Title	開発保守支援を目指したコードクローン分析環境
Author(s)	植田，泰士；神谷，年洋；楠本，真二 他
Citation	電子情報通信学会論文誌D-I. 2003, J86-D-I(12), p. 863-871
Version Type	VoR
URL	https://hdl.handle.net/11094/26456
rights	copyright©2003 IEICE
Note	

論　文

開発保守支援を目指したコードクローン分析環境

植田　泰士[†]　神谷　年洋[††]　楠本　真二[†]　井上　克郎[†]

Code Clone Analysis Environment for Supporting Software Development and Maintenance

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

あらまし　ソフトウェアの保守作業を難しくしている要因の一つとしてコードクローンがある．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．あるコード片にバグが含まれていた場合には，そのコード片のコードクローンすべてについて修正の是非を検討する必要がある．しかし，大規模なソフトウェアの場合，それらすべての箇所を手作業で発見し，修正の是非を検討することは非常に困難である．本研究では，コードクローン検出ツールCCFinderの検出結果を利用したコードクローン分析環境Geminiの構築を行う．本システムは，開発保守におけるコードクローンの利用を支援するため，コードクローンの位置情報の視覚化，コードクローンに関するメトリックス値の算出，及び対応したソースコードを参照する機能を備える．本システムを実際のプログラムに適用することで，特徴的なコードクローンなどを抽出できることを確認した．

キーワード　ソフトウェア保守，コードクローン，ソフトウェアメトリックス

1. まえがき

ソフトウェア保守とは,「納入後，ソフトウェア・プロダクトに対して加えられる，フォールトの修正，性能またはその他の性質改善，変更された環境に対するプロダクトの適応のための改訂」であると定義されている[9]．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，多くのソフトウェア会社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．

コードクローンは保守作業を困難にする要因の一つであると指摘されている[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．

コードクローンが保守作業を阻害する理由は，例えば，修正されるコード片がコードクローンであれば，すべての類似コード片を何らかの手段で同様に修正する必要性が生じるためである．対策としては，

- コードクローン情報を文書化しておくことで一貫した修正を可能にする，
- コードクローンを必要に応じて検出する

の二つがある[10]．

しかし，前者の方法はコードクローン情報を常に最新の状態に保つ手間がかかり，実現困難である．後者の方法として，いくつかのコードクローン検出手法やツールが提案されている[1]〜[6],[12]〜[14]．

我々が開発してきているコードクローン検出ツールCCFinder [11]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することが可能である．しかし，(1) CCFinderの出力はコードクローンの位置情報（ファイル名，行，カラム）であり，直感的な理解が困難である．また，(2) 大規模なソフトウェアから検出された膨大なコードクローンを分類・整理する機能が提供されないなど，実際の保守作業に利用するには問題があった．

[†] 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, Toyonaka-shi, 560–8531 Japan
[††] 独立行政法人科学技術振興機構さきがけ
PRESTO, Japan Science and Technology Agency, Japan

本論文では，大規模ソフトウェアにおけるコードクローンの分析を行うための統合環境 Gemini を構築する．Gemini は内部的に CCFinder を利用してコードクローンの検出を行い，様々なユーザインタフェースを通して，ソフトウェア技術者に対して有益なコードクローン情報を視覚的に提示する機能を備えている．また，本システムを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，その有用性を評価する．

以降，**2.** では，コードクローンに関する諸定義，コードクローン検出ツール CCFinder，コードクローンに関するメトリックス等について説明する．**3.** では，Gemini の設計と実装について，**4.** では，適用事例とその結果について述べる．最後に **5.** で，まとめと今後の課題を述べる．

2. 準　　備

本章では，コードクローンやその検出方法について説明する．

2.1 コードクローンの定義と関連用語

あるトークン列中に存在する二つの部分トークン列 α, β が等価であるとき，α と β は互いにクローンであるという．また，ペア (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いかなるトークン列も等価ではないとき，α，β を極大クローンという．また，クローンの同値類をクローンクラスと呼ぶ．ソースコード中でのクローンを特にコードクローンという．

2.2 コードクローン検出処理手順

Gemini がコードクローン検出器として利用する 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．処理は以下の四つのステップからなる．

