修士学位論文

題目

ソフトウェア資源の機能整理を目的とした類似関数集合の検出

指導教員
楠本 真二 教授

報告者
田中 健介

平成 22 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 21 年度 修士学位論文

ソフトウェア資源の機能整理を目的とした類似関数集合の検出

田中 健介

内容梗概

近年ソフトウェア開発は大規模化の一途をたどっており，開発コスト削減のため，既存の資源を有効に活用することが望まれる．しかし，再利用を行うためにはソースコードの内容を理解しなければならず，特に，再利用を考慮して作成されていない資源の場合，その労力は大きい．ソフトウェア分析手法としてコードクローン（ソースコード中の類似したまたは同一のコード片の集合）検出が注目を集めている．既存のソフトウェアにはこのコードクローンが多数存在し，同様の処理を行う関数が散在している．本研究では，これまでに実装された機能を効率的に把握・再利用するため，コードクローン検出技術を用いて，大量のソースコードから同様の機能を実現した関数集合群の検出を試みる．しかし，既存のクローン検出ツールには対象にできるソースコード量の制限，関数単位のクローン検出機能が無いなどの問題があり，ツールをそのまま適用するだけでは，大量のソースコードから類似関数群を得ることはできない．この問題を解決するために，対象ソースコードを分割し，分割した単位ごとに関数クローンの検出を行い，その後に検出結果を 1 つにまとめる．本手法をオープンソースソフトウェア群に対して適用した実験の結果，数千万行のソースコードからどのような関数が多数利用されているのか容易に調査することができた．さらに，クローン数や存在箇所を調べることにより，ソースファイル単位のクローンの発見やクローンの特徴別の分類をすることができた．

主な用語

プログラム解析
コードクローン
関数分類
再利用

目 次

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，ソフトウェア開発に要するコストが増加してきている．ソフトウェア開発コストの削減方法として，これまでに作成したソフトウェアやオープンソースソフトウェアなどの既存の資源の再利用が挙げられる．しかし，再利用を行うためには内容を理解しなければならず，特に，再利用を考慮して作成されていない資源の場合，その労力は大きい．

ソフトウェアの保守作業を困難にする要因の 1 つ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，ソースコード上に存在する同一もしくは類似したコード片のことであり，主にコピーアンドペースト等によって発生するといわれている [1]．コピーアンドペーストを行った際，コピー先に合わせた識別子（変数名や関数名）の変更や，文単位での追加および削除などの修正作業がしばしば行われる．この作業により，同様の機能を実装した処理が複数作りこまれる．このようにしてコピーアンドペーストを繰り返し開発を行うと，類似した処理の把握が困難になる．さらに，あるコード片に不具合が存在した場合や，保守変更が必要になった場合，その全てのコードクローンに対して同様の修正を行わなければならない．

本研究では，コードクローン情報を用いて，これまでに実装された機能を効率的に把握・再利用するため，大量のソースコードから同様の機能を実現した関数群を検出する手法を提案する．コードクローンは類似したコード片であるので，その処理内容も類似している．つまり，既存のソースコードからコードクローンを検出，クローンセットを生成することによって類似した処理集合を得ることができる．クローンセットは互いに類似したコード片の集合を表し，コード片を 1 つの関数として検出すると，クローンセットは同様の機能を実現した関数集合と考えることができる．既存のクローン検出ツールを用いてコードクローンを検出する場合，対象にできるソースコード量の制限，関数単位のクローン検出機能が無いなどの問題がある．大量のソースコードを対象としたクローンペアの検出手法 [19][20] はこれまでに提案されているが，クローンセットの生成を行う手法は提案されていない．本研究では，大量のソースコードを複数のグループに分割し，それぞれのグループについて関数単位のクローン (関数クローン) を検出，結果を組合わせることによって，関数単位のクローンセット (関数クローンセット) の生成を試みた．関数クローンの検出は，関数にまたがって存在するコードクローンに注目し，それらが関数内のコードに占める割合を求めることによって行う．また，複数の計算機で並列計算することにより，検出時間の短縮を行った．

実験の結果，2,700 万行のソースコードから約 15 時間で関数クローンセットを生成することができ，どのような関数が多数利用されているのか容易に調査することができた．また，類似した関数の数や関数が存在するソフトウェア数などを基にしたメトリクスを利用して関

数クローンセットを分類することにより，ソースファイル単位のクローンも発見することができた．本手法を利用して散在する類似関数の整理を行うことにより，ソースコードの内容理解やソフトウェアの保守性の向上支援が期待できる．

以降，2 章ではコードクローンに関する説明を行う．続いて，3，4，5 章では提案手法について述べる．6 章では，適用実験について説明し，その考察を 7 章で行う．8 章で関連研究について述べ，最後に 9 章でまとめと今後の課題を記す．

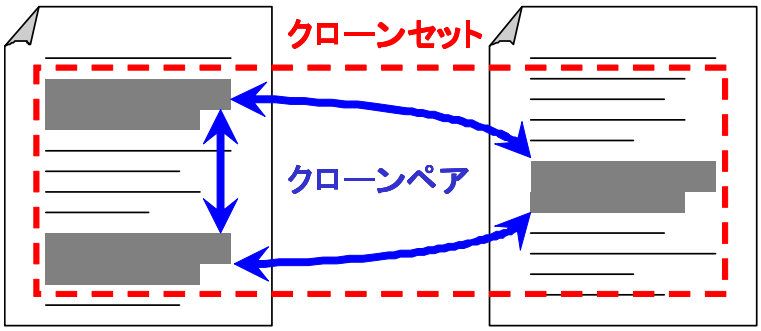

図 1: クローンペアとクローンセット

2 コードクローン

2.1 概要

コードクローン [1] 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．コードクローンは，コピーアンド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，ソースコード中に作りこまれる．互いに類似する 2 つのコード片をクローンペア，互いに類似するコード片の集合をクローンセットと呼ぶ (図 1)．一般に，コードクローンはソースコードに対する一貫した変更を難しくするといわれている．例えば，あるコードにバグが発見され，そのコードとコードクローンになっている箇所が存在した場合，すべてのコードクローンに同様の修正を行う必要がある．大規模なシステムの場合，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．コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある．

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存コードの再利用が多く行われるようになった．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語の機能的な制限

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS(Linux, FreeBSD, HP-UX や AIX など) や CPU(i386 系, amd64 系, alpha や sparc64 など) に対応したソフトウェアは，各プラットフォーム用のコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.2 コードクローン検出手法

これまでにさまざまなコードクローン検出手法が提案されており，それぞれの手法によって特徴が異なっている．

CCFinder[2]，CCFinderX[3]

ユーザ定義名のパラメータ化を行なった後，サフィックス木探索アルゴリズム [4] を用いてコードクローンを検出する．C/C++，Java，COBOL，FORTRAN など複数のプログラミング言語に対応している．

Deckard[5]

抽象構文木（AST）の各部分木を配列表現に変換し，局所感度ハッシュアルゴリズム [6] を用いて類似配列を求めることによって，コードクローンを検出する．

Dude[7]

行単位でコードクローンを検出した後，近接するコードクローンを 1 つにまとめることによってコードクローンを検出する．

CP-Miner[8]

ソースコードの字句解析と構文解析を行い，ユーザ定義名のパラメータ化を行なった後，頻出系列マイニングアルゴリズム [9] を用いてコードクローンを検出する．

CloneDR[10]

抽象構文木を比較することによって，コードクローンの検出を行う．部分的に異なっているコードクローンも検出することが可能である．

Dup[11, 12, 13, 14]

ユーザ定義名のパラメータ化を行なった後，行単位の比較によりコードクローンを検出する．マッチングアルゴリズムにはサフィックス木探索を用いているため線形時間で解析可能である．

Duploc[15]

各行に含まれる空白やタブ，コメント等を取り除いた後，行単位の比較によりコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行なう．

Duplix[16]

Fine-grained PDG というグラフ上での類似部分グラフを検出することでコードクローンを検出する．

CLAN[17]

関数に対して 21 種類のメトリクスを計測することによって，コードクローンの検出を行なう．

いずれの手法，ツールにおいても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，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出

アルゴリズムそのものによって定義される．Burd らは，複数のツールを用いて，それぞれ検出されるコードクローンの比較を行なっている [18]．それによれば，全ての面において他のツールよりも優れているツールは存在せず，ユーザが使う場面に応じて，適切なツールを選ぶことが必要であると述べている．

2.3 CCFinder

CCFinder は，単一または複数のファイルのソースコード中から全ての極大クローン検出し，それをクローンペアの位置情報として出力する．ある 2 つの字句列 (a, b) がクローンペアで，それぞれの字句列を真に包含する如何なる字句列も等価でないとき，(a, b) を極大クローンと呼ぶ．CCFinder の持つ主な特徴は次のとおりである．

細粒度のコードクローンを検出

字句解析を行うことにより，字句単位でコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば，Apache Ant（1,122 ファイル，237,279 行）のソースコードを約 30 秒（実行環境 Pentium4 3.00GHz RAM 2GB）で解析可能である．

