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.

複数のメソッドにまたがって存在するコードクローンの検出に向けて

肥後　芳樹†　　楠本　真二†

† 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　〒 565-0871 吹田市山田丘 1-5
E-mail: †{higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　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いるが，複数のメソッドに分散して存在しているコードクローンを検出できる手法は少ない．本稿では，システム依存グラフ（対象プログラム全体から構築したプログラム依存グラフ）を用いることにより，そのようなコードクローンを検出する手法を提案する．本稿で用いるシステム依存グラフはコードクローン検出に特化しており，従来のシステム依存グラフを用いた場合では検出できないコードクローンを検出することができる．また，提案するシステム依存グラフに対するプログラムスライシング方法も提案する．このスライシングを用いることにより，計算コストを抑えつつより高精度でコードクローンを検出することができる．

キーワード　コードクローン，プログラム依存グラフ

Towards Detecting Code Clones that are Scattered on Multiple Methods

Yoshiki HIGO† and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
E-mail: †{higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　At present, there are various kinds of code clone detection techniques, however, few of them can detect code clone that are scattered on multiple methods. This paper proposes a new system dependency graph, which represents dependencies over an entire system, for detecting such scattered code clones. The proposed graph is specialized for code clone detection. Consequently, the proposed graph realizes more precious code clone detection than the existing graphs. Also, this paper proposes a specialized program slicing on the proposed graph. The proposed slicing reduces the computational cost and improves the accuracy of code clone detection.

Key words　Code Clone，Program Dependency Graph

1. はじめに

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を表す．近年，コードクローンはソフトウェア工学における研究対象の一つとして注目を集めており，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いる [1], [5], [9]．多くの既存手法は，メソッド内や関数内，ファイル内など，モジュール内で閉じたコードクローンを検出する．

しかし，ソフトウェアシステムには多くのモジュールが存在しており，全体として機能を提供しているため，複数のモジュールに分散して存在しているコードクローンも，モジュール内クローンと同様に多く存在すると考えられる．一部の検出手法は，複数のモジュールに分散したコードクローンを検出可能である．それらの手法はまず，各モジュール内でのコードクローンを検出したのち，モジュールにまたがるコードクローンの検出へと処理を進める．そのため，各メソッド内に存在している重複コードはある程度の大きさがないと，コードクローンとして検出されない．

本稿では，既存研究の問題点を解決した，新しいコードクローン検出手法を提案する．提案手法では，対象ソフトウェア全体から，その要素間の依存関係を表すプログラム依存グラフを構築し，そのグラフ上での同形部分グラフをコードクローンとして検出する．

2. 準　　備

2.1 プログラム依存グラフ

プログラム依存グラフ（PDG）とは，プログラム内の要素（文）の間に存在する依存関係を表す有効グラフである．PDG の頂点はプログラムの要素であり，辺で結ばれた頂点に依存関係があることを表す．次に，PDG の二つの依存関係について

説明する．

データ依存　文 s で変数 v を定義し，文 t で変数 v を参照しており，文 s から文 t への経路のうち，変数 v を再定義しないものがある場合，文 s から文 t にデータ依存があるという．

制御依存　文 s が条件文または繰り返し文の条件式であり，文 s の条件判定の結果によって文 t を実行するか否かが直接決まる場合，文 s から文 t への制御依存関係があるという．

以降，本論文では，メソッド単位のプログラム依存グラフをメソッド依存グラフ（MDG），システム全体のプログラム依存グラフをシステム依存グラフ（SDG）と呼ぶ．

2.2　メソッド依存グラフを用いたコードクローン検出

MDG を用いたコードクローン検出の手順 [4], [8] を示す．

STEP1 : MDG の全ての頂点のハッシュ値を求め，ハッシュ値が同じ頂点毎にグループを作成する．ハッシュ値は頂点が表すプログラム要素の構造に基づいて計算される．ハッシュ値を計算する前に，プログラム要素の変換を行うこともある．例えば，利用している変数やリテラルをその型に変換することが考えられる．この処理を行うことにより，利用している変数やリテラルが異なっていても，それらの型が同じであり，そのプログラム要素の構造が等しければ，同じハッシュ値が生成される．