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，個々のファイルから得られたトークン列を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）：実用上意味をもたないコードクローンを取り除くこと，及び，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，変数名が付け換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，指定された長さ以上一致しているような部分クローン列をすべて検出する．

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

2.3 コードクローンに関するメトリックス

Gemini ではコードクローンを識別するために，以下の六つのメトリックスを利用する．ただし，四つのメトリックス LEN，POP，DFL，RAD については，文献 [11] で提案されているものである(注1)．

LEN(C) (Length) [11]:
クローンクラス C 内に含まれるコード片の最大トークン長を表す．

POP(C) (Population) [11]:
クローンクラス C 内のコード片の数を表す．

DFL(C) (Deflation) [11]:
クローンクラス C を再構築した場合に減少するコードの量の予測値を表す．ここでいう再構築とは，保守者が手作業等によって，クローンクラス C 内のコード片集合から抜き出した共通のロジックを実装するサブルーチンを作り，各コード片をそのサブルーチン呼出しに置き換えることを指す．$DFL(C)$ は，再構築前のコードの大きさ（すなわち，クローンクラス C 内のコード片の大きさの和）から，構築後のコードの大きさ（すなわち，サブルーチン呼出しの大きさの大きさの和+共通ロジックを実装するサブルーチンの大きさ）を引いたものとして定義される．実際には，クローンクラスの中には再構築が不可能なものも存在するので，DFL は再構築を行うべきクローンクラスの優先順位を付けるために用いることを想定している．

RAD(C) (Radius of clone class) [11]:
ソースファイルが階層型ファイルシステムに格納されている場合に，クローンクラスのファイルシステム内での「広がり」を表す．クローンクラス C 内のコード片を含むソースファイルの集合を F として，F 内各要素のファイルパスすべてに共通した最も下位の（ルートから遠い）ディレクトリから，F 内各要素のファイルまでの距離の最大値と定義する（図 1 参照）．

例えば，クローンクラス a のコード片は 2 ファイルに分散しているが，それら 2 ファイルのファイルパ

(注1)：ただし，メトリックス計測の実装は本論文で開発したツール Gemini において初めてなされたものである．

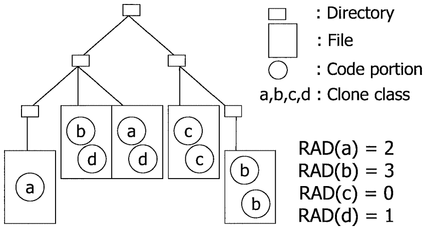

図 1　RAD 値の例
Fig. 1　Sample values of RAD.

スに共通した最も下位のディレクトリは，ルートディレクトリの 1 階層下の左側のディレクトリである．本ディレクトリから各コード片までの距離はそれぞれ 2 及び 1 になるため，$RAD(a)$ は 2 となる．

直観的には，RAD 値が低ければ，同一ファイル内や同一ディレクトリに閉じているため，そのクローンクラスをサブルーチンに置き換えることは比較的容易であると考えられる（修正を行う場合も同様）．逆に，RAD 値が高ければ，それらのコード片が様々なモジュールに分散している，あるいは，（特に複数の開発者がかかわる場合は）異なる開発者のモジュールに含まれるなど，修正の難易度は高くなると考えられる．

RSA(*f*) (Ratio of similarity to all other files): RSA は，ファイル f と，ファイル f 以外のすべてのファイルそれぞれとの間に存在するクローンペアによって，ファイル f のコードがどれだけカバーされているかの割合を表し，次の式で定義される．

$$RSA(f) = \frac{1}{Len(f)} \sum_{cf \in CF(f)} Len(cf)$$

($Len(f)$: ファイル f のトークン長)

($Len(cf)$: コード片 cf のトークン長)

ここで，$CF(f)$ は，ファイル f 以外のファイルとクローンの関係をもつファイル f 内のコード片の集合を表し，cf は 1 コード片を表す．ただし，本総和においては，重複しているコード部分は，1 度しか考慮しないものとする．