様々なプログラミング言語に対応可能

言語依存部分を切り替えることで，様々なプログラミング言語に対応できる．現在は，C，C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応言語に対しても完全一致判定によるコードクローンは検出可能である．

実用的に意味の持たないコードクローンを取り除く

コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致字句数を指定することができるため，そのようなコードクローンの検出を防ぐことができる．また，モジュールの区切りを認識し，一連の意味のある処理の流れのみをコードクローンとして検出する．

ある程度の違いは吸収可能

コピーアンドペーストし，少々コードに変更を加えたとしても同様のバグが混入している場合がある．このような場合でもコードクローンとして抽出できれば，同じ対処でバグ修正が可能となるなどのメリットがある．下記にどのような違いを吸収可能であるかについて示す．

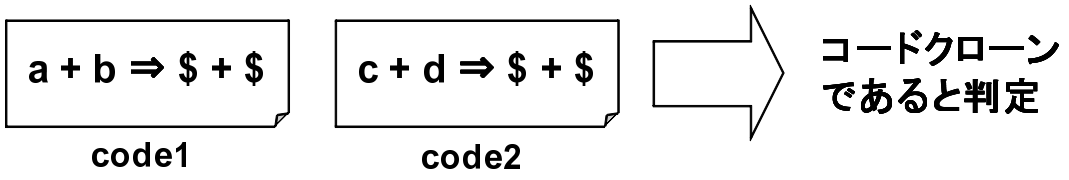

図 2: パラメータ化による比較

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．図 2 に例を示す.
- 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ことで，その違いを吸収できる.
- その他，テーブル初期化コード，可視性キーワード（pulrotected，public，private 等），コンパウンド・ブロックの中括弧表記等の違いも吸収できる.

3　関数クローンペア

3.1　定義

2 つの関数間で字句列が閾値以上等価であるとき，互いに関数クローンペアと定義する．関数 A，関数 B が式 (1)(2)(3) を満たす場合，互いに関数クローンペアになる．

$$DUP(A,B) = \frac{CL(A,B)}{LOC(A)} \tag{1}$$

$$DUP(B,A) = \frac{CL(B,A)}{LOC(B)} \tag{2}$$

$$min(DUP(A,B), DUP(B,A)) \geq THRESHOLD \tag{3}$$

A, B ： 関数 A, 関数 B

$DUP(A,B)$ ： 関数 A の関数 B に対する重複度

$CL(A,B)$ ： 関数 A において関数 B とコードクローンになっている行数

$LOC(A)$ ： 関数 A の行数

$THRESHOLD$ ： 関数重複度の閾値

この定義からわかるように，閾値を満たす関数 A，関数 B は互いにソースコードの類似したペアである．

3.2　検出手順

関数クローンペアの検出手順を以下に示す．

Step 1 : ソースコードから関数情報を取得する．

Step 2 : ソースコードからコードクローンを検出する．

Step 3 : 関数情報，コードクローン情報を式 (1)(2)(3) に適応し，閾値を満たす関数のペアを関数クローンペアとして検出する．

各 STEP の詳細を次に述べる．

Step 1 : 関数情報の取得

関数クローンペアを検出するために，ソースコードから関数の位置や行数などの情報を抽出する必要がある．本研究では Ctags[21] を利用する．Ctags の入力・出力を下記に示す.

入力: ソースファイル

出力: 関数やクラス，構造体などの名前，型，行番号，ソースファイル名，定義行

上記の出力から関数名と開始行，ソースファイル名を取得し，終了行を中括弧をカウントすることによって取得する.

Step 2 : コードクローンの検出

これまでに様々なコードクローン検出手法が提案されている．今回は大量のソースコードからコードクローンを検出するので，適用対象が大きくなる．そのためスケーラビリティの高い CCFinder を用いる．CCFinder の入力・出力を下記に示す.

入力: ソースコード

出力: コードクローンとその開始位置・終了位置

また，オプションにより，検出するクローンの長さや検出領域などの指定することができる．オプションには様々な種類があるが，本手法に関連のあるものを下記に示す.

Minimum Clone Length

Language Select

Detection Option

Parameterize Option

Minimum Clone Length はコードクローンとして検出する字句列の最小の長さ，Language Select は対象プログラミング言語，Detection Option はどのような範囲でクローンペアを検出するのかを指定，Parameterize Option は字句をパラメータ化するか指定できる．パラメータ化に関しては 2.3 節で述べている．以降，字句をパラメータ化せずにコードクローンを検出することを完全一致検出，字句をパラメータ化してコードクローンを検出することを名前変更検出と呼ぶ．完全一致検出は字句列が等価であっても，識別子が異なる場合，コードクローンであると判定されない．そのため，コピーアンドペーストされた後に識別子が変

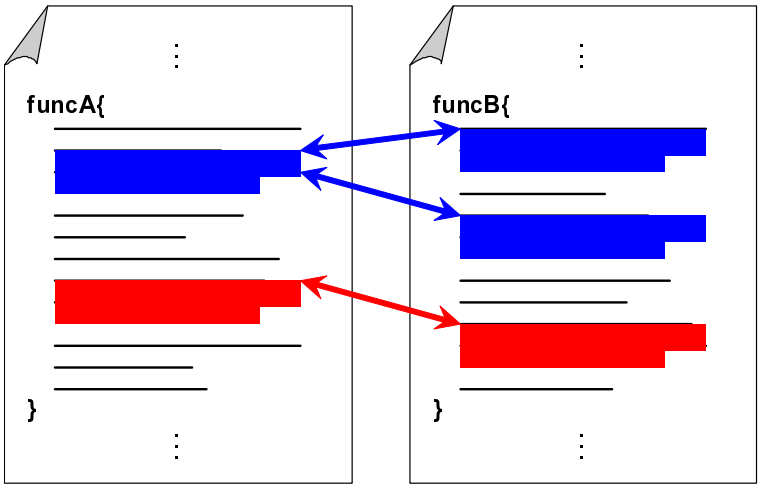

図 3: 関数間のクローンペア

更されたものはコードクローンとして検出されなくなる．名前変更検出は識別子が異なっていてもコードクローンであると判定されので，コピーアンドペーストされた後に識別子の変更があってもコードクローンとして検出できる．しかし，誤検出をする確率は上がってしまう．つまり，完全一致検出は低再現率，高適合率でやや少ないコードクローンを，名前変更検出は低再現率，高適合率で多くのコードクローンを検出することができる．

Step 3 : 関数クローンペアの検出

2 つの関数 A と関数 B が関数クローンペアになるか調べる手順を下記する．

1. 関数とクローンの位置情報から関数 A と関数 B 間でコードクローンになっているクローンペアを取得 (図 3).
2. それぞれの関数についてコードクローンが関数内のコード行数の何%を占めているか調べる．
3. 関数 A と関数 B のコードクローンが占める行の割合が式 (3) を満たす場合，関数 A と関数 B は互いに関数クローンペアとなる．

以上の手順をソースコードから検出した全ての関数の組合せで調べる．

3.3 並列計算を用いた実装

本研究では大量のソースコードを検出対象とするため，一度に対象全体からコードクローン検出を行うことは不可能である．このため，検出対象をソースファイル群の小さなグルー

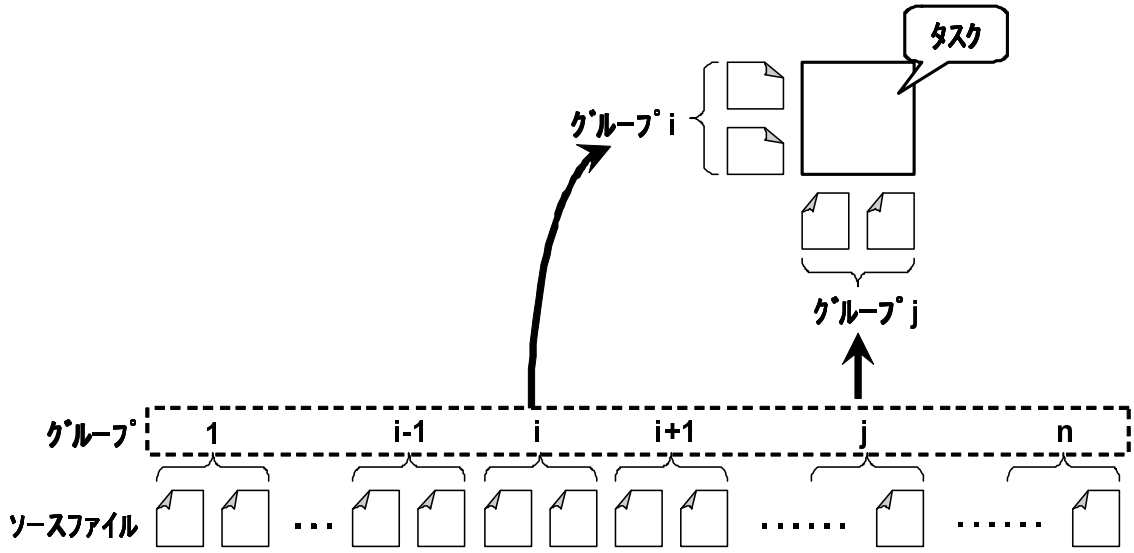

図 4: 計算モデル

プに分割し，グループの組合せ (タスク) 単位で関数クローンを検出する (図 4)．検出対象のソースファイル群を n 個に分割しているとする．このとき，任意のタスクは (i, j) で表すことができる（ただし，$1 \leq i, j \leq n$）．タスク (i, j) に含まれるコードクローンはタスク (j, i) に含まれるコードクローンと同様であるため，後者については検出を行わない．これにより，関数クローンを検出するタスク数は $n(n+1)/2$ となる．

