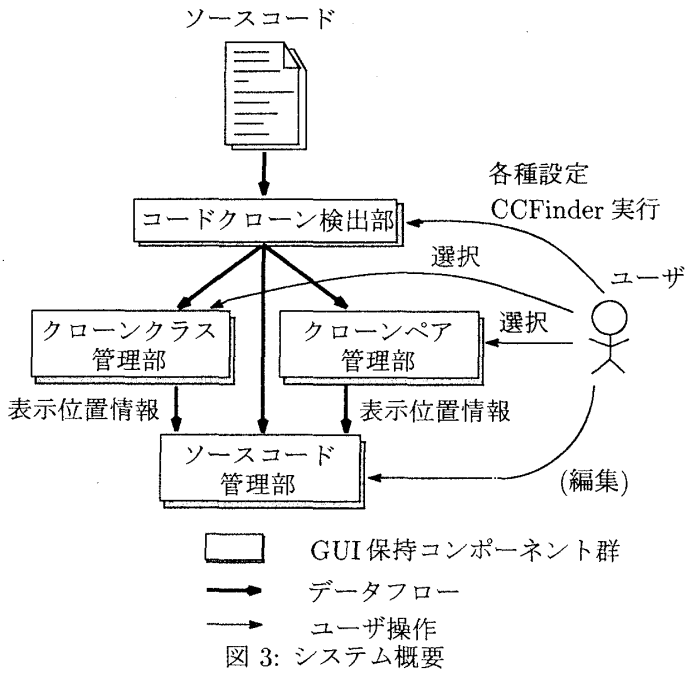


図 3: システム概要

以降では図3上でGUIを保持しているコンポーネント群毎にその構成と機能について述べる．

3.2.1 コードクローン検出部

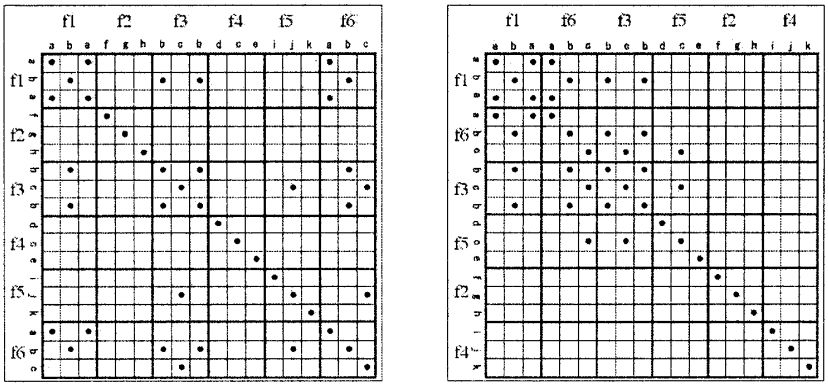
ここではCCFinderを実行させるものとして必要なコンポーネント群を管理する．解析対象言語 (C/C++，Java，COBOL，平文テキスト) や最小一致トークン数等の指定ができたり，解析対象ファイルリストを1階層のグループに分類しながら作成できる．

3.2.2 クローンペア管理部

ここでは，コードクローン検出部によって得られたクローンペアの位置情報を取得し，クローン散布図ビュー，クローンペア一覧ビューを管理する．クローン散布図ビューおよびクローンペア一覧ビュー上で，利用者は特定のクローンペアを指定することによって，そのクローンペアを「選択状態」にすることができる．選択状態になったクローンペアは，他のビューおよびソースコード管理部が保持しているソースコードビューにおいても選択された状態として表示される (以下「ビューの同期機能」).

クローン散布図

ここでいうクローン散布図というのは図4(a) のような図のことである．座標の原点は左上角にとってあり，水平軸，垂直軸には同じように解析対象ファイルが1行目から順に並ぶ．図では，簡単の為，各ファイルを3トークンずつにしている．つまり各軸に並ぶ「a，b，a,...」はトークンを表す．黒丸 (点) は，その座標の水平軸，垂直軸のトークン



(a) プロット例　　(b) 図 4(a) のソート後

図 4: クローン散布図プロット例とそのソート例

内容が一致していることを表す．当然左上から右下への対角線上では自己比較が行われるため，必ず一本の線が引かれ，その対角線に対して線対称な分布状態となる (実際のシステムでの描画単位は行単位であるので，点ではなく直線として描画する)．　コンセプト 1 を踏まえ，この散布図に持たせた機能として，**ソースコードブラウズ機能，ズーム機能，コードクローンを持たないファイルの表示/非表示機能，ファイル区切りグループ区切りの表示/非表示機能，ソート機能**などが挙げられる．

