The University of Osaka
Institutional Knowledge Archive

Title	コードクローン解析に基づくリファクタリングの試み
Author(s)	肥後, 芳樹; 植田, 泰士; 神谷, 年洋 他
Citation	情報処理学会論文誌. 2004, 45(5), p. 1357-1366
Version Type	VoR
URL	https://hdl.handle.net/11094/50160
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

コードクローン解析に基づくリファクタリングの試み

肥後芳樹[†] 植田泰士[†] 神谷年洋[††]
楠本真二[†] 井上克郎[†]

近年，ソフトウェアの大規模化・複雑化にともない，保守作業に要するコストは増大している．ソフトウェアの保守を困難にしている要因の1つとしてコードクローンがあげられる．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．たとえば，あるコード片にバグが含まれていた場合，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考慮する必要がある．コードクローンに関する処理を支援するために，我々はコードクローン分析環境 Gemini を開発してきた．Gemini をさまざまなプロジェクトに適用する中で，いくつかの問題点に直面した．クローン検出の目的の1つとしてリファクタリングがあげられるが，Gemini によってユーザに示されるクローンはリファクタリングを適用できる単位となっていなかった．本論文ではこの問題を解決し，Gemini を機能拡張した．また，オープンソースのソフトウェアに対して適用実験を行い，本手法の有用性を確認する．

On Software Maintenance Process Improvement Based on Code Clone Analysis

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

Maintaining software systems is getting more complex and difficult task. Code clone is one of the factors that make software maintenance more difficult. A code clone is a code portion in source files that is identical or similar to another. If some faults are found in a code clone, it is necessary to correct the faults in its all code clones. We have developed a maintenance support environment, Gemini, which provides the user with the useful functions to analyze the code clones and modify them. However, through case studies, several problems were reported. That is, the clones extracted by Gemini were not necessarily appropriate to merge into one module. In this paper, we intend to extend the functionality of Gemini to cope with the problems. Finally, we apply the extended Gemini to several software and evaluate the applicability of the new functions.

1. はじめに

近年，ソフトウェアの大規模化・複雑化にともない，ソフトウェアの保守作業がますます困難なものになってきている．ソフトウェア保守では，フィールドバグの修正，環境変化に対する機能追加・変更，将来トラブルにつながりそうな箇所に対する対応などの作業が実施される[20]．しかし，保守対象のソフトウェアがうまく設計されていない，度重なる変更により構造が分かりにくくなってしまう，変更履歴のドキュメントが存在しない，などの問題により，ソフトウェア保守コストは増大してきている．実際に，多くのソフトウェア会社が既存システムの保守に非常に多くの人的，時間的コストをかけていると報告されている[24]．

コードクローンはソフトウェア保守を困難にしている1つの要因といわれている[8]．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．コードクローンが生成される原因はさまざまな理由が考えられるが，その最も大きな原因の1つとしてコピーアンドペーストによる修正，拡張作業があげられる．コード片にバグが含まれていた場合，そのコード片のコードクローンとなっている部分すべてに対して修正の是非を考慮する必要がある．このような作業は，特に大規模ソフトウェアでは非常に手間がかかる．したがって，コードクローン検出の効率

† 大阪大学大学院情報科学研究科
Graduate School of Information and Science Technology, Osaka University
†† 科学技術振興機構さきがけ
PRESTO, Japan Science and Technology Agency

化はソフトウェア保守工程の改善において有効である．これまでにコードクローンを自動的に発見するためのさまざまな手法が提案されている[3)~7),12)~16),18),19)]．

その手法の 1 つとして，我々はコードクローン検出ツール CCFinder[13)] と分析環境 Gemini[21)] を開発してきている．ユーザは Gemini を用いることによりコードクローンの解析，ソースコードの修正を容易に行うことができる[23)]．Gemini は主に，クローン散布図とメトリクスグラフをユーザインタフェースとして提供する．クローン散布図はソースコード中のコードクローンの分布状態を俯瞰的に表示する．またメトリクスグラフは各々のコードクローンについての定量的な情報を提供し，その値を用いることによって保守を阻害するコードクローンの選択をすることが可能である．選択されたコードクローンのソースコードは容易に閲覧できる．ユーザはこれらの機能を用いることによってソフトウェアの保守作業を改善することができると期待できる．

我々は Gemini を数十のソフトウェア会社に配布し，さまざまなプロジェクトに用いることによって評価した．その結果，ソフトウェア会社からのフィードバックよりいくつかの問題点が発見された．最も多く指摘された問題は，Gemini をリファクタリング[8)] に利用する際に発生する問題であった．一般的に，コードクローンを除去することを目的として，コードクローンになっている部分を 1 つのメソッドやクラスにまとめるリファクタリングが適用される．しかし Gemini によって検出されたコードクローンは，必ずしも 1 つのモジュールとしてまとめるのに適していない．

本論文では，この問題を解決するために Gemini のコードクローン検出部に対して行った拡張について論ずる．そして最後に，提案した機能の有用性を確認するために行った適用実験の結果について述べる．