本提案手法は，関数クローンペアの検出を複数のコンピュータで並列に実行することにより，計算時間の短縮を行う．各タスクの演算（関数クローンペア検出）は他のタスクの演算結果に全く依存しないため，タスクの割り当て処理は単純に行える．関数クローンペアが未検出のタスクを，アイドル状態のコンピュータに割り当て，検出結果を回収するだけでよい．この方法は Livieri ら [19][20] の手法を参考にした．

演算はタスク管理用計算機 1 台，ソースファイル保存用計算機 1 台，関数クローンペア検出用計算機複数台で行う．それぞれの計算機の役割を下記に示す．

タスク管理用計算機

図 5 に示すように計算タスクを管理し，アイドル状態の関数クローンペア検出用計算機にタスクを割り当てる．また，関数クローン検出用計算機からの演算の結果を受け取り，結果を集約する役割も持つ．

ソースファイル保存用計算機

計算対象のソースファイル群を保持する．関数クローンペア検出用計算機はこの計算

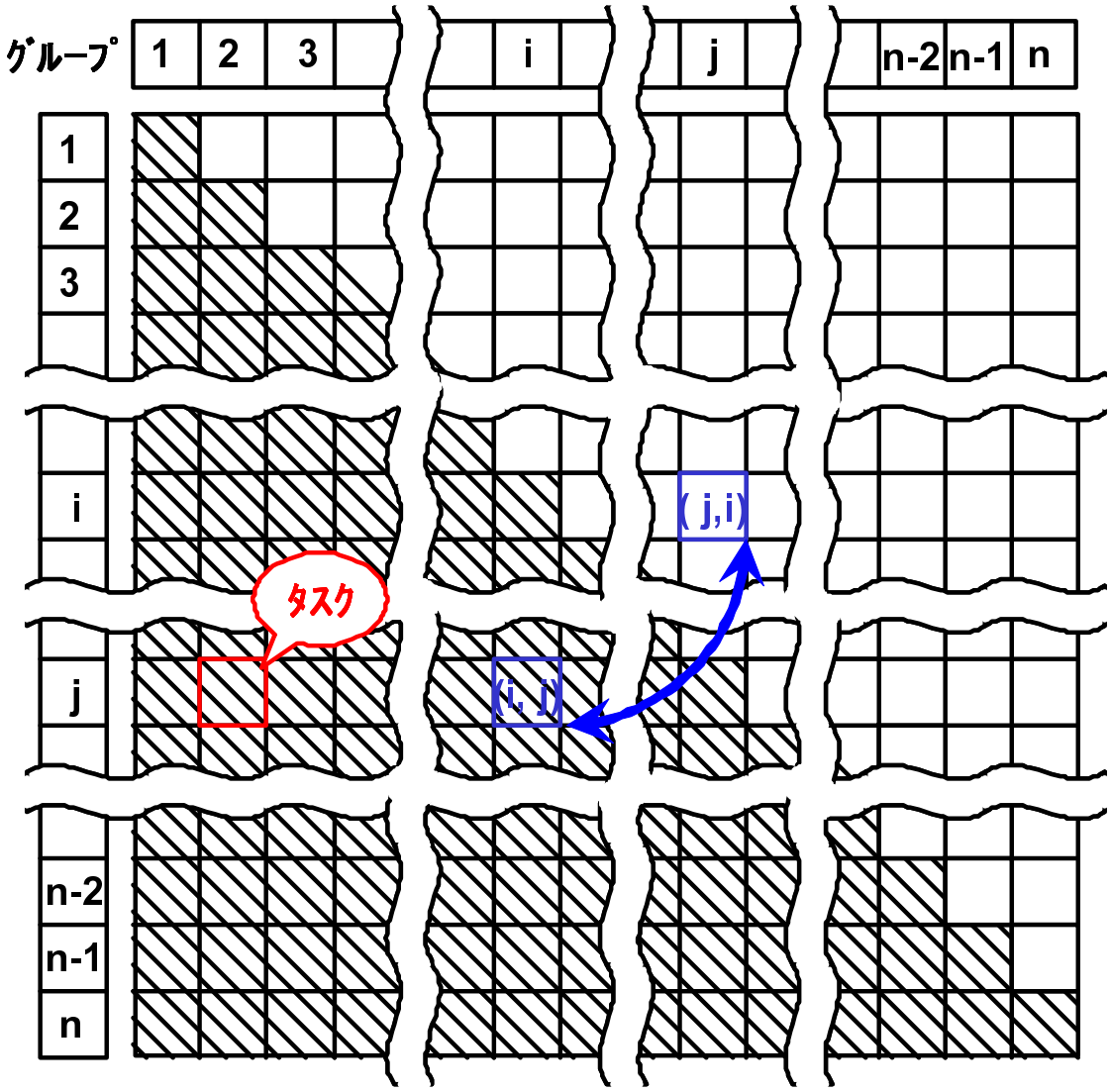

図 5: タスクマップ

機からソースファイルをダウンロードする．

関数クローンペア検出用計算機

タスク管理用計算機から割り当てられたタスクに必要なソースファイルをソースファイル保存用計算機からダウンロードし，3.2 節で述べた手順で演算を行う．関数クローンペアの検出結果をタスク管理用計算機に送信し，次のタスクを受け取る．

4　関数クローンセット

4.1　定義

関数 a_i と関数クローンペアとなる関数の集合を $\{b_1, b_2, b_3, ..., b_n\}$ をする場合，集合 $\{a_i, b_1, b_2, b_3, ..., b_n\}$ を関数 a_i をもとにした関数クローンセットと定義する．この集合は関数 a_i からコピーアンドペーストされた関数の集合と考えることができる．以降，関数 a_i のような関数を基底関数，$b_1, b_2, b_3, ..., b_n$ のような関数を派生関数と呼ぶ．

4.2　生成方法

生成のイメージを図 6 に示す．ノードは関数，エッジは関数クローンペアの関係を表しており，ノードの色の違いによりユニークな関数を表している．この方法で関数クローンセットを生成した場合，関数クローンセット間で基底関数は異なるが，要素となっている関数が全て一致することがある．例えば，関数 a_i が基底関数となる関数クローンセット $\{a_i, b_1, b_2, b_3, ..., b_n\}$ と関数 b_1 が基底関数となる関数クローンセット $\{a_i, b_1, b_2, b_3, ..., b_n\}$ が存在する場合を考える．この場合，関数 a_i と関数 b_1 が基底関数となる関数クローンセット $\{a_i, b_1, b_2, b_3, ..., b_n\}$ のようにマージする．マージのイメージを図 7 に示す．このようにマージを行うことによって情報量を保ったまま関数クローンセット数を少なくし，機能の類似した関数の把握を容易にする．

4.3　関数クローンセットの意味

関数クローンセットを生成することによって，類似した機能を持った関数集合を得ることができる．関数クローンペアの閾値でフィルタリングすることによってどの程度以上類似した関数群を生成するか調節することもできる．これによって，コピーアンドペーストされた後にコードに変更があったとしても違いを吸収して検出することが可能である．