ソート機能は，コンセプト 1 を実現する本システムの最も重要な機能である．本来，座標軸上のファイルの並びをファイルを読み込んだ順番等にすると，場合によって図 4(a) のようにコードクローンが目立った偏りがなく広範囲に広がってしまう場合がある．それではどの部分に着目し分析を行えばいいのか判断が難しく，分析に相当の労力が必要となる．ソート機能は，その労力を軽減することを目的としている．

そのアイデアは，できるだけコードクローンが散布図中で分散しないようにすることである．具体的には，座標軸上でのファイルの並びに関して，できる限り類似しているものを近くに配置する．例えば，図 4(a) においては，仮にファイル $f1$ の位置が決定すれば，その次に配置するのは，まだ配置場所の決まっていないファイル集合の中から，$f1$ に対して最も類似率の高いファイル $f6$ を配置する．$f6$ に対しても同じことを繰り返していけば隣り合うもの同士が類似しているため，類似率の高いファイル集合は自然と集中していく (図 4(b) 参照)．

また類似率は，ファイル $f1$ に対するファイル $f2$ のコードクローンによるコードカバー率 (以下ファイル間クローンカバレッジ) として，次式で定義する．

$$FCC = \frac{\sum_{cf \in CF} cf \text{ の行数 (但し重複部分は 1 度だけ)}}{f1 \text{ 全体の行数}}$$

ここで，FCC は $f1$ に対する $f2$ のファイル間クローンカバレッジを，CF は $f1$ 内に含まれるコードクローンの中で $f2$ とクローンペアとなっている $f1$ 側のコードフラグメント集合を表す．

次に，先頭に配置するファイルを決めなければならない．一概にどれが良いかという判断基準が難しいが，本研究ではコードクローンの分布は，出来る限り左上角に集合させたいという方針をとる．従って，先頭のファイルはできるだけ，次のファイルが高い類似率を持ってこれるよう誘導する必要がある．そのため，ファイルの組み合わせでなく 1 ファイルに対し 1 つ定まる値を定義しなければならない．そこで，ファイル f に対する解析対象全ファイルのコードクローンによるコードカバー率 (以下，単にクローンカバレッジと呼び，定義はファイル間クローンカバレッジの $f1$, $f2$ を f，解析対象全ファイルに読み替える) を用いる．これが最も高いものを先頭のファイルとすれば，2 番目に来るファイルと高い類似率を持つ可能性が高い．

また，先頭ファイルから順に配置を決定していっても，途中で，最も高いファイル間クローンカバレッジでも値が 0 という場合がでてくるが，ここでもクローンカバレッジを用いて次のファイルを決定する．

3.2.3　クローンクラス管理部

ここではコンセプト 2 に基づき，分析に多角性を持たせる為，コードクローン検出部によって得えられたクローンペアの位置情報から，コード片をクローンクラスに分類し，各クローンクラスについてクローンメトリクスを算出する．

クローンクラス管理部のインターフェイスはメトリクスグラフビューとクローンクラス一覧ビューである．メトリクスグラフビュー，クローンクラス一覧ビュー，ソースコードビューについてもビューの同期機能が実装されている．

クローンメトリクスグラフ

用いるクローンメトリクスは 2.2 節で述べた，$RAD(C)$,

$LEN(C)$, $POP(C)$, $DFL(C)$ の4つ (定義より $DFL(C)$ は $LEN(C)$ と $POP(C)$ の積に比例する為, 実質3種類) である. しかし, 各メトリクスを各軸にとって2次元グラフを書くと, 分析時に複数のグラフを参照せねばならず手間がかかる. また今後の拡張性等 (メトリクスの追加) も考慮した上で, 本研究では, 多次元並行座標表現 [9] を用いる. 多次元並行座標表現は, 大量のデータから価値のある情報を導き出すデータマイニングを行う際に有効である. 本システムの多次元並行座標表現では, 1クローンクラスにつき1本の折れ線が描画される.

このグラフの持つ機能として**ソースコードブラウズ機能**, **フィルタリング機能**(メトリクス値の上限と下限を指定することでクローンクラスを選択する機能), **折れ線描画色のグラデーション機能**(指定したメトリクスの値の大きさを基準), **軸の上下反転機能**(密集した折れ線をみやすくするため) がある.