2. コードクローン解析

2.1 コードクローンの定義[9)]

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 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ないとき，α，β を極大クローンと呼ぶ．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 コードクローン分析環境 Gemini

文献 21) において我々はコードクローン分析環境 Gemini を開発した．**図 1** はシステムのアーキテクチャを示している．Gemini は内部の CCFinder にソースコードを渡し，CCFinder の解析結果をさまざまなユーザインタフェースを通してユーザに提供する機能を有する．

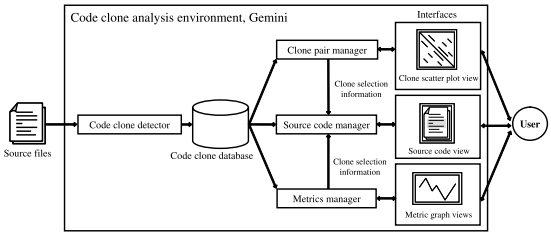

図 1 Gemini 全体図
Fig. 1 Overview of Gemini.

本章では，簡単に Gemini と CCFinder の特徴を説明する．

2.2.1 CCFinder

CCFinder はプログラムのソースコード中に存在するコードクローンを検出し，その位置をクローンペアのリストとして出力する．検出されるコードクローンの最小トークン数はユーザが前もって設定することができる．

CCFinder のコードクローン検出手順（ソースコードを読み込んで，クローンペア情報を出力する）は以下の 4 つの STEP からなる．

STEP1（字句解析）：ソースファイルを字句解析することにより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，単一のトークン列を生成する．

STEP2（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，些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する．たとえば，この変換により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．

STEP3（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部分をクローンペアとしてすべて検出する．

STEP4（出力整形処理）：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位置情報を出力する．

CCFinder の詳細については文献 13) で述べられている．

2.2.2 Gemini

Gemini は GUI ベースのコードクローン分析環境で

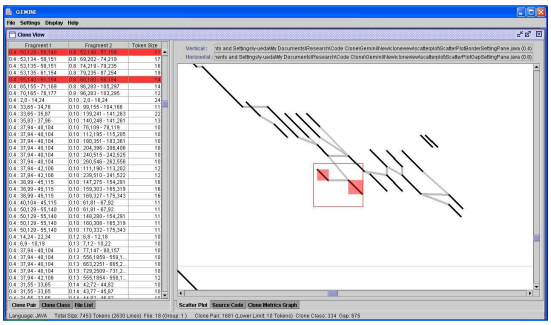

図 2　クローン散布図表示例
Fig. 2　Scatter plot.

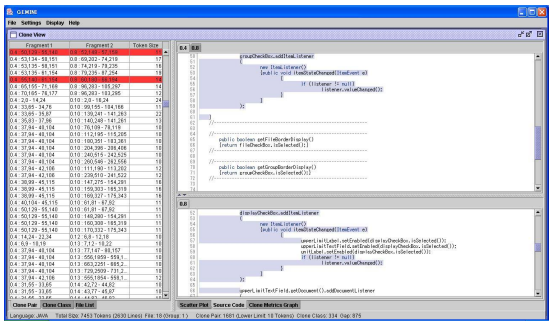

図 3　ソースコードビュー表示例
Fig. 3　Source code view.

あり，内部的にコードクローン検出部として CCFinder を用いている．Gemini は以下のユーザインタフェースを提供し，対話的な解析を可能としている．

- クローン散布図，
- メトリクスグラフ，
- ソースコードビュー．

クローン散布図はソースコードのどの部分にクローンペアが存在するのかを示す図である．一目でソースコード中のコードクローンの分布状況が分かるので，コードクローン解析の初期段階では非常に有効な解析手段となりうる．図上でユーザはマウスを用いて任意のクローンペアを選択することが可能である．例を**図 2** に示す．クローン散布図の詳細については後ほど論ずる．

またメトリクスグラフを用いることにより，ユーザはコードクローンを定量的な特性に基づいて選択できる．それぞれのクローンクラスについて複数のメトリクス値が示されていて，ユーザは長いコードクローンや，出現数の多いコードクローンを選択できる．

ソースコードビューはクローン散布図やメトリクスグラフと組み合わせて用いられる．ユーザはクローン散布図やメトリクスグラフで選択されたクローンのソースコードをソースコードビューを用いることにより閲覧できる．**図 3** では，図 2 において選択されたコードクローンのソースコードを表示している．

2.2.3　クローン散布図

図 4 はクローン散布図の例を示している．クローン散布図の縦軸と横軸にはソースコード中のトークンが出現順に配置される．ここではクローン散布図を説明するために以下の文字列を用いる．

コード片 X：“ABCDCDEFBCDG”，

コード片 Y：“ABCEFBCDEBCD”