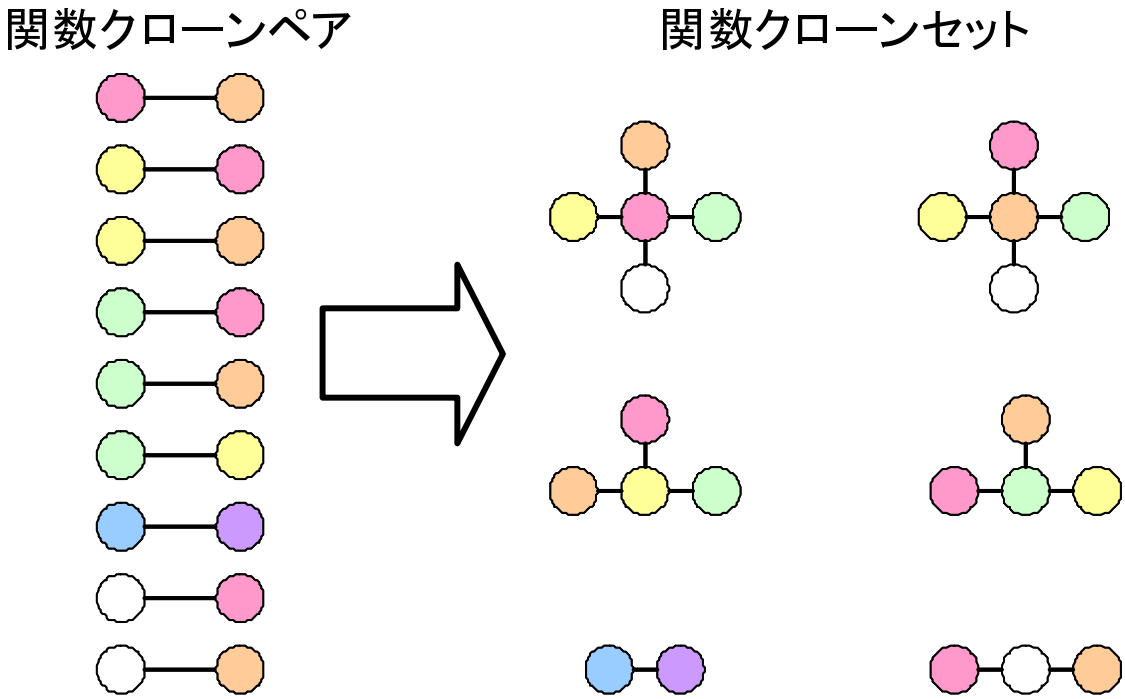

図 6: 関数クローンセットの生成方法

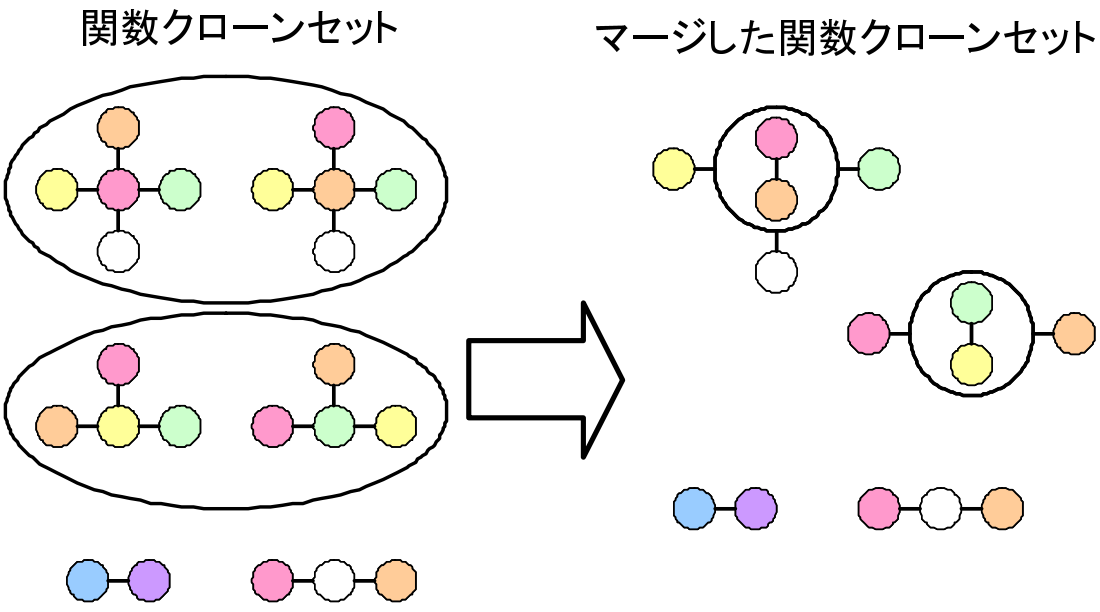

図 7: 関数クローンセットのマージ

5　関数クローンセットメトリクス

1 つの関数クローンセットはある 1 つの機能と考えることができる．その機能がどのような特徴を持っているのかメトリクスを用いて数値化する．本メトリクスは肥後ら [22][23] の手法を参考にした．以下に利用したメトリクスとその定義，意図を述べる．

POP (Population)

関数クローンセットの要素数を表す．ある関数クローンセットを同様の機能を持った関数の集合であると考えると，POP はその関数がどの程度利用されているのかを表す．リファクタリング [24] やライブラリ化を行う場合，POP の高い関数クローンセットを選択することによってより高い効果が得られると考えられる．

LEN (Length)

関数クローンセットに含まれる関数の平均行数を表す．LEN の値が大きいほどより多くの行数を要する機能であるといえる．例えば，POP と合わせて利用することで規模面での削減効果が高い関数を選択することができる．

FOF (Frequency Of Functions)

関数クローンセットに含まれる関数がいくつの異なるソフトウェアに存在するか表す．この値が高いほど複数のソフトウェアで利用されている機能であるといえる．例えば，ライブラリ作成を目的として多くのソフトウェアで利用されている関数を抽出したい場合，FOF 値の高いものを選択することによって，より汎用性の高い関数を得ることができる．

DOS (Dispersion Of Software)

関数クローンセットに含まれる関数がどの程度異なるソフトウェアに分散しているか表す．定義式を以下に示す．

$$DOS = \frac{FOF - 1}{POP - 1} \tag{4}$$

同じソフトウェアで何度も利用されている関数の場合は値が低く，あまり利用されていない場合は値が高くなる．例えば，多くのソフトウェアで利用されているが同じソフトウェア内での利用が少ない関数を得たい場合，FOF，DOS 共に高いものを選択する．

GVOF (General Versatility Of Functions)

関数クローンセットに含まれる関数がいくつの異なるドメインに存在するか表す．ドメインとはソフトウェアの種類を表す．例えば FreeBSD 用ソフトウェア集合 ports には dns, editors，X11 のようにソフトウェアの種類を分けるドメインが存在する．ドメインに関わらず汎用的な機能を持った関数を得たい場合，GVOF の高いものを選択する．

DOD (Dispersion Of Domains)

関数クローンセットに含まれる関数がどの程度異なるドメインに分散しているか表す．

$$DOD = \frac{GVOF - 1}{POP - 1} \tag{5}$$

同じドメインで何度も利用されている関数の場合は値が低く，あまり利用されていない場合は値が高くなる．例えば，あるドメインで集中的に利用されている関数を得たい場合，DOD の低いものを選択する．

6 実験

6.1 目的

提案手法の有効性を確かめるため，実際のソースファイル群への適用実験を行った．実験の目的を下記に示す．

- 関数クローンと関数クローンペアの閾値，完全一致・名前変更検出との関係を確認
- 関数クローンセットメトリクスと関数クローンセットの関係を確認

6.2 実験環境

3.2.2 節で述べた関数クローンペアを検出する並列計算の環境を下記に示す．

タスク管理用計算機

CPU : Intel(R) Xeon(R) CPU 5150 2.66GHz
メモリ : 8.00GB
OS : Windows Vista Business

ソースファイル保存用計算機

CPU : Dual-Core AMD Opteron(tm) Processor 290 2.8GHz
メモリ : 16GB
OS : Solaris10

関数クローンペア検出用計算機 (13 台)

CPU : Dual-Core AMD Opteron(tm) Processor 2210 HE 1.80GHz
メモリ : 6.00GB
OS : Windows Vista Business

関数クローンセットの生成，関数クローンセットメトリクスの計算は関数クローンペアを全て収集した後，タスク管理用計算機で行う．

6.3 実験対象

本稿でのコードクローン検出対象は，オープンソースオペレーティングシステム FreeBSD 用のソフトウェア集合 Ports システムに含まれているソースファイルであり，各ソースファイルは 1 つのプロジェクトに属している．全てのプロジェクトは，bind9，emacs，apache など，一意に特定可能な名前を持っている．共通の特徴を持ったプロジェクトは同じドメイ

表 1: 実験対象

ドメイン名	プロジェクト数	総行数
accessibility	23	2,033,328
archivers	182	3,175,704
benchmarks	62	537,812
converters	88	691,430
dns	134	3,126,332
finance	82	1,359,475
ftp	115	2,060,621
irc	140	4,129,587
mail	717	27,076,351
misc	613	3,351,933

表 2: 実験対象規模

ケース	ドメイン数	プロジェクト数	総行数
1,2	1	134	3,126,332
3,4	1	717	27,076,351
5,6	8	1,306	18,339,890

ンに所属している．例えば，emacs や vim，gedit などは editors ドメインに所属している．また，本研究では検出対象を C 言語で記述されたソースコードに限定しているが，Java などの CCFinder，Ctags が扱えるプログラミング言語であれば，同様に適用可能である．

表 1 に実験対象ドメイン，表 2 に各ケースの実験対象規模，表 3 に各ケースで得られた関数クローンペア数，表 4 に各ケースで得られた関数クローンセット数を示す．次節以降で各ケースについて述べる．