フィルタリングは, グラフ全体にフィルタを貼り, そのフィルタをドラッグ操作により大きさを調整することで各軸の値範囲の選択を行えるようにした.

3.2.4 ソースコード管理部

ここでは, クローンペア管理部, クローンクラス管理部から選択状態の情報を受け取り, そのソースコードビューにコードを表示 (コードクローン部分は強調表示) する. クローンペアに関しては上下対応させて表示する.

3.3 実装

本システムは Java 言語 [4][7][15] を用いて実装を行った. 表示例 (ソースコードの表示時) を図5に示す.

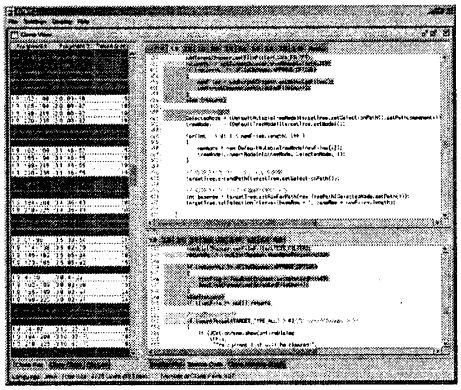


図 5: 表示例

4 プログラミング演習への適用

4.1 概要

大阪大学基礎工学部情報科学科のあるプログラミング演習において作成されたプログラムに対し, 本システムを適用した. またソースコードは C 言語で書かれたものである.

課題内容は「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アセンブラ言語 CASL で記述されたプログラムに翻訳 (変換) するコンパイラを作成する」である. 作成にあたり, 学生には指導書が渡され, 指導書にはコンパイラの仕様が掲載されている. また別の講義ではコンパイラの作成法について学習し, その講義で用いられる教科書には, サンプルプログラムが掲載されている. 課題1として構文チェッカ (Parser) を, 課題2として意味チェッカ (Checker) を, 課題3としてコンパイラ (SPC) をそれぞれ順に開発する. なお, 課題2,3においては, 課題1,2のプログラムを拡張, 再利用することで実現することが求められた. 収集したソースコードのサンプル数は計 76 人分 (約 38 万行) である. これは Parser,Checker,SPC の全てが揃っていたもののみを対象としている.

4.2 結果とその分析

以下に示す結果は, 最小一致トークン数を 30 トークンに設定してコードクローン検出を行ったものである.

結果 1:ソートの有効性確認

3.2.2 節で述べたソート機能が, どの程度有効であるか確認を行う.

ここでは,Parser プログラムのソースコードに対するコードクローン検出結果を例に用いて説明する. また, 収集したソースコードはある年度とその前年度の2年分のものであるため, 図 6(a) に示した図は. グループ区切りを開発年度にし, 各グループ内では, 各個人の名前の辞書順にファイル集合を並べている (1個人が単一ファイルに対応しているとは限らない).

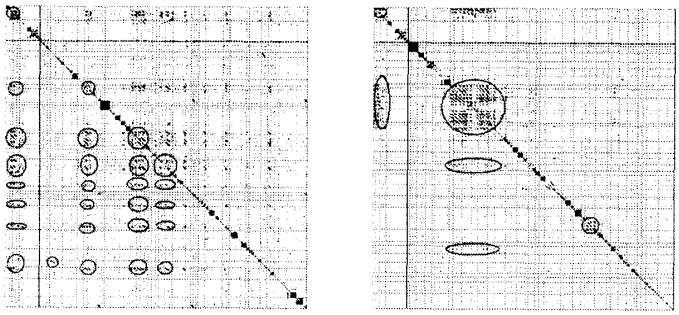
図のようにやはりクローンペアの分布状態は散布図全体に広がり, 点在箇所それぞれを何回にも分けて分析しなければならないことが分かる. そこで, グループ (開発年度) の区切りを保ったままグループ内でソートを行った. すると 30 箇所近くあったクローンペア密集領域が, わずか 5,6 箇所に整理された (図 6(b) 参照). 図 6(a) では分かり辛かったが, 類似ファイル集合が年度越しに派生していることが容易に確認できる.

また,Checker,SPC についても同様にすると, これらの類似ファイル集合に含まれる人数は,Parser,Checker,SPC へと拡張されるにつれ, 次第に減少していくことが確認できた (12 人→ 6 人→ 3 人).