ここでは，“A” や “B” などは文字や，トークン，行，文などのある一定の単位を表すとする．図 4 の格子内

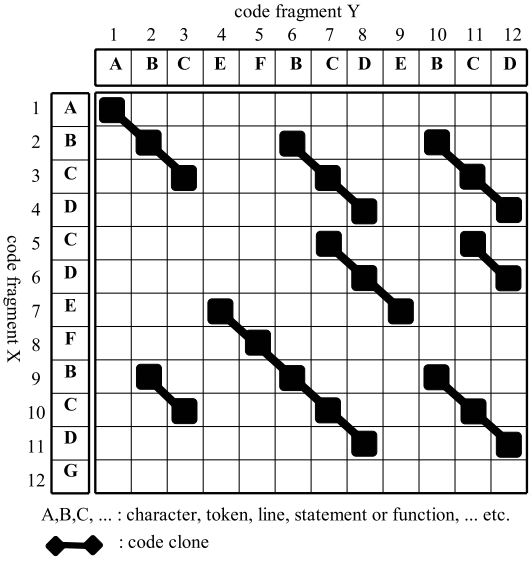

図 4　クローン散布図モデル
Fig. 4　Scatter plot of code clones.

の黒色の矩形はその縦軸の要素と横軸の要素が等しいことを意味している．このことからクローンペアはクローン散布図においてある一定以上の長さを持った線分として出現することになる．もし縦軸と横軸に配置される要素が同じソースファイルである場合は，主対角線上に黒色の矩形がプロットされる．またこのときはこの対角線に対してクローン散布図は線対称となる．

3.　提案手法

3.1　これまでの手法の問題点

我々は，Gemini（含む CCFinder）をさまざまな商用，及び非商用ソフトウェアに適用してきた．そしてフィードバックとしていくつかの問題点が指摘された．

検出したコードクローンをリファクタリング[8] に用いる場合の問題点として，単に最大長のクローンを検

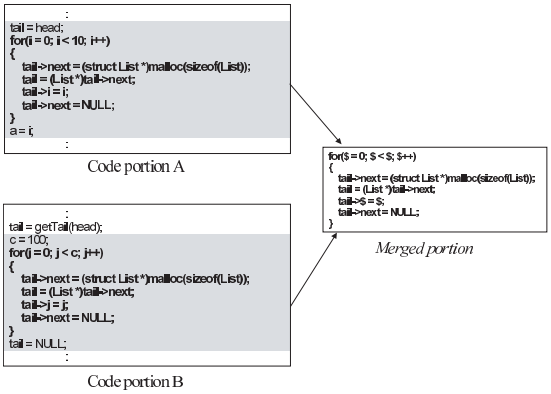

図 5 コードクローンのリファクタリング例
Fig. 5 Example of merging two code fragments.

出してもその部分を 1 つの関数，メソッドなどにまとめることは困難であることがあげられる．我々はこれまでに多くの実験を行ってきたが，その実験で検出された大部分のコードクローンは，プログラミング言語の意味にてらして構造的なまとまりを持たないものであった．**図 5** に例を示す．図 5 では，A と B の 2 つのコード片が示されている．A と B それぞれの灰色の部分は，その部分が A と B の間の最大長のコードクローンであることを示している．コード片 A ではいくつかのデータがリスト構造の先頭から順に連続して格納されている．一方コード片 B では，リスト構造の後方から順に連続してデータが格納されている．これら 2 つのコード片間にはともにリスト構造を扱っているという点において論理的に共通している．しかしながらこれらのコード片の for 文の前後には偶然クローンとして含まれてしまった部分が存在している．リファクタリングの視点からは，灰色の部分全体よりも for 文のみをコードクローンとして抽出する方が望ましい．

コードクローンをリファクタリングするための研究はいくつか行われている．文献 14) と文献 15) では，プログラム依存グラフを用いてコードクローンの凝集度を測定し，関数やメソッドの抽出に用いる方法が述べられている．しかしそれらの方法では，プログラム依存グラフの構築に非常にコストがかかることから大規模なソフトウェアへの適用が難しく，スケーラビリティの面から見て問題がある．一方 CCFinder では，検出過程において字句解析のみにとどめているため，その検出処理は非常に高速である．しかし，CCFinder によって検出されるコードクローンは単に最大一致トークン数によるものであり，コードクローンの凝集度は考慮されていない．つまり CCFinder を用いたコードクローン検出をリファクタリングに適用しようとした場合，ユーザは自ら CCFinder の検出結果から構造的なまとまりのある部分を抽出する必要がある．

この問題を解決するため，我々はまず最大長のコードクローンを検出し，次に，そのコードクローンに含まれる構造的なまとまりのある部分のみを抽出し，最後にユーザに指定された最小トークン以上のクローンを取り出すという 3 段階のアプローチを提案する．

3.2 アプローチ

CCFinder によって検出されたコードクローンから 1 つの関数やメソッドとしてまとめるのに適した部分のみを取り出したコードクローンを**シェイプドクローン**と呼ぶ．ソフトウェアからシェイプドクローンを取り出す過程は以下の 3 つからなる．