STEP2 : プログラムスライシングを行い，同形部分グラフを検出する．スライス基点は，同じグループに属する頂点のペア (r_1,r_2) であり， 二つのスライシングは同期して行われる．スライシングにより新たにたどった頂点のハッシュ値が等しい場合はそれらを同形部分 グラフの頂点として加える．スライシングが下記条件のいずれかを満たすとき，たどった頂点は同形部分グラフに加えられず，スライシングを終了する．

- 新たにたどった頂点のペア (p_1, p_2) が異なるハッシュ値を持つ場合．
- (p_1, p_2) のハッシュ値は等しいが，r_1 のグラフ（または r_2 のグラフ）がすでに p_1（または p_2）を含んでいる場合（無限ループを回避するための処理）．
- (p_1, p_2) のハッシュ値は等しいが，r_1 のグラフ（または r_2 のグラフ）が p_2（または p_1）を含んでいる場合（二つの同形部分グラフが頂点を共有するの回避するための処理）．

この処理を同じグループに属する全ての頂点のペアに対して行う．スライシング終了後に特定されているグラフのペア（二つの同形部分グラフ）が本手法において検出されるクローンペアである．

STEP3 : あるクローンペア (s_1, s_2) が他のクローンペア (s'_1, s'_2) に含まれている場合（$s_1 \subseteqq s'_1 \cap s_2 \subseteqq s'_2$），そのクローンペアを検出されたクローンペアの集合から削除する．他のクローンペアに包含されたクローンペアをユーザに対して提示する理由はなく，またこれらの存在は検出結果を肥大化させてしまうからである．

STEP4 : 同じグラフを持つクローンペアからクローンセットを形成する．例えば，二つのクローンペア (s_1, s_2), (s_2, s_3) があった場合，クローンセット $\{s_1, s_2, s_3\}$ が形成される．

```
 1: public class Sample {
 2:   void method1() {
 3:     int x = 10;
 4:     int y = 20;
 5:     int z = x + y;
 6:     System.out.println(z);
 7:   }
 8:   void method2() {
 9:     int x = 10;
10:     int y = 20;
11:     int z = x - y;
12:     System.out.println(z);
13:   }
14:   void method3() {
15:     int x = 10;
16:     int y = 20;
17:     Operation o = new Plus();
18:     int z = o.operate(x, y);
19:     System.out.println(z);
20:   }
21:   void method4() {
22:     int x = XXX.getX();
23:     int y = XXX.getY();
24:     Operation o = new Plus();
25:     int z = o.operate(x, y);
26:     System.out.println(z);
27:   }
28: }
29: abstract class Operation {
30:   abstract int operate(int a, int b);
31: }
32: class Plus extends Operation {
33:   int operate(int a, int b) {
34:     int c = a + b;
35:     return c;
36:   }
37: }
38: class Minus extends Operation {
39:   int operate(int a, int b) {
40:     int c = a - b;
41:     return c;
42:   }
44: }
```