Gemini において RSA は，コードクローン情報を効果的に視覚化するために利用される（**3. 2. 2** 参照）．

RST(f_1, f_2) (Ratio of similarity between two files): $RST(f_1, f_2)$ は，ファイル f_1 がファイル f_2 とのクローンペアによってどれだけカバーされているかの割合を表し，次の式で定義される．

$$RST(f_1, f_2) = \frac{1}{Len(f_1)} \sum_{cf \in CF(f_1, f_2)} Len(cf)$$

ここで，$CF(f_1, f_2)$ は，ファイル f_2 とクローンの関係をもつファイル f_1 内のコード片の集合を表す．RSA 同様，本総和においては，重複しているコード部分は，1 度しか考慮しないものとする．

RST は，2 ファイル間の類似度として用いることができる．もし f_1，f_2 が，あるソースファイルの二つのバージョンであれば，RST の計測値はバージョン間の類似度を意味する．計測値が小さければ，バージョンアップの際に多くの変更が加えられた可能性がある．

Gemini において RST は，RSA 同様，コードクローン情報を効果的に視覚化するために利用される（**3. 2. 2** 参照）．

3.　コードクローン分析環境 Gemini

3. 1　Gemini の機能概略

Gemini はコードクローンの分析を行うための統合環境であり，GUI により，(1) クローン散布図及び (2) **2. 3** で定義したコードクローンに関するメトリックスのグラフを表示・操作する機能，(3) 個々のコードクローンのソースコードを表示する機能をもつ．なお，クローン散布図は Duploc [6] でも実装されている機能である．

クローン散布図はクローンペアを視覚的に位置表示する図であり，プロットされたそれぞれの点は，その座標に対応するソースコード位置が一致していることを表す（詳細は **3. 2. 2** で述べる）．

クローン散布図を使うことで，クローンペア全体の分布状態を俯瞰的に表示し，ソースコードを素早く参照することができるなど，対話的操作に有効な視覚化手法である．しかし，本研究が想定するような大規模なソフトウェアのコードクローン分析においては散布図は膨大な数の点を含むため，単にクローン散布図を表示するだけではなく，着目すべき部分を強調する，フィルタリングを行うなど，利用者を補助するための方法が必要である．本課題に対して提案する我々の手法は，**3. 2. 2** で述べる．

また，コードクローンメトリックスのグラフは，クローンクラスやファイルを対象としており，クローンペアを表示するクローン散布図とは異なった対話的な分析機能を提供する．詳細は **3. 2. 3** で述べる．

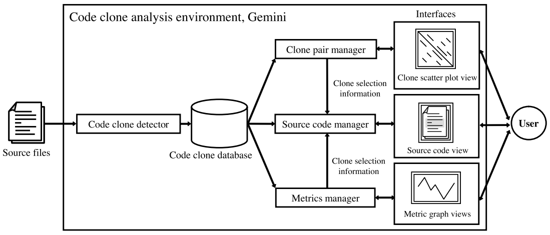

図 2　システムの構成
Fig. 2　Architecture.

3. 2　システム構成

本システムは，内部的に CCFinder を実行し，CCFinder から得られた解析結果をもとに分析環境を提供する．システムの構成を図 2 に示す．

Gemini は主に五つのコンポーネント：(1) コードクローン検出部（Code clone detector），(2) クローンペア管理部（Clone pair manager），(3) メトリックス管理部（Metrics manager），(4) ソースコード管理部（Source code manager），(5) ユーザインタフェースで構成されている．

まずはじめに，コードクローン検出部にソースファイルが入力され，すべてのコードクローンを検出する．次に，クローンペア管理部と，メトリックス管理部がその解析結果であるコードクローン情報を各インタフェースを通して視覚化する．それらのインタフェース上では，ユーザは任意のクローンペア（あるいは，クローンクラス）を選択することができ，その選択によって，実際のソースコードをソースコード管理部とそのインタフェースを通して参照することができる．

次節から図 2 における各コンポーネントについて順に述べる．

3. 2. 1　コードクローン検出部

解析対象のファイルや，コードクローン検出のパラメータを管理する．CCFinder を利用して検出したコードクローンの位置情報も管理する．

3. 2. 2　クローンペア管理部

クローンペア位置情報をもとにして，利用者の要求に応じて，クローン散布図を表示する．クローン散布図上での GUI による操作として，拡大・縮小，及び，任意のクローンペア集合を「選択状態」にすること，がある（選択状態は，後述する他のサブシステムと連携した操作で使われる）．図 3 に選択状態のクローンペア集合を含んだクローン散布図の例を示す．方形で囲まれたクローンペアが選択状態になっている．