6.4 ケース 1：dns ドメイン，完全一致検出

1 つ目のケースでは，dns ドメインを実験対象とし，完全一致検出で関数クローンの検出を行う．プロジェクト数は 134，総行数は 3,126,332 行の規模である．関数クローンペアの検出時間は約 30 分であった．この実験の目的は，FOF，DOS 値と関数クローンとの関係を調査することである．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア

表 3: 関数クローンペア数

ケース	THRESHOLD(%)				
	60	70	80	90	100
1	96,894	86,744	83,286	80,417	78,037
2	1,004,612	355,017	274,729	262,796	257,207
3	475,022	446,592	421,502	403,447	383,091
4	1,064,941	929,141	819,848	763,538	718,048
5	88,913	80,417	72,405	65,927	56,832
6	407,860	335,802	259,790	221,538	201,565

表 4: 関数クローンセット数

ケース	THRESHOLD(%)				
	60	70	80	90	100
1	7,779	7,750	7,692	7,599	7,529
2	7,350	7,217	7,174	7,033	6,896
3	70,796	69,910	69,024	67,941	67,064
4	72,196	70,971	69,313	67,881	66,455
5	36,980	34,798	32,717	30,468	28,016
6	49,290	45,966	41,818	38,941	36,378

閾値が上昇するにつれて減少している．関数クローンペア閾値が高くなるほどより多くの行がコードクローンになっていなければならないので，関数クローンペア数が減ると考えられる．結果はこの予測に従っているといえる．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP 値と関数クローンセット数の関係は，反比例ような関係になっていた．POP 値が 6 の関数クローンセットが最も多く，続いて POP 値 2，5 のように POP 値が非常に小さい関数クローンセットがほとんどであった．また，POP 値が上位の関数クローンセット数は非常に少なかった．

LEN 値は，10〜50 付近の関数クローンセットが大部分を占め，LEN 値が 10〜50 付近を離れる程，関数クローンセットは少なくなっていた．FOF 値，DOS 値，関数クローンペア閾値 100 %の関数クローンセット数の関係を表 5 に示す．表に示す通り，DOS 値 0.9〜1.0，FOF 値 2 以上に関数クローンセットが集中していることがわかる．これは，関数単位のクローンはプロジェクト内よりもプロジェクト間で多いことを表す．最も関数クローンセット数の多い，DOS 値 0.9〜1.0，FOF 値 6 の関数クローンセットの要素を調べたところ，ほぼ全て bind94，bind95，bind96，bind9，bind9-sdb-postgresql，bind9-sdb-ldap プロジェクト間の関数クローンセットであった．さらに，これらのクローンはほぼ全てソースファイル単位のクローンであり，バージョン間で修正されずに利用されているソースファイルであった．また，関数クローンペアの閾値を下げた場合においても，関数クローンセット数の増加が小幅であるため，傾向の変化は無かった．

表 5: ケース 1:関数クローンペア閾値 100 %の関数クローンセット数の分布

DOS / FOF	0.0 〜0.1	0.1 〜0.2	0.2 〜0.3	0.3 〜0.4	0.4 〜0.5	0.5 〜0.6	0.6 〜0.7	0.7 〜0.8	0.8 〜0.9	0.9 〜1.0
1	58	0	0	0	0	0	0	0	0	0
2	0	0	0	7	6	0	0	0	0	1,852
3	0	0	0	0	1	0	7	0	0	496
4	0	0	0	0	0	0	0	16	0	279
5	0	2	5	0	7	0	0	21	0	1,379
6	0	1	3	2	30	1	0	3	44	2,795
7	0	0	0	0	54	0	0	0	7	449
8	0	0	0	0	0	0	0	0	0	4
10	0	0	0	0	0	0	0	0	0	0

6.5 ケース 2：dns ドメイン，名前変更検出

2 つ目のケースでは，名前変更検出でその他はケース 1 と同条件で行う．この実験の目的は，完全一致検出と名前変更検出の結果がどの程度ことなるか確認することである．関数クローンペアの検出時間は約 30 分であった．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア閾値が上昇するにつれて減少しており，ケース 1 と同様であった．しかし，ケース 1 とは異なり, 関数クローンペア数は閾値 60 %と 70 %の間に大きな差がある．これは，識別子を変更した関数は 30 %程度コードの変更があるものが多いことを表す．また，3.2.1 節で述べた通り，完全一致検出より名前変更検出のほうがより多くのコードクローンが検出されるので，ケース 2（名前変更検出）のほうがケース 1（完全一致検出）より多くの関数クローンが検出されると予想できる．実験の結果，予想通りケース 2 のほうがより多くの関数クローンペアが検出された．しかし，関数クローンセット数はケース 2 に比べケース 1 のほうが多くなっている．これはケース 1 では識別子が異なっていたために別の関数クローンセットになっていたものが字句をパラメータ化することによって 1 つの関数クローンセットになったことが原因である．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP，LEN 値と関数クローンセット数との関係はケース 1 と同様であった．FOF 値，DOS 値，関数クローンペア閾値 100 %の関数クローンセット数の関係を表 6 に示す．大まかな傾向はケース 1 と同じであるが，若干の違いが見られる．ケース 1 に比べ，DOS 値 0.9〜1.0, FOF 値 2 以上の関数クローンセット数が減少し，DOS 値 0.9 未満の関数クローンセット数の増加が確認できる．これは，名前変更検出により関数クローンペアがより多く検出され, 関数クローンセットの POP 値が増加し，DOS 値が減少したためであると考えられる．

6.6 ケース 3：mail ドメイン，完全一致検出

3 つ目のケースでは，dns ドメインより巨大な mail ドメインを実験対象とし，完全一致検出で関数クローンの検出を行う．プロジェクト数は 717，総行数は 27,076,351 行の規模である．関数クローンペアの検出時間は約 13 時間半であった．この実験の目的は，より大きなドメインを対象とした場合の FOF，DOS 値と関数クローンとの関係を調査することである．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア閾値が上昇するにつれて減少しており，ケース 1，2 と同様であった．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP 値と関数クローンセット数との関係はケース 1，2 と同様であった．LEN 値は，15 付近で関数クローンセット数が最も多く，LEN 値が上がるにつれて反比例するように減少してい

表 6: ケース 2:関数クローンペア閾値 100 %の関数クローンセット数の分布

FOF \ DOS	0.0 ～0.1	0.1 ～0.2	0.2 ～0.3	0.3 ～0.4	0.4 ～0.5	0.5 ～0.6	0.6 ～0.7	0.7 ～0.8	0.8 ～0.9	0.9 ～1.0
1	227	0	0	0	0	0	0	0	0	0
2	7	27	0	43	10	0	0	0	0	1,630
3	0	4	0	8	2	0	26	0	0	399
4	0	3	0	0	6	0	0	17	0	241
5	2	13	29	6	40	0	1	71	0	1,098
6	13	60	35	10	176	1	0	12	80	2,210
7	7	5	7	0	42	0	0	0	21	306
8	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	1

た．FOF 値，DOS 値，関数クローンペア閾値 100 %の関数クローンセット数の関係を表 7 に示す．表に示す通り，DOS 値 0.9～1.0，FOF 値 2～6 に関数クローンセットが集中していることがわかる．最も関数クローンセット数の多い，DOS 値 0.9～1.0，FOF 値 4 の関数クローンセットの要素を調べたところ，大部分が lightning，thunderbird，enigmail-thunderbird，enigmail-seamonkey プロジェクト間，続いて postfix，postfix23，postfix24，postfix-current プロジェクト間，続いて mutt，mutt-lite，mutt-devel，mutt-devel-lite プロジェクト間，続いて bogofilter，bogofilter-qdbm，bogofilter-sqlite，bogofilter-tc プロジェクト間の関数クローンセットであった．さらに，これらのクローンはほぼ全て修正されずにソースファイル単位で利用されていた．また，関数クローンペアの閾値を下げた場合においても，関数クローンセット数の増加が小幅であるため，傾向の変化は無かった．

6.7 ケース 4：mail ドメイン，名前変更検出

4 つ目のケースでは，名前変更検出でその他はケース 3 と同条件で行う．関数クローンペアの検出時間は約 14 時 12 分であった．ケース 3 に比べ検出時間を必要としたのは，完全一致検出より名前変更検出のほうがより多くのコードクローンを検出するためである．この実験の目的は，より大きなドメインを対象とした場合の完全一致検出と名前変更検出の結果がどの程度異なるか調査することである．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア

表 7: ケース 3:関数クローンペア閾値 100 %の関数クローンセット数の分布

FOF \ DOS	0.0 ∼0.1	0.1 ∼0.2	0.2 ∼0.3	0.3 ∼0.4	0.4 ∼0.5	0.5 ∼0.6	0.6 ∼0.7	0.7 ∼0.8	0.8 ∼0.9	0.9 ∼1.0
1	766	0	0	0	0	0	0	0	0	0
2	7	45	21	127	149	0	0	0	0	10,884
3	2	23	21	249	6	0	65	0	0	8,481
4	2	78	133	14	1,092	10	0	353	0	41,759
5	1	15	29	68	84	31	8	197	0	844
6	0	4	3	3	36	1	0	0	31	1,237
7	0	0	0	0	0	0	0	0	12	33
8	0	0	0	0	0	0	0	0	0	32
9	0	0	0	0	0	0	0	4	2	9
10	0	0	0	0	0	0	5	0	0	13
11	0	0	0	0	0	0	0	0	0	16
12	0	0	0	0	0	0	0	0	0	12
13	0	0	0	0	0	0	0	1	0	22
14	0	0	0	0	0	0	0	0	0	6
15	0	0	0	0	0	0	0	0	0	3
16	0	0	0	0	0	0	0	0	0	3
17	0	0	0	0	0	0	0	0	0	3
18	0	0	0	0	0	0	0	0	0	4
20	0	0	0	0	0	0	0	0	0	1
21	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	3
26	0	0	0	0	0	0	0	0	0	1

閾値が上昇するにつれて減少しており，ケース 1，2，3 と同様であった．しかし，ケース 2 とは異なり，大幅に減少する部分は見られなかった．完全一致検出と名前変更検出の違いについては，ケース 1 とケース 2 の関係同様，関数クローンペア数はケース 4 のほうが多くなっている．しかし，関数クローンセット数については，関数クローンペアの閾値が小さいときはケース 4 のほうが多く，大きいときはケース 3 のほうが多くなっている．これは，関数クローンペアの閾値が小さいときは，名前変更検出の効果も合わさり，より多くのパターンの関数クローンセットが存在し，閾値が大きくなるにつれて，コードが変更された関数が閾値を満たさず，関数クローン数が減ったためである．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP，LEN 値と関数クローンセット数との関係はケース 1，2，3 と同様であった．FOF 値，DOS 値，関数クローンペア閾値 100 %の関数クローンセット数の関係を表 8 に示す．大まかな傾向はケース 3 と同じであるが，若干の違いが見られる．ケース 3 に比べ，DOS 値 0.9〜1.0，FOF 値 3〜6 付近の関数クローンセット数が減少し，DOS 値 0.9 未満の関数クローンセット数の増加が確認できる．これは，名前変更検出により関数クローンペアがより多く検出され，関数クローンセットの POP 値が増加し，DOS 値が減少したためであると考えられる．

6.8　ケース 5：複数ドメイン，完全一致検出

5 つ目のケースでは表 1 に示す dns，mail 以外の 8 ドメインを実験対象とし，完全一致検出で関数クローンの検出を行う．プロジェクト数は 1,306，総行数は 18,339,890 行の規模である．関数クローンペアの検出時間は約 6 時間半であった．この実験の目的は，GVOF，DOD 値と関数クローンとの関係を調査することである．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア閾値が上昇するにつれて減少しており，ケース 1，2，3，4 と同様であった．しかし，関数クローンセット数に関して，同ドメイン内の関数クローンセットの場合に比べ，減少する割合が大きいことがわかる．これは，ドメイン間の関数クローンのほうがドメイン内の関数クローンに比べ，コードの変更が多いことを表している．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP 値と関数クローンセット数との関係はケース 1，2，3，4 と同様であった．LEN 値は 10〜40 付近で最も関数クローンセット数が多く，LEN 値が上がるにつれ関数クローンセット数が減少していた．GVOF 値は 1 で最も関数クローンセット数が多く，2 になると急激に関数クローンセット数が減少し，GVOF 値が上がるにつれ関数クローンセット数が減少していた．DOD 値は 0 の関数クローンセットがほとんどでその他の DOD 値の関数クローンセットは非常に少なかった．関数クローンペア閾値 100 %の関数クローンセット 28,016 個

表 8: ケース 4:関数クローンペア閾値 100 %の関数クローンセット数の分布

FOF \ DOS	0.0 ∼0.1	0.1 ∼0.2	0.2 ∼0.3	0.3 ∼0.4	0.4 ∼0.5	0.5 ∼0.6	0.6 ∼0.7	0.7 ∼0.8	0.8 ∼0.9	0.9 ∼1.0
1	3,405	0	0	0	0	0	0	0	0	0
2	40	108	36	307	252	0	0	0	0	9,942
3	38	57	66	411	34	0	148	0	0	7,643
4	164	510	537	263	2,661	19	0	920	0	35,891
5	7	91	58	90	88	40	11	199	0	812
6	1	16	22	17	36	3	2	0	15	53
7	0	13	4	1	8	3	2	0	15	53
8	0	4	5	0	4	0	0	1	1	30
9	0	1	1	0	0	0	1	4	5	17
10	0	0	0	0	0	0	5	0	2	12
11	0	0	1	0	0	0	1	0	0	16
12	0	0	0	0	0	0	0	1	0	13
13	0	0	0	0	0	1	1	5	1	20
14	0	0	1	0	0	0	2	7	1	3
15	0	0	0	0	1	0	0	0	0	1
16	0	0	0	0	0	0	1	0	0	3
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	2
20	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	3
23	0	0	0	0	0	0	0	0	0	2
24	0	0	0	0	0	0	0	0	0	3
26	0	0	0	0	0	0	0	0	0	1

を GVOF 値 2 以上でフィルタリングすると 2,863 個の関数クローンセットが得られた．これは，ドメイン間の関数クローンが非常に少ないことを表している．

6.9　ケース 6：複数ドメイン，名前変更検出

6 つ目のケースでは名前変更検出でその他はケース 5 と同条件で行う．関数クローンペアの検出時間は約 14 時間 20 分であった．ケース 5 に比べ検出時間を必要としたのは，完全一致検出より名前変更検出のほうがより多くのコードクローンを検出するためである．この実験の目的は，完全一致検出と名前変更検出の結果がどの程度異なるか調査することである．

表 3，4 に示す通り，関数クローンペア数，関数クローンセット数共に関数クローンペア閾値が上昇するにつれて減少しており，ケース 1，2，3，4，5 と同様であった．完全一致検出と名前変更検出の違いについては，関数クローンペア数は名前変更検出のほうが多く，同ドメイン内の関数クローンの場合と同様であったが，関数クローンセット数は同ドメイン内とは異なり，常に名前変更検出のほうが多かった．これは，ドメイン間の関数クローンはドメイン内の関数クローンに比べて様々なものが存在し，名前変更検出でより多くの関数クローンペアを検出したが，類似した関数が少なかったためだと考えられる．

次に関数クローンセットメトリクスと関数クローンセット数の関係の結果について述べる．POP 値と関数クローンセット数との関係はケース 1，2，3，4，5 と同様であった．LEN 値はケース 5 とは異なり，LEN 値 4 で最も関数クローンセット数が多くなっており，名前変更検出では，4 行程度の非常に小さな関数クローンが多く検出されていた．GVOF，DOD の傾向はケース 5 と同様であった．

表 9: 関数クローンセットに含まれるユニークな関数の総数

ケース	THRESHOLD(%)				
	60	70	80	90	100
1	35,593	35,309	35,048	34,707	34,310
2	36,688	36,254	35,919	35,457	34,984
3	255,050	252,275	249,466	246,275	242,868
4	269,463	265,787	261,757	257,541	253,455
5	83,070	78,311	73,912	69,443	64,122
6	113,387	107,011	99,980	94,238	88,918

7 考察

7.1 ケース 1，2

ケース 1，2 では dns ドメインを対象に実験を行った．全ての関数クローンセットに含まれる要素でユニークな関数の総数を表 9 に示す．関数クローンペア閾値が増加するにつれて関数クローンペア数が減少しているため，関数クローンとなる関数も減少している．また，名前変更検出のほうがより多くの関数が関数クローンとなっている．これは，コピーアンドペーストされた後にコードに変更が加えられた関数も検出しているためである．

ここで，完全一致検出，名前変更検出に関わらず，関数クローンペアの閾値を上げていくにつれて減少する関数クローンセット数，要素となっている関数の数は小幅なものであることがわかる．このことから，閾値 60 %以上の関数クローンのほとんどが閾値 100 %の関数クローンであることがわかる．dns ドメインには 60,263 個の関数が存在するので，名前変更検出で関数クローンペア閾値 100 %の関数クローンセット群は，60,263 個の関数から 34,984 個の関数を選択，6,896 種類に分類したものと考えることができる．次にこの関数クローンセット群から再利用性の高い関数を抽出することを目指す．

ここでは再利用性の高い関数を多くのプロジェクトで利用されている関数であるとする．実験の結果から，DOD 上位の関数クローンセットは完全一致検出より名前変更検出のほうが少なかった．これは名前変更検出のほうが多くの関数クローンが得られるため，POP 値が大きくなってしまうことが原因であった．しかし，FOF 値に関しては増加することはあっても，減少することはない．よって，FOF 値を基準とし，3 以上も関数クローンセットを選択する．また，完全一致検出に比べ名前変更検出のほうが識別子の変更に対応することができ，類似した関数をまとめるのに効果的である．しかし，誤検出が増加するので，関数クローンペア閾値が 100 %の関数クローンセット群から選択を行う．表 10 に関数クローンペア閾値