結果 2 : 再利用性

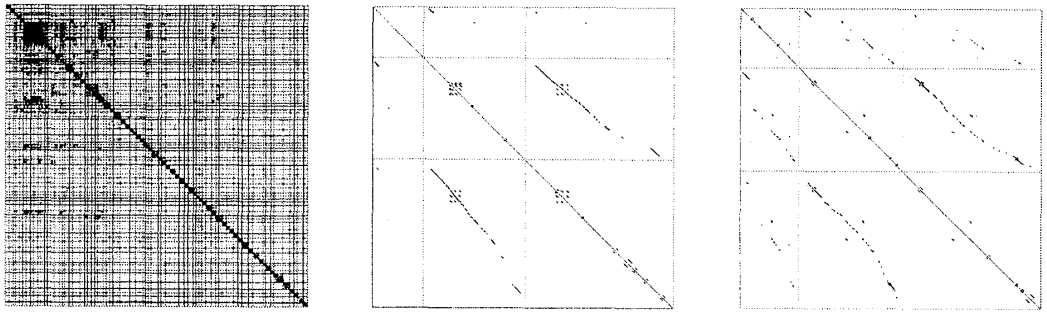
1グループを1個人とし, グループ内では全課題のソースコードを含めてコードクローンを検出した場合, 全員の再利用性を容易に知ることができる. 図 7(a) は, グループ内において課題順に Parser,Checker,SPC が配置され, グループソート完了後の散布図である (ファイル区切りは非表示).

対角線上に並ぶ正方形の各領域は各個人内での比較なの



(a) ソート前　　(b) ソート後

図 6: 年度区切り散布図 (Parser)



(a) 全体図 (各個人間類似率でソート済み)

(b) 課題 1,2 間での再利用性の低い例 (左 (上) から順に Parser,Checker,SPC)

(c) 課題 1,2 間での再利用性が高い例 (左 (上) から順に Parser,Checker,SPC)

図 7: 個人区切り散布図 (Parser+Checker+SPC)

で, その正方形内の課題間領域に再利用性が現れてくることになる. 拡大図が, 図 7(b) と図 7(c) であるが, それぞれ, 課題 1,2 間での再利用性の低い例と高い例となっている.

全員の各個人内散布図を分析したところ, ほぼ 8 割が図 7(b) のように課題 1,2 間での再利用性がかなり低い (平均約 7.5%) ことが確認された. しかしながら, 課題 2,3 間ではほぼ全員にある程度の再利用性の向上みられた (平均約 30.3%).

結果 3 : メトリクスグラフからの特徴のあるコードフラグメントの抽出

以下は, 再利用性を分析した際と同じ解析対象 (グループ分け) を設定している.

- **DFL 値分析**

  DFL 値が非常に高いクローンクラスを調べたところ, Parser で作り込まれた膨大なコードクローンを Checker 拡張時に的確に 1 モジュール内に再構築した例と, それを残したまま拡張している例を発見できた.

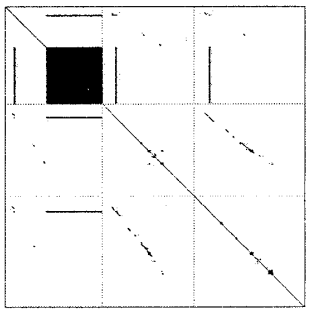
  図 8(a) を見れば,Parser 後半部分に大きな黒い長方形が存在し, 類似コードが何回も繰り返されている (それぞれの間には異なるコードフラグメントが挟まれている) のが分かる. しかしながら,Parser と Checker の比較領域を見れば, その長方形が一本の線に変わっているのが分かる. 実際にソースコードを見ても, 一つのルーチンとして置き換えられており, 機能拡張であるのにも関わらず 100 行のコードサイズ減少が確認できた.

  逆に, 図 8(a) では,Parser で作り込まれたコードクローンがほとんど再構築されず, コードクローン間の距離が拡大し, 後々のプログラム内に分散していく様子がみられる. この場合, 保守性が下がっているかと予測し, Parser から Checker,Checker から SPC への拡張時の平均増加コード量と比較したが, 大した差は現れてはいなかった. 正確に拡張に要した時間, コンパイル回数等を平均値と比較すれば何らかの違いが確認できるかもしれない.