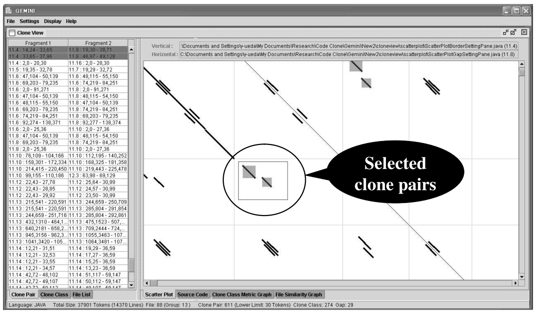

図 3　クローン散布図ビューの表示例
Fig. 3　GUI snapshot of clone scatter plot view.

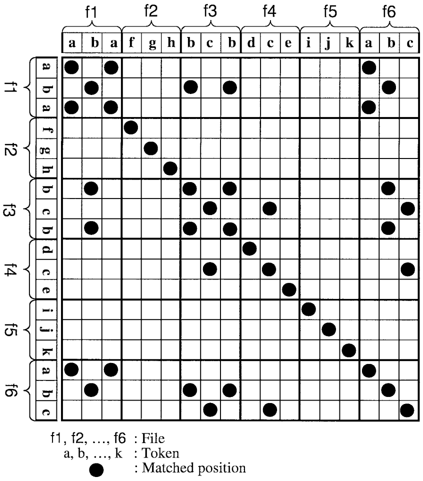

図 4　クローン散布図モデル
Fig. 4　Simple model of clone scatter plot.

[クローン散布図]

Gemini が表示するクローン散布図の簡単なモデルを図 4 に示す．散布図の原点は左上隅にあり，水平軸，垂直軸は，それぞれソースコードの並びに対応している．両軸上で，原点から順に，それぞれのソースファイルに含まれるトークンが並んでいる．座標平面内に点がプロットされている部分は，その両軸の対応するトークンが一致することを意味する．したがって，散布図の主対角線は，両軸の同じ位置のトークンを比較することになり，すべての点がプロットされることになる．また，点の分布は主対角線に対して線対称となる．一定の長さ（CCFinder で設定される最小一致トークン数）以上の対角線分が，検出されたクローンペアである．図 4 では，`f1` から`f6` までのファイルが，

それぞれ三つのトークンを含んでいる．ファイルの並びはファイル名の辞書順となっている．検出されるクローンペアは`f1`と`f6`に含まれる“ab”というトークン列と，`f3`と`f6`に含まれる“bc”というトークン列である．

[クローン散布図のソート機能]

一般に，ソースファイルにはコードクローンを含まないものがあり，クローンペアのクローン散布図内で疎に分布する可能性がある．大規模なソフトウェアのクローン散布図を操作する場合，疎な分布であることは，すなわち，広大なクローン散布図を全体にわたって参照しながら分析を行うことを意味し，分析の手間が増えることになる．

これを解決するため，クローン散布図のファイル順序を調節することで，分布を密にする方法（クローン散布図のソート機能）を提案する．ソート機能により，クローン散布図内の点の分布が密になり，大規模ソフトウェアのコードクローン分析労力を軽減することができる．このソート機能が，Duplocのもつ散布図に対する優位性をもたらすものである．

ソート機能により，類似したファイル群がまとまって配置され，また，散布図の原点周り（若しくは対角線近辺）にクローンが集められる．

本ソート手順は次のようになる．

ステップ1: 対象ファイルのうち，RSA計測値が最も高いファイルfを先頭のファイルとする．

ステップ2: 順序が決定されていない対象ファイルのうち，fとの間でRSTが最も高いファイルf'を次のファイルとする．f'をfとして，すべての対象ファイルの順位が決定するまで，ステップ2を繰り返す．

図4のファイルでは，`f1`が最も高いRSAをもつので，`f1`が先頭のファイルになり，以下，`f1`とのRSTが最も高い`f6`，`f6`とのRSTが最も高い`f3`，という順番になる．図5に，図4に対してソートを行った結果のクローン散布図を示す．上述のとおり，原点近くに点が集まっていることが確認できる．