(1) 対象のソフトウェアに対し CCFinder を実行し，コードクローンを検出する．
(2) 次に，対象のソフトウェアを解析し，構造的なまとまりをもったブロックの位置情報を抽出する．たとえば，Java プログラムを対象とした場合は，構造的なまとまりを持ったブロックとは具体的には以下で示すものである．
　宣言：class { }，interface { }
　メソッド：メソッド，コンストラクタ
　文：if 文，for 文，while 文，do 文，switch 文，try 文，synchronized 文，static 文
　ブロック：‘{’ と ‘}’ で囲まれた範囲
(3) 最後に，コードクローンの情報と，ブロックの位置情報を突き合わせ，ユーザに指定された最小トークン数以上のクローンをシェイプドクローンとして取り出す．

CCFinder のコードクローン検出過程では，ソースコードに含まれるトークン数を n，検出された最も長いクローンのトークン数を t とした場合，$O(nt)$ 時間で解析結果を得ることが可能である，詳しくは文献 13) を参照していただきたい．CCFinder の検出したコードクローンからシェイプドクローンを抽出する解析では，$O(n)$ の処理時間で対象ソースコードを構文解析しソースコード中の構造的なまとまりのある部分を抽出できる．また $O(cs\ log\ c)$（c は 1 つのファイルあたりに含まれるコードクローンの数，s は対象ソースコードの数，c，s いずれも n に対して非常に小さい値となる）の処理時間で CCFinder の検出したコードクローン情報と，ソースコード中の構造的なまとまりのある部分から，シェイプドクローンを抽出できる．

他のアプローチ，たとえばプログラム依存グラフを

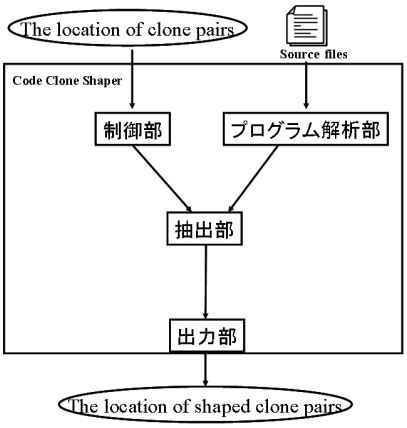

図 6 Code Clone Shaper 概要
Fig. 6 System of Code Clone Shaper.

用いる手法と比較した場合，プログラム依存グラフ構築の時間コストが $O(m^2)$（m はソースコード中の文や式の数，m は n と比例関係にある）である[11] ことを考慮すると，この提案手法では，対象が大規模なソフトウェアであっても実用的な時間でリファクタリングの容易なコードクローンを検出できると期待できる．

3.3 実　　装

ソースコードからシェイプドクローンを取り出す機能を Gemini に実装した（**図 6** 参照）．ユーザは Gemini の GUI の一部であるオプション設定ウィンドウからシェイプドクローン抽出の有無を設定し，抽出するシェイプドクローンの最小トークン数も指定することが可能である．このオプション機能を設定することでユーザは構造的なまとまりを持つコードクローンを Gemini を用いて分析できる．Gemini の詳しい使用方法は，文献 21) を参照していただきたい．

この機能は約 1 万 2 千行の Java プログラムで実装されている．現在のところ，対象言語は Java のみである．ここでは，実装したシェイプドクローン検出プログラム（Code Clone Shaper 以後 CCShaper と略す）について説明する．実装したプログラムは図 6 に示されるように大きく分けて 4 つのモジュール，制御部，プログラム解析部，抽出部，出力部から構成されている．

制　御　部

制御部は CCFinder の出力したコードクローン情報を解析し，プログラム解析部や，抽出部，管理部を呼び出す役割を担っている．

プログラム解析部

プログラム解析部では対象のソースコードを字句解析，構文解析して，プログラム中の構造的なまとまりを持つブロックの位置情報を取得する．本手法におけるブロックの位置情報はソースコードのファイルパス，ファイル内でのブロックの開始行，開始列，終了行，終了列の 5 つの情報からなる．ソースコードのファイルパスは，ユーザから指定されたファイルに対して解析を行っているのですでに取得済みである．残りの 4 つの値はソースコードを構文解析する過程で取得される．なお，このモジュールは JavaCC[10] を用いて構築されている．

抽　出　部

プログラム解析部で抽出したブロックの位置情報と，CCFinder の出力結果であるコードクローン情報を突き合わせ，コードクローンである部分からプログラミング言語の意味にてらして構造的にまとまりのある部分のみを抽出する．ブロックの位置情報と同様にコードクローン情報もソースコードのファイルパス，開始行，開始列，終了行，終了列の 5 つの情報からなる．つまり，CCFinder の検出したコードクローン内に含まれる一番外側の構造的なまとまりを持ったブロックがシェイプドクローンとして抽出される．