表 10: ケース 2:FOF 値の閾値別関数クローンセット数

FOF 閾値	3	4	5	6	7	8
関数クローンセット数	4,952	4,513	4,246	2,986	389	1

表 11: ケース 2:FOF 値の閾値別関数の総数

FOF 閾値	3	4	5	6	7	8
関数の数	30,860	29,559	28,410	21,746	3,775	10

100 %の関数クローンセット群から FOF 値別にフィルタリングした関数クローンセット数, 表 11 にユニークな関数の総数を示す．例えば，FOF 値 7 以上でフィルタリングした場合, 60,263 個の関数から 384 種類 3,775 個の関数を得ることができる．得られた関数クローンの例を図 8 に示す．この関数は src を dst にコピーする機能を持つ．このように FOF，DOS を利用することによって，ドメイン内の再利用性の高い関数を得ることができた．

7.2 ケース 3，4

ケース 3，4 では mail ドメインを対象に実験を行った．表 9 に示す通り，関数クローンペアの閾値が増加するにつれて関数クローンペアの数が減少しているため，関数クローンとなる関数も減少している．また，コピーアンドペーストされた後に識別子が変更された関数も検出しているため，名前変更検出のほうがより多くの関数が関数クローンとなっている．さらに，閾値 60 %以上の関数クローンのほとんどが閾値 100 %の関数となっている．ここでもケース 1，2 の考察と同様に，名前変更検出で関数クローンペアの閾値が 100 の関数クローンセット群から再利用性の高い関数の取得を行う．

関数クローンペアの閾値 100 %の関数クローンセット群から FOF 値を用いてフィルタリングを行う．表 12 に関数クローンペアの閾値 100 %の関数クローンセット群から FOF 値別にフィルタリングした関数クローンセット数，表 13 にユニークな関数の総数を示す．例えば，FOF 値 8 以上でフィルタリングした場合，617,713 個の関数から 184 種類 1,909 個の関数を得ることができる．得られた関数クローンの例を図 9 に示す．この関数は文字列の指定した位置のポインタを返す機能を持つ．ケース 1，2 の考察同様，FOF，DOS を用いてドメイン内の再利用性の高い関数を得ることができた．

```
File:bind9/work/bind-9.3.6-P1/lib/isc/string.c

size_t
isc_string_strlcpy(char *dst, const char *src, size_t size)
{
    char *d = dst;
    const char *s = src;
    size_t n = size;

    /* Copy as many bytes as will fit *;/
    if (n != 0U && --n != 0U) {
        do {
            if ((*d++ = *s++) == 0)
                break;
            } while (--n != 0U);
        }

    /* Not enough room in dst, add NUL and traverse rest of src */
    if (n == 0U) {
        if (size != 0U)
            *d = '¥0';      /* NUL-terminate dst */
        while (*s++)
            ;
    }

    return(s – src – 1);  /* count does not include NUL */
}
```

図 8: ケース 2:フィルタリング後の関数クローン例

表 12: ケース 4:FOF 値の閾値別関数クローンセット数

FOF 閾値	3	4	5	6	7	8
関数クローンセット数	52,365	43,968	3,003	1,607	283	184

表 13: ケース 4:FOF 値の閾値別関数の総数

FOF 閾値	3	4	5	6	7	8
関数の数	222,635	197,077	17,812	10,591	2,520	1,909

```
File:hashcash/workhashcash-1.22/getopt.c

static char *
my_index (str, chr)
     const char *str;
     int chr;
{
     while (* str)
          {
               if (*str == chr)
                    return (char *) str;
               str++;
           }
     return 0;
}
```

図 9: ケース 4:フィルタリング後の関数クローン例

7.3 ケース 5，6

ケース 5，6 は複数のドメインを対象に実験を行った．表 9 に示す通り，関数クローンペアの閾値が増加するにつれて関数クローンペアの数が減少しているため，関数クローンとなる関数も減少している．しかし，ケース 1，2，3，4 とは異なり，関数の減少割合が大きいことがわかる．これは，ドメイン間の関数クローンはコードが変更されているものが多く，関数クローンペア閾値の影響が大きかったためだと考えられる．完全一致検出と名前変更検出の差が大きいこともこれが理由であると考えられる．以上のことから，ドメイン間の関数クローンは完全一致クローンのほうがより安全に類似した関数を検出できると考える．次に完全一致検出で関数クローンペア閾値 100 %の関数クローンセット群から再利用性の高い関数を抽出することを目指す．

ここでは再利用性の高い関数を多くのドメイン・プロジェクトで利用されている関数であるとする．実験の結果から GVOF を利用することにより再利用性の高い関数を得ることができると考えられる．名前変更検出で関数クローンペア閾値 100 %の関数クローンセット 28,016 個から GVOF が 2 以上のものを選択すると 2,863 個の関数クローンセットが得られた．この関数クローンセット群を FOF 値でフィルタリングすることにより再利用性の高い関数を得る．表 14 に FOF 別にフィルタリングした関数クローンセット数，表 15 にユニークな関数の数を示す．例えば，FOF 値 8 以上でフィルタリングした場合，421,008 個の関数から 77 種類 726 個の関数を得ることができる．得られた関数クローンの例を図 10 に示す．この関数はメモリ領域を解放する機能を持つ．このようにドメイン間の再利用性の高い関数は GVOF，FOF を利用することによって得ることができた．

表 14: ケース 5:FOF 値の閾値別関数クローンセット数

FOF 閾値	3	4	5	6	7	8
関数クローンセット数	1,862	423	304	206	141	77

表 15: ケース 5:FOF 値の閾値別関数の総数

FOF 閾値	3	4	5	6	7	8
関数の数	6,591	2,422	2,007	1,499	1,162	726

File:converter/recode/work/recode-3.6/lib/gettext.c

```
static void __attribute__ ((unused))
free_mem (void)
{
    if (string_space != NULL)
        free(string_space);
    if (map != NULL)
        free(map);
}
```

図 10: ケース 5:フィルタリング後の関数クローン例

7.4 妥当性への脅威

本手法ではクローン検出ツールに CCFinder を用いている．2 章で述べたようにクローン検出ツールには，字句や行単位のものだけでなく，抽象構文木や PDG を用いたものなど様々ある．それぞれのクローン検出ツールによってコードクローンの定義が異なり，本手法を他のツールを用いて行った場合，異なる結果が得られるであろう．

本実験では対象にオープンソースソフトウェアを用いた．一般に，オープンソースソフトウェアのクローン含有率は 5〜20 %，商用ソフトウェアのクローン含有率は約 50 %といわれており，適用するソフトウェアによって検出結果が異なる可能性がある．

8 関連研究

Al-Ekram らはオープンソースソフトウェア間のコードクローンはコピーアンドペーストにより生じたものか，あるいは偶然生じたものであるか調査を行った [25]．テキストエディタ，ウィンドウマネージャの 2 つのドメインのオープンソースソフトウェアに対して，同ドメインのソフトウェア間でコードクローンを検出し，クローンペア，クローンセットの要素を確認して調査を行った．調査の結果，多くのクローンは API やライブラリの利用によるイディオムであったが，ドメイン特有の問題を解決するために再利用されているコードも見つかった．Al-Ekram らはオープンソースソフトウェアはソフトウェアシステムとその問題領域に関する知識、経験、解決策の共有の一躍を担っていると結論付けている．

しかし，同様の目的を持ったオープンソースソフトウェアは複数種類存在するので，オープンソースソフトウェアを利用する際，選択の迷いや複数利用する際の手間が生じる．そこで，複数のオープンソースソフトウェア間で本手法を適用し，関数クローンセットを生成することによって，対象のソフトウェア群の問題領域で頻繁に利用されている解決策をまとめることができる．つまり，オープンソースソフトウェアを用いた知識の共有をよりスムーズに行うことに利用できるのではないかと考える．

Al-Ekram らはコード片単位でコードクローンを検出しているのに対し，本手法は関数単位でコードクローンを検出している．そのため，本手法で関数クローンペア閾値を 100 %にして検出したクローンは Al-Ekram らの検出結果の部分集合になっている．本手法における閾値 100 %の検出では短いコードクローンの組合せからある機能を実現した関数を得ることはできない．このような関数を検出できるように関数クローンペアの閾値を調節するのだが，十分に検出できていない，あるいは誤検出を多く含んでいる可能性があり，今後の課題となる．

Livieri らは超大規模ソースコード集合からコードクローンを検出する D-CCFinder を提案した [19][20]．大量のソースコードを複数のユニットに分割し，それらを組合わせたピースを複数の計算機にタスクとして渡し，並列に計算する方法である．約 4 億行のソースコードから 2 日余りでクローンペアを検出することができた．Livieri らは大量のソースコードからクローンペアの検出を行ったが，本手法では関数単位でクローンペアの検出，さらにクローンセットの生成も行い，内容を発展させたといえる．しかし，関数単位でクローンを検出しているため，Livieri らに比べ計算速度は劣る．