3.2.3　メトリックス管理部

クローンペア位置情報から，6種のメトリックスを算出する．利用者の要求に応じて，メトリックスグラフビューによってメトリックス計測値を表示する．クローンクラスに関するグラフ（RAD, LEN, POP, DFLを表示）とファイルに関するグラフ（RSA，RSTを表示）がある．利用者は，各グラフ上で計測メトリックス値の分布状態を確認することができる．

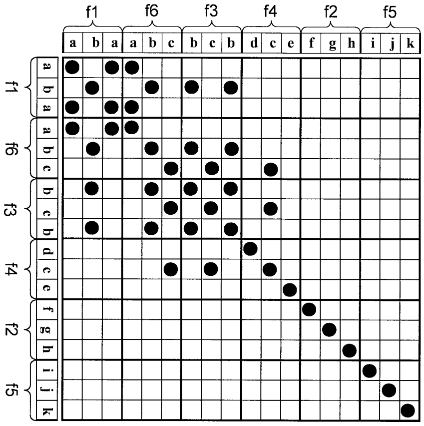

図5　図4のクローン散布図のソート後
Fig. 5　Sorted model of clone scatter plot in Fig. 4.

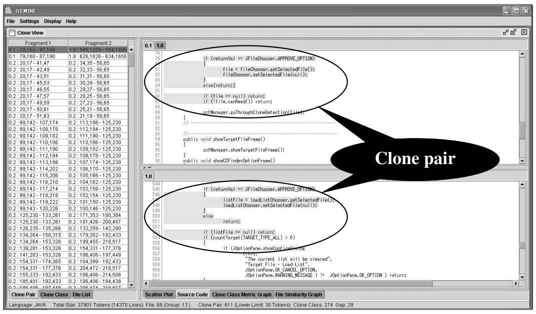

図6　ソースコードビューの表示例
Fig. 6　GUI snapshot of source code view.

また，GUIによる操作を行い，グラフの一部を「選択状態」にすることができる．

3.2.4　ソースコード管理部

指定されたソースファイルの内容をソースコードビューを通じて表示する（図6参照）．

3.3　サブシステム間の連携

クローンペア管理部，ソースコード管理部，メトリックス管理部の間で「選択状態」を用いた連携が可能になっている．この連携により，利用者は例えば(1)クローン散布図でクローンペアを選択した後，そのソースコードを表示する，(2)メトリクスグラフで一部のクローンクラスを選択した後，そのソースコードを表示する，(3)メトリックスグラフで長い（LENが大きい）クローンクラスだけを選択した後，クローン散布図ではそれらがどこにあるのか調べる，といった対話的操作を行うことができる．

3.4 実装

本システムは，Java で実装されており（約 13,000 行），JDK1.3VM が実行可能な環境で動作する．表示画面の例を図 3，図 6 に示す．

4. 適用事例

4.1 概要

大阪大学のあるプログラミング演習で作成されたプログラムに対し，Gemini の適用を行った．

本演習において，各学生は，Pascal 言語（のサブセット）で書かれたプログラムを CASL 言語（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．作成にあたり，学生にはコンパイラの仕様が掲載された指導書が与えられた．また，学生は，他の講義で，簡単なコンパイラのサンプルプログラムが記載されたテキストを用いてコンパイラの設計方法についての説明を受けている．本演習は，次の三つのステップ（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拡張して作成することが課題として要求されている．つまり，*Checker* は *Parser* を，*SPC* は *Checker* を拡張することで作成される．

本適用に際し，69 人の学生（$S_1..S_{69}$）のプログラム（*Parser*, *Checker*, *SPC*）を収集した（計 360KLOC）．

4.2 分析

本適用において，次の項目について分析を行った．

（1） メトリックス値（*DFL* 値）を用いたクローン抽出

（2） 全プログラムの間での類似性

まず,（1）では，各学生の各プログラム内での分析を行う．具体的には，メトリックス *DFL* の値とクローン散布図を組み合わせることにより，いくつかのモジュールに再構築できるような特徴的なコードクローンを抽出できるかどうかを確認する．このようなコードクローンは，実際の保守においてもリファクタリング対象となり得るので，Gemini の有効性を示す一例となる．