図 1　サンプルソースコード

3.　システム依存グラフの構築

本章では，コードクローン検出に特化した SDG の構築法を提案する．SDG の構築は以下の手順からなる．

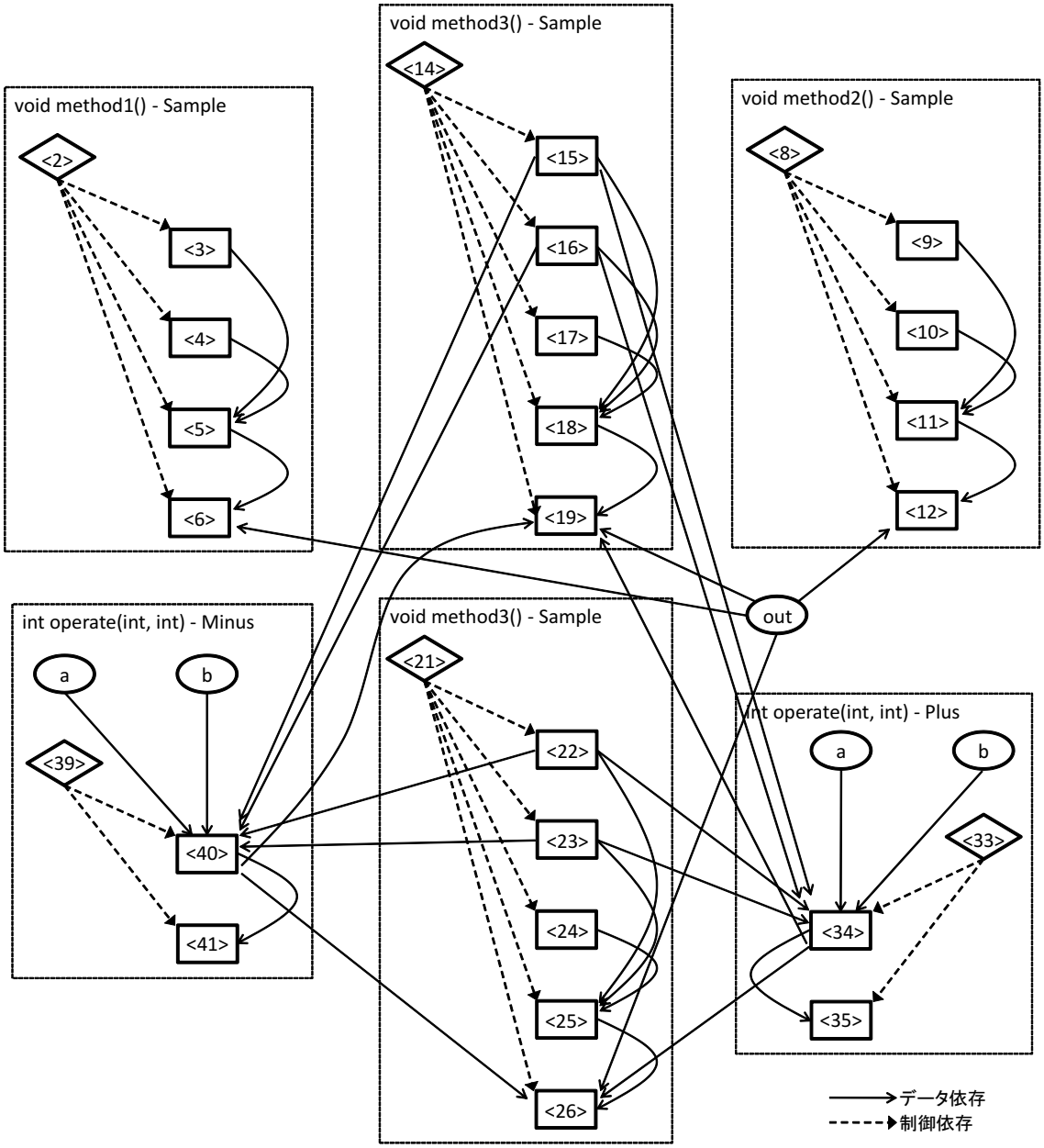

図 2　サンプルソースコードか生成された SDG

手順 1：対象システム内の各メソッドに対して MDG を構築
手順 2：MDG を連結することにより SDG を構築

以降，本節では，各手順について詳細に説明し，その後 SDG 構築に必要な文の分解について述べる．

3.1　手順 1：対象システム内の各メソッドに対して MDG を構築

手順 1 では，各メソッドから MDG を構築する．MDG の構築方法は文献 [3], [6] で提案されているものを用いる．本稿で用いる MDG の特徴としては，下記のものが挙げられる．

- 各 MDG には入り口を表す頂点が存在する．この頂点は便宜上条件式とみなされ，直内に存在している文の頂点に対して制御依存辺が引かれる．
- 仮引数を表す頂点が存在し，仮引数を再定義すること無く参照している文に対してデータ依存辺が引かれる．

図 1 は本稿で用いるソースコードの例，図 2 はそのソースコードから生成される MDG および SDG の例である．図 2 において．各枠線の範囲内が各メソッドの MDG を表している．条件節を表す頂点は菱形，引数などの変数を表す頂点は楕円，その他の文を表す頂点は長方形で表されている．頂点の数字は，その頂点の文が定義されているソースコード上での行番号を表す．なお，仮引数を表す頂点については，行番号ではなく，仮引数の変数名で表している． x や y などの変数の定義・参照を行なっている文の頂点間にデータ依存があるのがわかる．