出　力　部

抽出部で取り出したシェイプドクローンを適切な順番で並べ変え，Gemini に与えるデータとして一貫性のあるものに変換する．

3.4 リファクタリング方法

本節では，提案機能を使用したリファクタリング方法を説明する．リファクタリングの手順は以下の STEP1〜STEP3 からなる．

STEP1（クローン検出）：Gemini を用いて対象ソースコードからシェイプドクローンを検出する．

STEP2（クローン選択）：Gemini のクローン散布図，メトリクスグラフを用いてリファクタリング対象コードクローンを特定する．

STEP3（クローン除去）：STEP2 で選択したコードクローンを除去する．

STEP1 については，Gemini の GUI の一部であるオプション設定ウィンドウからシェイプドクローン抽出機能の設定ができることは 3.3 節で述べた．

STEP2 では，検出したシェイプドクローンをクローン散布図とメトリクスグラフを用いて分析し，リファクタリング対象となるコードクローンを決定する．

クローン散布図を用いた場合には，クローンが密集しているファイルのコードクローンを取り除くことが有効である．これまでに行われたケーススタディ[17] では，各ファイルにおいてそのファイルのクローンとなっている行の割合，ファイル中の最大長クローンとその

ファイルの改版数を定量的に比較し，一定以上の割合でクローンが含まれている場合や，非常に長いコードクローンが含まれていた場合，そのファイルの改版数が多くなることが示されている．つまり，クローン散布図を用いてクローンが密集しているファイルのコードクローンをリファクタリング対象として選択し除去した場合，長期的なそのファイルの改版数はリファクタリングしない場合に比べ少なくなることが予測され，保守コストの削減が期待できる．

一方，メトリクスグラフを用いた場合には各クローンクラスの特性に基づいてリファクタリング対象コードクローンを選択することが可能である．メトリクスグラフではクローンクラスを以下の 4 つのメトリクスを用いて特徴づけている．

RAD(C)：クローンクラス C のディレクトリ階層における分散度を示すメトリクスである．クローンクラス C 中のすべてのコード片がある 1 つのファイル内に存在する場合は 0，1 つのディレクトリ内に存在する場合は 1 と，広い範囲で分散しているほどこのメトリクス値は大きくなる．

LEN(C)：クローンクラス C に含まれるコード片の最大長を表す．

POP(C)：クローンクラス C に含まれるコード片の数を表す．

DFL(C)：クローンクラス C に含まれるコード片に共通する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の呼び出しに置き換えた場合の減少が予測されるトークン数である．

たとえば，将来的にバグが見つかった場合の修正コストをなるべくおさえたい場合は，POP 値の大きいクローンクラスを除去することが有効である．また DFL 値の大きいクローンクラスを除去することは，ソースコードの行数を多く減少させることを意味するので，ソフトウェアの規模を小さくするという意味で有効である．

STEP3 では，ユーザはまず STEP2 で選択したコードクローンのソースコードを Gemini のソースコードビューを用いて閲覧し，どのように除去するのかを決定しなければならない．たとえば，選択したクローンクラスのコード片のすべてもしくは大半が 1 つのクラス内に存在する場合は，そのクローンクラスのコード片に共通するロジックを新しいメソッドとして実装し，各コード片をそのメソッドの呼出文に置き換えることが有効である．また，Fowler が提唱しているリファクタリングパターン[8] の中にもクローンの除去に適用可能なパターンが存在する．たとえば，複数のクラスにメソッド単位のコードクローンが存在した場合を考える．もしそれらのクラスが共通の親クラスを継承しているならば，クローンとなっている各メソッドに対して “Pull Up Method” パターンを適用することで，クローンを除去できる．除去方法を決定した後は，ユーザはテキストエディタなどを用いて，実際にソースコードに変更を加え，コードクローンを除去する．

表 1 ソースコードサイズ
Table 1 Source code size.

	ファイル数	行数	トークン数
ANTLR	239	43,548	140,802
Ant	508	141,254	221,203

4. 適用実験

4.1 実験概要

前章で提案したシェイプドクローンの検出方法の有用性を確認するためにオープンソースの Java で記述されたソフトウェアである ANTLR[2] と Ant[1] に対して適用実験を行った．

ANTLR（ANother Tool for Language Recognition）は構文解析器を生成するためのツールである．生成された構文解析器は Java もしくは C++で出力される．Ant は Java 用のビルドツールであり，ビルド手順は XML で記述される（**表 1**）．

提案手法の評価をするために，CCShaper を用いた Gemini と用いていない Gemini を用いてそれぞれ対象のソースファイルを解析し，その結果を比較した．この適用実験では，CCFinder が検出するコードクローンの最小一致トークン数は 50，CCShaper が抽出するコードクローンの最小トークン数は 30 に設定した．

またこの適用実験では，クローン散布図を用いてリファクタリングを行うコードクローンを決定した．文献 17) では，コードクローンを一定以上含むファイルが他のファイルに比べ改版数が多くなることを示している．クローン散布図でクローンが密集している部分はコードクローンを多く含むファイルであるため，このようなファイル中に存在するコードクローンをリファクタリングすることにより，そのファイルの保守性を高めることができる．