（2）では，一種のコード盗用分析を行っている．プログラミング演習では，作成期限に間に合わせるなどのために，他の学生のプログラムをコピー及び修正することで，プログラムを作成するといったことが起こり得る．そこで，メトリックス *RSA* の値等を調べることで，学生が作ったプログラム間の類似性を確認する．もちろん，（2）の分析は，Gemini の保守作業における有用性を直接的に評価することにはならないが，教育環境におけるコードクローン分析の一つの応用事例になると考えている．

4.2.1 メトリックス値（*DFL* 値）を用いたクローン

DFL 値を算出するにあたって，各サブルーチン呼び出しは 5 トークン（“サブルーチン名”，“(”，“引き数”，“)”，“;”）とし，共通ロジックを実装するサブルーチンの大きさはコードクローンの各コード片と同じとする．したがって，

$$\begin{aligned} DFL(C) &= LEN(C) \times POP(C) \\ &\quad -(POP(C) \times 5 + LEN(C)). \end{aligned}$$

表 1 に各プログラムにおける *DFL* 値の最大値を示すが，メトリックスグラフビューに表示された *DFL* 値の分布状態から，S_1 の *Parser* ($DFL = 3538$) 及び S_2 の *SPC* (DFL=3439) は，*DFL* 値に関して特に突出した値をもっていることが確認された．

表 1 から，S_1 の $DFL(Parser)$，$DFL(Checker)$，$DFL(SPC)$ の値は，それぞれ 3538，163，189 となっていることがわかるが，$DFL(Parser)$ は非常に高い値であるのに対し，$DFL(Checker)$，$DFL(SPC)$ の値は，ほぼ平均値に近くなっている．これは，*Parser* から *Checker* へ拡張が行われる際に，コードクローンを除去する再構築が行われたことを意味する．

一方，S_1 のプログラムに関するクローン散布図を

表 1 各プログラムにおける *DFL* 値の最大値
Table 1 The maximum values of *DFL* in each program.

	Parser	Checker	SPC
S_1	**3538**	163	189
S_2	100	211	**3439**
S_6	79	131	131
S_7	145	199	199
S_3	75	97	199
S_4	75	75	391
S_5	75	119	233
≈	· · ·		≈
S_{69}	223	211	258
max.	3538	603	3439
min.	0	47	51
ave.	196	183	311

図 7 に示す．本散布図では，両座標軸上に *Parser*，*Checker*，*SPC* が課題の順に並んでいる．*Parser* は，*DFL* 値が突出したクローンクラスを含んでいた（図中 A の部分）．*Parser* と *Checker* の比較（図中 B の部分）においては，図中 A の部分のような密集したクローンは存在せず，一本の線分が現れている（図中 C の部分）．これは S_1 が *Checker* を作成する際に，*Parser* に多く存在したコードクローンを一つのサブルーチンにマージしたことを意味しており，実際，ソースコードを参照することにより，その再構築が確認できた．

次に，図 8 に S_2 のクローン散布図を示す．A と同様の，コードクローンが密集した部分が存在した（図中 D の部分）．ソースコードを参照することで，これらのコードクローンが，S_1 の場合と同様に再構築可能かを確認した．これらのコード片で異なっている部分は，ある 2 箇所の定数名のみであり，それらをパラメータ化することで，容易に一つのサブルーチンにまとめることが可能であった．

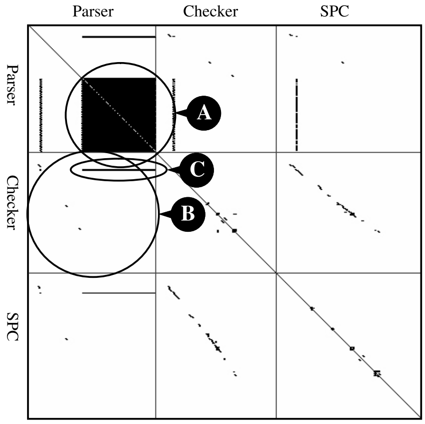

図 7　S_1 のクローン散布図
Fig. 7　Clone scatter plot for S_1' programs.