Livieri らは D-CCFinder を用いて Linux カーネルの進化の調査も行っている [26][27]．Linux カーネルの各バージョンのペアにつき，類似度を算出した．80 台のワークステーションで実行したところ，コードクローンの検出に約 6 時間，類似度の算出に約 50 分要している．実験結果は近いバージョン間では類似度が高く，メジャーアップグレードを跨いでいる

場合はマイナーアップグレード間に比べて，類似度が低く，多くの変更が施されていることが確認された．

Livieri らが同プロジェクト異バージョン間でコードクローンを検出・ソフトウェアの進化の様子を調査したのに対して，本研究では複数のプロジェクトを対象にコードクローンを検出，ドメイン間，プロジェクト間での関数の再利用の様子の調査，さらに関数の機能の分類を行ったといえる．

Liu らはソフトウェア間で PDG[28] の類似したソースコードを発見することにより，ソースコードの盗用を検出する手法を提案している [29]．PDG はフォーマット・識別子・文の順序変更に対して変化がなく，制御文変更やコードの挿入などの変化にも柔軟に対応できるので，盗用された後に偽装されたコードも発見することができる．Liu らの手法はソフトウェア間で PDG を用いたコードクローンを検出しているといえ，再利用コードを発見することが可能である．本手法は字句単位のコードクローンを検出する CCFinder を用いているが，Liu らの手法は PDG を用いているため，よりコードの変更に対応した検出が可能であるではないかと考える．

9 あとがき

本研究では，大量のソフトウェアから同様の機能を実装した関数集合群を検出する手法を提案した．まず，対象のソースコード群を複数のタスクに分割し，複数のマシンで並列に関数クローンペアを検出する．そして，その結果を一台の計算機に収集して，ある関数と関数クローンペアとなる集合を関数クローンセットとして検出した．さらに，関数クローンセットメトリクスを用いたフィルタリングを行った．

本研究では，クローン検出ツールに CCFinder を用いた．CCFinder はスケーラビリティ，再現率が高いが，誤検出が多く，適合率が低いというデメリットがある．計算の工夫をすることにより，他のツールを用いても高速で，適合率の高い関数クローンを検出できる可能性がある．クローン検出ツール別に結果を比較することで興味深い結果が得られるであろう．

今回は並列計算の方法として Java RMI[30] を利用したが，MapReduce[31] のような分散並列処理を行うことによってより高いパフォーマンスが得られる可能性がある．また，GUI による支援ツールや新たな関数クローンセットメトリクスを導入することで，より効率的に利用者のニーズにあった関数クローンが抽出できると考える．

謝辞

本研究の全過程を通して，理解ある御指導を賜り，的確な御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 教授に深く感謝致します.

本研究において，有益且つ的確な御助言を頂きました 同 岡野 浩三 准教授に深く感謝致します.

本研究において，常に適切な御指導および御助言を頂きました 同 肥後 芳樹 助教に深く感謝致します.

本研究において，多大なる御助言を頂きました 同 柿元 健 特任助教に深く感謝致します.

最後に，その他様々な御指導，御助言を頂いた大阪大学 大学院情報科学研究科コンピュータサイエンス専攻 楠本研究室の皆様に深く感謝致します.

参考文献

[1] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会 D, vol.91-D, no.6, pp.1465-1481, 2008.

[2] T.Kamiya, S.Kusumoto, and K.Inoue. CCFinder : A multi-linguistic token-based code clone detection system for largescale source code. IEEE Transactions on Software Engineering, Vol. 28, No. 7, pp. 654?670, 2002.

[3] CCFinderX. http://www.ccfinder.net/ccfinderx-j.html.

[4] D.Gusfield. Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997.

[5] L.Jiang, G.Misherghi, Z.Su, and S.Glondu. DECKARD : Scalable and Accurate Tree-based Detection of Code Clones. In Proc. of the 29th International Conference on Software Engineering, pp.96-105, 2007.

[6] M.Datar, N.Immorlica, P.Indyk, and V.S.Mirrokni. Locality-Sensitive Hashing Scheme based on P-Stable Distributions. In Proc. of the 20th Symposium on Computational Geometry, pp.253-262, 2004.

[7] R.Wettel and R.Marinescu. Archeology of Code Duplication : Recovering Duplication Chanins from Small Duplication Fragments. In Proc. of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp.63-70, 2005.

[8] Z.Li, S.Lu, S.Myagmar, and Y.Zhou. CP-Miner : Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering, Vol.32, No.3, pp.176?192, 2006.

[9] X.Yan, J.Han, and R.Afshar. Clospan : Mining Closed Sequential Patterns in Large Datasets. In Proc. of 2003 SIAM International Conference on Data Mining, pp.166-177, 2003.

[10] I.B.Baxter, A.Yahin, L.Moura, M.Sant'Anna, and L.Bier. Clone Detection Using Abstract Syntax Tree. In Proc. of the International Conference on Software Maintenance, pp.368-377, 1998.

[11] B.S.Baker. Finding Clones with Dup : Analysis of an Enperiment. IEEE Transactions on Software Engineering, Vol.33, No.9, pp.608-621, 2007.

[12] B.S.Baker. A Program for Identifying Duplicated Code. In Proc. of the 24th Symposium of Computing Science and Statistics, Vol.6, pp.49-57, 1992.

[13] B.S.Baker. On Finding Dupulication and Near-Duplication in Large Software Systems. In Proc. of the 2nd Working Conference on Reverse Engineering, pp.86-95, 1995.

[14] B.S.Baker. Parameterized Duplication in Strings : Algorithms and an Application to Software Maintenance. SIAM Journal on Computing, Vol.26, No.5, pp.1343-1362, 1997.

[15] S.Ducasse, M.Rieger, and S.Demeyer. A Language Independent Approach for Detecting Duplicated Code. In Proc. of the International Conference on Software Maintenance, pp.109-118, 1999.

[16] J.Krinke. Identifying Similar Code with Program Dependence Graphs. In Proc. the 8th Working Conference on Reverse Engineering, pp.301-309, 2001.

[17] J.Mayrand and C.Leblanc and E.M.Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proc. of the 1996 International Conference on Software Maintenance, pp.244-254, 1996.

[18] E.Burd and J.Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, pp.36-43, 2002.

[19] S.Livieri, Y.Higo, M.Matushita, and K.Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In Proc. of the 29th International Conference on Software Engineering, pp.106-115, 2007.

[20] リビエリシモネ, 肥後芳樹, 松下誠, 井上克郎. D-CCFinder: 超大規模ソースコード集合を対象とした分散処理型コードクローン検出・可視化システム. 電子情報通信学会技術研究報告, SS2006-68, Vol.106, No.427 pp.19-25, Dec 2006.

[21] Ctags. http://ctags.sourceforge.net/.

[22] Y.Higo, T.Kamiya, S.Kusumoto, and K.Inoue. Aries:Refactoring support environment based on code clone analysis. In The 8th IASTED International Conference on Software Engineering and Applications(SEA 2004), pp.222-229, 2004.

[23] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローンを対象としたリファクタリング支援環境. 電子情報通信学会論文誌 D-1, Vol.J88-D-I, No.2, pp.186-195, 2005.

[24] M.Fowler. Refactoring: improving the design of existing code. Addison Wesley, 1999. (児玉公信, 友野晶夫, 平澤章, 梅澤真史　訳 (2000)『リファクタリング　プログラミングの体質改善テクニック』. ピアソン・エデュケーション).

[25] R.Al-Ekram, C.Kapser, R.Holt, and M.Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In International Symposium on Empirical Software Engineering, 2005.

[26] S.Livieri, Y.Higo, M.Matsushita, and K.Inoue, Analysis of the Linux Kernel Evolution Using Code Clone Coverage. Proc. of the 4th Workshop on Mining Software Repositories, pp.22.1-22.4, 2007.

[27] リビエリ シモネ, 肥後 芳樹, 松下 誠, 井上 克郎. コードクローン検出技術を用いたLinux カーネル進化の調査. 電子情報通信学会論文誌 D-I, Vol.J91-D, No.2, pp.509-511, 2008.

[28] ELLCEY, S.J. The program dependence graph: interprocedural information representation and general space requirements. Master's thesis, Dept. of Computer Science, Michigan Technological Univ., Houghton, MI, 1985.

[29] C.Liu, C.Chen, J.Han, and P.Yu. GPLAG: Detection of Software Plagiarism by Program Dependence GraphAnalysis. In Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp.872-881, 2006.

[30] java.rmi (Java Platform SE 6).
http://java.sun.com/javase/ja/6/docs/ja/api/java/rmi/package-summary.html

[31] J.Dean and S.Ghemawat. MapReduce: Simplified Data Processing on LargeClusters. In Proc. of OSDI ’04: 6th Symposium on Operating System Design and Implemention, San Francisco, CA, 2004