4.2 ANTLR

ANTLR は 239 個のソースファイルから構成されており，総行数は 4 万 4 千行である．

CCShaper の適用前，適用後のクローン散布図をそれぞれ**図 7**，**図 8** に示す．また，適用前後のコードクローン数の比較を**表 2** に示す．CCShaper を用いない場合には，ANTLR 内に発見されるコードクローンは

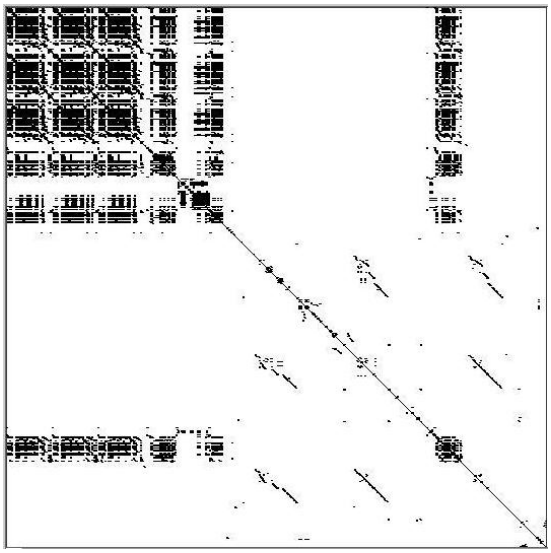

図 7 ANTLR のクローン散布図（CCShaper なし）
Fig. 7 Result without Code Clone Shaper in ANTLR.

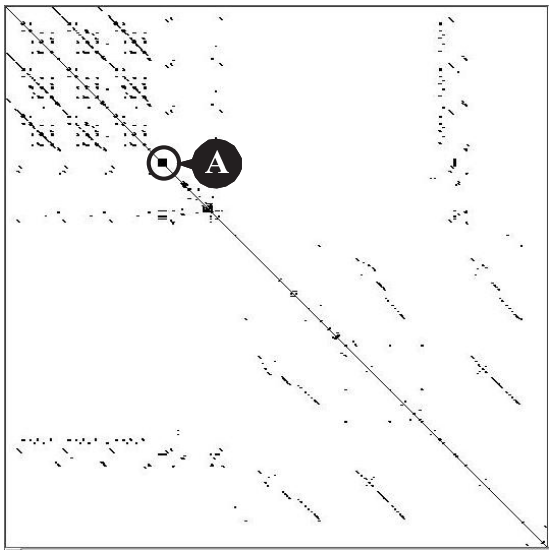

図 8 ANTLR のクローン散布図（CCShaper あり）
Fig. 8 Result with Code Clone Shaper in ANTLR.

表 2 ANTLR から検出されたコードクローン数の比較
Table 2 Numbers of clones in ANTLR.

	CCShaper なし	CCShaper あり
クローンペア数	338,574	972
クローンクラス数	1,072	142

非常に多くなってしまい，どのコードクローンを取り除くべきか，取り除くことができるかを判断するのに，非常に手間がかかってしまうと考えられる．一方で，CCShaper を用いた場合には，リファクタリングが難しいコードクローンを取り除くことにより，クローンペアの 99%以上（33 万個以上），クローンクラスの約 85%（830 個）を省くことができている．

また，どのようなコードクローンが検出されているのかを調べるために，図 8 において最もクローンペアが密集しているラベル A で示されている部分のソース

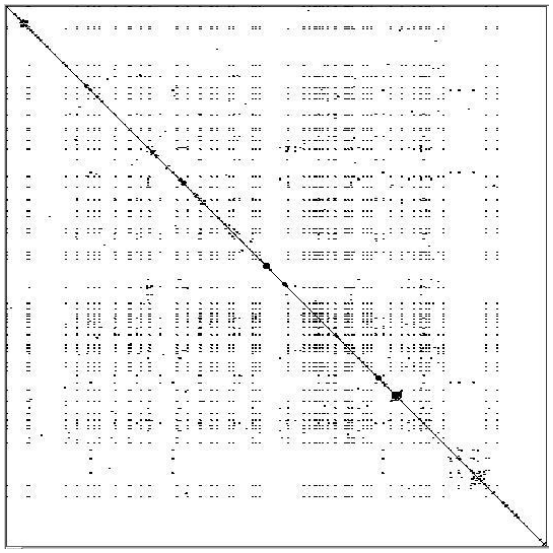

図 10 Ant のクローン散布図（CCShaper なし）
Fig. 10 Result without Code Clone Shaper in Ant.