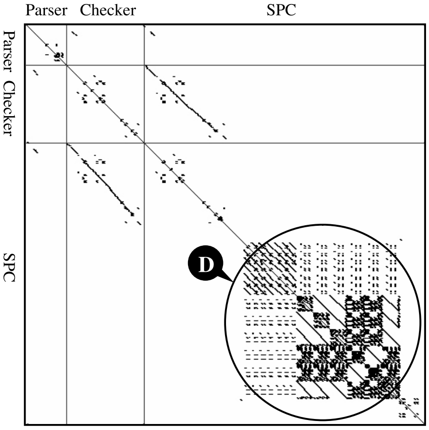

図 8　S_2 のクローン散布図
Fig. 8　Clone scatter plot for S_2' programs.

実際の保守作業においても，このように *DFL* 値の高いコードクローンを分析することは,「共通ロジックをサブルーチンにまとめる」等のリファクタリングを行うことができる箇所の発見に有用であると考えられる．ただし，パラメータの追加が必要な場合については，モジュールの結合性を増大による保守容易性低下の可能性も考慮する必要がある．

4.2.2　全プログラムの間での類似性

RSA 値の計測結果を表 2 に示す．表 2 における *Parser*，*Checker*，*SPC* の *RSA* 値の平均値は，それぞれ 0.089，0.032，0.019 となっており，課題が進むにつれ，各個人間での類似性は低くなっていくことが確認された．これは，後の課題（課題 2 や課題 3）の方がより学生のオリジナリティを必要としているため，自然な結果と考えられる．

しかしながら，幾人かの学生は，課題 3 においても高い *RSA* 値をもっていた．そこで，クローン散布図を用い，学生間のコードクローンがどのように分布しているのか調べた．図 9 のクローン散布図は，全学生の SPC のソースコードを比較した結果であり，格子は各個人間の区切りを表している．本散布図上では，コードクローンが散布図全体に広く分散している（Duploc における散布図に相当）．そこで Gemini に実装されたソート機能を用いて，ファイル順序の並べ換えを行った．ソート機能によってコードクローンの

表 2　*RSA* 値
Table 2　Values of *RSA*.

	Parser	Checker	SPC	ave.
S_1	0.028	0.009	0.011	0.016
S_2	0.000	0.000	0.000	0.000
S_3	0.029	0.244	**0.159**	0.144
S_4	0.162	0.271	**0.148**	0.194
S_5	0.326	0.211	**0.137**	0.224
S_6	0.297	0.244	**0.160**	0.234
S_7	0.348	0.151	**0.142**	0.214
		···		
S_{69}	0.032	0.004	0.003	0.013
ave.	**0.089**	**0.032**	**0.019**	0.046
max.	0.407	0.271	0.160	0.234
min.	0.000	0.000	0.000	0.000

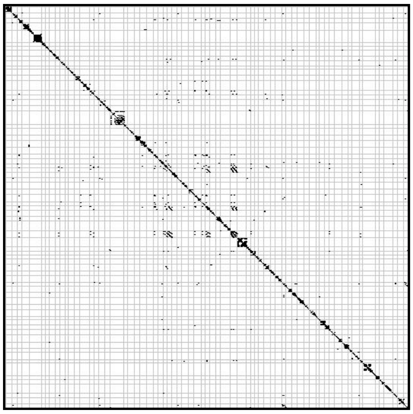

図 9　全学生の SPC を比較したクローン散布図（ソート前）
Fig. 9　Unsorted scatter plot of all students' SPC.

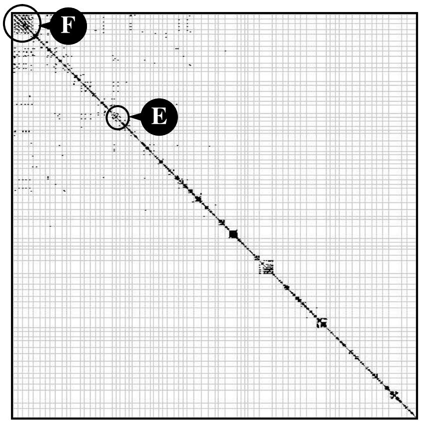

図 10　全学生の SPC を比較したクローン散布図（ソート済み）
Fig. 10　Sorted scatter plot of all students' SPC.