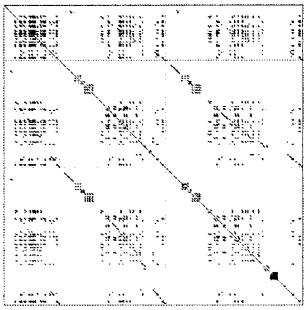
- **RAD,POP 値分析**

  この場合,RAD 値が高いということは, 複数人物で使用されたコードフラグメントであり, POP 値が高いということは, 多数回使用されているコードフラグメントであるので, この値が共に高いクローンクラスは, 多くの人にかなり使われたコードフラグメントである.

  実際のソースコードを参照したところ,Pascal 風言語における「変数宣言の並び」もしくは「仮パラメータの並び」の構文解析ルーチンであった. 指導書による



(a) 再構築してから拡張



(b) コードクローンを残したまま拡張

図 8: 拡張特性 (左 (上) から順に Parser,Checker,SPC)

と「変数宣言の並び」と「仮パラメータの並び」の定義は以下のようであった.

```
変数宣言の並び     = 変数名の並び ":" 型 ";"
                     { 変数名の並び ":" 型 ";" }
仮パラメータの並び = 変数名の並び ":" 型 ";"
                     { 変数名の並び ":" 型 ";" }
```

定義は完全に同一のものであり，実装を行うとほぼ同じコードになる．しかしながら，多くの学生はそれぞれを別ルーチンとして処理している．ルーチンに引数を持たせ多少の処理分岐を行えば単一ルーチンとしてまとめることも可能である．しかし，宣言変数と仮パラメータの変数では処理的意味が全く異なるので，以後の保守性を下げる可能性もある．これは一概にコードクローンを全て再構築すべきであるとはいえない例となっている．

5 まとめと今後の課題

本論文では，コードクローン検出ツール CCFinder の検出結果を利用し，コードクローンに基づいたソースコード分析を行うシステムの試作を行った．2 つの主な機能，一つはクローンペアの位置情報をクローン散布図で表現し，対応ソースコードを参照する機能，もう一つは，クローンクラスからソフトウェアメトリクスを算出，グラフ化し，対応ソースコードを参照する機能を持つ．

また有効性の確認のため，本システムをプログラミング演習に適用した．クローン散布図においては，クローン分析における散布図ソート機能の有効性の確認等，クローンメトリクスグラフにおいては，特徴あるコードフラグメント集合を容易に抽出できることを示した．

今後の課題としては，**クローンペア・クローンクラスの含む構文による分類，メトリクス値を散布図に反映させる，繰り返し開発の行われたソフトウェアのソースコードを分析する**などが挙げられる．

参考文献

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

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

[3] M. Fowler, "Refactoring: improving the design of existing code", Addison Wesley, 1999.

[4] D. M. Geary, "Graphic Java2, Mastering the JFC 3rd Edition vol.2", ASCII, 2000.

[5] D. Gusfield, "Algorithms on Strings, Trees, And Sequences", Cambridge University Press, 1997.

[6] J. Helfman, "Dotplot Patterns: a Literal Look at Pattern Languages, TAPOS, 2(1):31-1,1995.

[7] S. Holzner, "Java Swing プログラミング Black Book", 株式会社インプレスコミュニケーションズ, 2000.

[8] 神谷 年洋, 楠本 真二, 井上 克郎, "コードクローン検出における新手法の提案及び評価実験", 電子情報通信学会技術研究報告 SS2000-42〜52 Vol.100 No.570 pp.41-48, 2001.

[9] 加藤 博己, "データベースのビジュアルな検索と分析 (OLAP)", IPSJ Magazine Vol.41 No.4 pp. 363 - 368, 2000.

[10] C. McClure, "The Three Rs of Software Automation Reengineering, Repository, Reusability, Prentice-hall",1992, (藤本 厚, 藤堂 清, 小野 誠, 堀田 耕治, 芦沢 真佐子 [訳],"ソフトウェア開発と保守の戦略", 共立出版株式会社, 1993).

[11] M. Rieger, S. Ducasse, "Visual Detection of Duplicated Code", 1998.

[12] M. Rieger, "DUPLOC Tutorial", 1999.

[13] Standard for Software Maintenance, 1219, IEEE Computer Society, 1993.

[14] 辻野 嘉宏, "コンパイラ", 昭晃堂, 1996.

[15] Java(TM) 2 Platform Standard Edition, http://java.sun.com/j2se/1.3/.