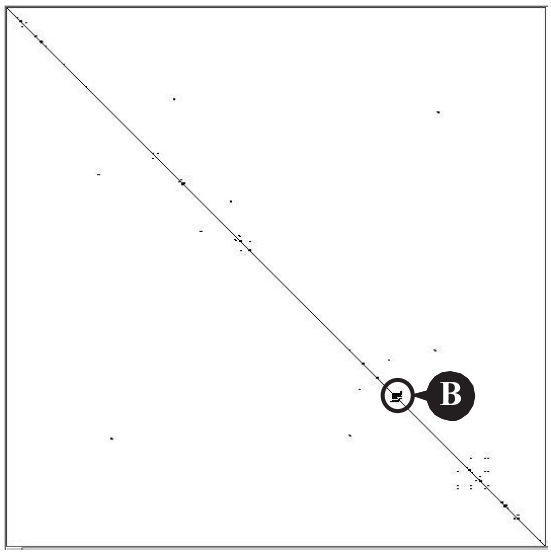

図 11 Ant のクローン散布図（CCShaper あり）
Fig. 11 Result with Code Clone Shaper in Ant.

表 3 Ant から検出されたコードクローン数の比較
Table 3 Numbers of clones in Ant.

	CCShaper あり	CCShaper なし
クローンペア数	12,033	103
クローンクラス数	856	53

```
public final void mOPEN_ELEMENT_OPTION(boolean _createToken)
throws RecognitionException, CharStreamException, TokenStreamException {
    int _ttype;
    Token _token=null;
    int _begin=text.length();
    ttype = OPEN_ELEMENT_OPTION;
    int _saveIndex;

    match('<');
    if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
        _token = makeToken(_ttype);
        _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
    }
    _returnToken = _token;
}
```

```
public final void mCLOSE_ELEMENT_OPTION(boolean _createToken)
throws RecognitionException, CharStreamException, TokenStreamException {
    int _ttype;
    Token _token=null;
    int _begin=text.length();
    _ttype = CLOSE_ELEMENT_OPTION;
    int _saveIndex;

    match('>');
    if ( _createToken && _token==null &&_ttype!=Token.SKIP ) {
        _token = makeToken(_ttype);
        _token.setText(new String(text.getBuffer(), _begin,  ext.length()-_begin));
    }
    _returnToken = _token;
}
```

```
public final void MERGED_METHOD(boolean _createToken,int i,char c)
throws RecognitionException, CharStreamException, TokenStreamException {
    int _ttype;
    Token _token=null;
    int _begin=text.length();
    _ttype = i;
    int _saveIndex;

    match(c);
    if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
        _token = makeToken(_ttype);
        _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
    }
    _returnToken = _token;
}
```

```
public final void mRCURLY(boolean _createToken)
throws RecognitionException, CharStreamException, TokenStreamException {
    int _ttype;
    Token _token=null; int _begin=text.length();
    _ttype = RCURLY;
    int _saveIndex;

    match('}');
    if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
        _token = makeToken(_ttype);
        _token.setText(new String(text.getBuffer(), _begin, text.length()-_begin));
    }
    _returnToken = _token;
}
```

図 9 ラベル A のコードクローンのソースコードとリファクタリング例
Fig. 9 Merged clone sample in ANTLR.

コードを調べ，**図 9** に示す特徴的なクローンを発見した．このクローンクラスは 28 個のコード片から成り立っており，それぞれのコード片の長さは 82 トークンであった．またこのクローンクラスのコードクローンはすべてメソッド単位のクローンであり，コード片の差異は図 9 の灰色で示されている部分 2 カ所の定数名の違いのみであった．このクローンクラスのすべてのコード片はすべて同一のクラス内に存在しているメソッドであり，引数を 2 つ追加することによって容易に 1 つのメソッドにまとめることができる．

4.3 Ant

次に Ant に対しての実験結果を示す．Ant は 508 個のファイルから成り立っており，総行数は 14 万 1 千行であった．

CCShaper を適用する前，適用後のクローン散布図をそれぞれ**図 10**，**図 11** に示す．また，適用前後のコードクローン数の比較を**表 3** に示す．CCShaper を用いない場合には，Ant 内には一様にコードクローンが散布しており，どのコードクローンを取り除くべきか，取り除くことができるかを判断するのに，非常に手間がかかってしまうと考えられる．一方で，CCShaper を用いた場合には，リファクタリングが難しいコードクローンを取り除くことにより，クローンペアの 99%以上（1 万個以上），クローンクラスの約 94%（803 個）を省くことができている．

実際に検出されたコードクローンのソースコードを調べるために，最もクローンペアが密集しているラベル B で示す部分をソースコードビューで閲覧する．

```
public void getAutoresponse(CommandLine cmd) {
  if (m_AutoResponse == null) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } else if (m_AutoResponse.equalsIgnoreCase("Y")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_YES);
  } else if (m_AutoResponse.equalsIgnoreCase("N")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_NO);
  } else {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } // end of else
}
```

図 12 ラベル B のコードクローンのソースコード
Fig. 12 Entirely same clone in Ant.