分布は，図 10 のように対角線周辺に密集した．

図 10 のクローン散布図では，数箇所でコードクローンが密集していることがわかる．E を付した部分は 2 人の学生 S_3，S_4 の SPC が含まれ，F を付した部分は 3 人の学生 S_5，S_6，S_7 の SPC が含まれていた．

まず，S_3，S_4，S_5，S_6，S_7 の RSA 値（表 2 参照）を調べてみると $RSA(SPC)$ と $RSA(Checker)$ の値については，全学生の中で上位 5 位を占めていた．更に S_5，S_6，S_7 の $RSA(Parser)$ 値は，上位 10 位内に入っていた．実際に，ソースコードビューを用いて対応したコードクローンを調べてみたところ，まず S_3，S_4 における類似コードのほとんどは，テキストに記載されたサンプルコンパイラのコードであることが確認された．一方，S_5，S_6，S_7（すなわち F の部分）における類似コードは，変数名やコメントまで類似が見られた．つまり，3 人のソースコードの間で互いに何らかの参照が行われた可能性が高いと考えられる．

4. 2 でも述べたように，学生のプログラム間の類似性を評価することは，Gemini の保守作業における有用性を評価することにはならない．しかし，実際の保守作業では，設計品質を適切に維持管理できていなければ，類似サブシステム（ファイル）が，コピーとその部分的な修正による再利用によって多数存在し，かつその類似サブシステム群を追跡しきれなくなることが頻発する．そのような場合，本散布図におけるソート機能を用い，本適用のように類似ファイル群を探し出すことは設計品質の向上を目的とした設計見直し等に有用であると考えられる．

5. むすび

本論文では，開発保守支援を目指したコードクローン分析環境 Gemini の構築を行った．Gemini が提供するクローン散布図やメトリックスグラフ等のインタフェースを用いることで，保守者はシステム全体に散在するコードクローンの位置情報，その特徴等を直観的に認識することができ，保守作業における類似コード修正や設計見直し等が必要な箇所の発見に役立てることができる．特にクローン散布図におけるソート機能は大規模ソフトウェアのコードクローン分析労力を軽減することに役立つと期待される．

また，Gemini を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（約 36 万行）へ適用した．その結果，メトリックス値と散布図を用いたクローン抽出の結果，リファクタリングを行うことでプログラムの理解容易性を向上させることができるようなコードクローンを発見した．更に，クローン散布図の提供するソート機能を用いることで，プログラム間の類似性をより直観的に判断することができた．

今後の課題としては，まず，大規模なソフトウェアや実際の保守現場へ Gemini を適用することが考えられる．例えば，百万行以上の大規模プログラムに対しても Gemini が問題なく動作するかどうかを確認しなければならない（Gemini のスケーラビリティの評価）．また，実際に発見されるコードクローンが保守を行う上で有用であるかどうかを調べることも必要である（コードクローンの特性評価）．更に，クローン

散布図に導入したソート機能が実利用において役立つかどうか（ソートアルゴリズムの妥当性評価）を確認することも必要である．

文　献

[1] B.S. Baker, “A program for identifying duplicated code,” Computing Science and Statistics, vol.24, pp.49–57, 1992.

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

[3] 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, Boca Raton, USA, Nov. 1999.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, “Partial redesign of Java software systems based on clone analysis,” Proc. 6th IEEE International Working Conference on Reverse Engineering, pp.326–336, Atlanta, USA, Oct. 1999.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proc. IEEE International Conference on Software Maintenance-1998, pp.368–377, Bethesda, USA, Nov. 1998.

[6] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” Proc. IEEE International Conference on Software Maintenance-1999, pp.109–118, Oxford, UK, Sept. 1999.

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

[8] D. Gusfield, Algorithms on strings, trees, and sequences, Cambridge University Press, 1997.

[9] IEEE Std 1219: Standard for software maintenance, 1997.

[10] 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, 2001.

[11] 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, 2002.

[12] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” Proc. 8th International Symposium on Static Analysis, pp.40–56, Paris, France, July 2001.

[13] J. Krinke, “Identifying similar code with program dependence graphs,” Proc. 8th Working Conference on Reverse Engineering, pp.562–584, Stuttgart, Germany, Oct. 2001.

[14] J. Mayland, C. Leblanc, and E. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” Proc. IEEE International Conference on Software Maintenance-1996, pp.244–253, Monterey, USA, Nov. 1996.

（平成 15 年 3 月 31 日受付，7 月 21 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了．現在，宇宙航空研究開発機構所属．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，独立行政法人科学技術振興機構 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（メトリックス，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 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．工博．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．