図 12 はこの部分のコードクローンのソースコードを示している．このメソッド単位のコードクローンは 7 つのクラスに，各々1 つずつ存在した．またこれら 7 つのクラスはすべて同一のクラスを親クラスとして定義していた．これらのことからこのコードクローンは親クラスに引き上げることによって取り除くことができる．

5. まとめ

本論文では，既存のコードクローン分析環境 Gemini を拡張してリファクタリングの候補となりうるコードクローンのみを抽出する機能（Code Clone Shaper）を提案した．提案手法をツールとして実装し，オープンソースのソフトウェアである ANTLR，Ant に適用した．Code Clone Shaper を適用することでコードクローンをフィルタリングすることに成功し，抽出されたコードクローンはいずれもリファクタリングの容易なものであった．これらのことから本論文で提案したコードクローン抽出手法は非常に有用であると考えることができる．しかし，この提案手法ではコードクローンをリファクタリングしやすい単位として抽出したものであり，抽出した各クローンがどのようにリファクタリングできるのかまでは考慮していない．今後は抽出したクローンをどのようにリファクタリングできるのかによって分類する手法の研究を行うとともに，このツールを実際のソフトウェア保守の場面で利用し，そのプロセス改善に役立てたいと考えている．

謝辞 CCFinder，Gemini の評価について多くのフィードバックをいただいた．CCFinder メーリングリストの参加者に感謝します．

参考文献

1) Ant 2002. http://jakarta.apache.org/ant/
2) ANTLR 2000. http://www.antlr.org/
3) Baker, B.S.: A Program for Identifying Duplicated Code, *Computing Science and Statistics*, Vol.24, pp.49–57 (1992).
4) Baker, B.S.: On Finding Duplication and Near-Duplication in Large Software Systems, *Proc. IEEE Working Conf. on Reverse Engineering*, pp.86–95 (July 1995).
5) Baker, B.S.: Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance, *SIAM Journal on Computing*, Vol.26, No.5, pp.1343–1362 (1997).
6) Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM*) *'98*, Bethesda, Maryland, pp.368–377 (Nov. 1998).
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*, Oxford, England, pp.109–118 (Aug. 1999).
8) Fowler, M.: *Refactoring: improving the design of existing code*, Addison Wesley (1999).
9) 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).
10) JavaCC 2003. http://javacc.dev.java.net
11) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, Issue.3, pp.319–349 (1987).
12) Johnson, J.H.: Identifying Redundancy in Source Code using Fingerprints, *Proc. CASCON '93*, Toronto, Ontario, pp.171–183 (1993).
13) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
14) Komondoor, R. and Horwitz, S.: Using slicing to identify duplication in source code, *Proc. 8th International Symposium on Static Analysis*, Paris, France (July 16–18, 2001).
15) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. 8th Working Conference on Reverse Engineering* (2001).
16) Mayland, J., Leblanc, C. and Merlo, E.M.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM*) *'96*, Monterey, California, pp.244–253 (Nov. 1996).
17) 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，情報処理学会論文誌，Vol.44, No.8, pp.2178–2187 (2003).

18) Prechelt, L., Malpohl, G. and Philippsen, M.: Finding plagiarisms among a set of programs with JPlag, submitted to Journal of Universal Computer Science (Nov. 2001), taken from http://wwwipd.ira.uka.de/~prechelt/Biblio/.
19) Rieger, M. and Ducasse, S.: *Visual Detection of Duplicated Code* (1998).
20) Pigoski T.M.: Maintenance, *Encyclopedia of Software Engineering*, 1, John Wiley & Sons (1994).
21) Ueda, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Gemini: Maintenance Support Environment Based on Code Clone Analysis, *8th International Symposium on Software Metrics* (June 4–7, 2002).
22) 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，電子情報通信学会論文誌 D-I, Vol.86-D-I, No.12, pp.863–871 (2003).
23) Ueda, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: On Detection of Gapped Code Clones using Gap Locations, Submitted to 9th Asia-Pacific Software Engineering Conference (2002).
24) Yip, S.W.L. and Lam, T.: A software maintenance survey, *Proc. APSEC '94*, pp.70–79 (1994).

(平成 15 年 8 月 13 日受付)
(平成 16 年 3 月 5 日採録)

肥後　芳樹（学生会員）
平成 14 年大阪大学基礎工学部情報科学科中退．平成 16 年同大学大学院博士前期課程修了，現在同大学院博士後期課程 1 年．コードクローン分析の研究に従事．

植田　泰士
平成 13 年大阪大学基礎工学部情報科学科卒業．平成 15 年同大学大学院博士前期課程修了．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（正会員）
平成 8 年大阪大学基礎工学部情報工学科中退．平成 13 年同大学大学院博士課程修了．現在，科学技術振興機構さきがけ研究者．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．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

楠本　真二（正会員）
昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同大学講師．平成 11 年同大学助教授．平成 14 年大阪大学基礎工学部情報工学科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．IEEE，電子情報通信学会，JFPUG，PM 学会各会員．

井上　克郎（正会員）
昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59 年〜昭和 61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．工学博士．平成 14 年大阪大学基礎工学部情報工学科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．日本ソフトウェア科学会，IEEE，電子情報通信学会，ACM 各会員．