3.2　手順 2：MDG を連結することにより SDG を構築

手順 1 で構築した各 MDG において，メソッド呼び出しを行なっている文を表す各頂点に関して，呼び出されたメソッドの MDG との間に下記のデータ依存関係を構築する．

- 引数を介したデータ依存関係
- 共有変数を介したデータ依存関係
- 返り値を介したデータ依存関係

これら三つのデータ依存関係は，二つの頂点集合，Src と Des において，特定の条件が満たされる場合に，前者に含まれる各頂点から，後者に含まれる全ての頂点に対してデータ依存辺が引かれる．なお，多態性によりどのメソッドが呼び出されるかが一意に特定できない場合は，呼び出される可能性のあるメソッド全てに対して依存辺が引かれる．

引数を介したデータ依存関係では，二つの頂点集合とデータ依存辺の引くための条件は下記のように定義される．

引数を介したデータ依存関係

Src(C, V_1)：メソッド呼び出し C を持つ頂点に対して，実引数 V_1 についてのデータ依存辺を持つ頂点の集合
Des(M, V_2)：メソッド M の仮引数 V_2 がデータ依存辺を持つ頂点の集合
条件：メソッド M はメソッド呼び出し C において呼び出される可能性があり，実引数 V_1 に対応する仮引数は V_2 である

例えば，図 1 と図 2 の例では，18 行目のメソッド呼び出しに関して，33 行目から始まるメソッドに対しては<15>→<34>と<16>→<34> の二つのデータ依存辺が引かれ，39 行目から始まるメソッドに対しては<15>→<40>と<16>→<40> の二つのデータ依存辺が引かれる．

メソッド呼び出し元で代入処理した共有変数を呼び出したメソッドで参照している場合のデータ依存関係の構築については，二つの集合とデータ依存辺を引くための条件は下記のように定義される．

共有変数を介したデータ依存関係 (1)

Src(C, V_1)：メソッド呼び出し C が実行される直前における共有変数 V_1 の値を決定した頂点の集合（メソッド呼び出し C が実行される前において，最後に変数 V_1 に対して代入を行なった頂点の集合）
Des(M, V_2)：メソッド M において，共有変数 V_2 を再定義すること無く参照している頂点の集合
条件：メソッド M はメソッド呼び出し C において呼び出される可能性があり，共有変数 V_1 と V_2 は同じ変数である

また，メソッド呼び出し先で代入処理した共有変数を呼び出し元で参照している場合のデータ依存関係の構築では，二つの集合は次のように定義される．

共有変数を介したデータ依存関係 (2)

Src(M, V_1)：メソッド M において，共有変数 V_1 に対して代入処理を行なっている頂点のうち，そのメソッド内で V_1 が再定義されていない頂点の集合
Des(C, V_2)：メソッド呼び出し C が実行された後に，共有変数 V_2 を再定義することなく参照している頂点の集合
条件：メソッド M はメソッド呼び出し C において呼び出される可能性があり，共有変数 V_1 と V_2 は同じ変数である

最後に，返り値を介したデータ依存関係では，二つの集合は以下のように定義される．

返り値を介したデータ依存関係

Src(M)：メソッド M において，その return 文に対してデータ依存を持つ頂点の集合
Des(C)：メソッド呼び出し C の返り値を受け取った変数を再定義すること無く参照している頂点の集合
条件：メソッド M はメソッド呼び出し C において呼び出される可能性がある

図 1 と図 2 の例では，18 行目のメソッド呼び出しに関して，33 行目から始まるメソッドについては<34>→<19>のデータ依存辺が引かれ，39 行目から始まるメソッドについては<40>→<19> のデータ依存変が引かれる．

```
33:   int operate(int a, int b) {
34:     return = a + b;
35:   }
```

図 3　return 文のオペランドが二項演算になっている例

```
33:   int operate(int a, int b) {
34:     int $1 = a + b;
35:     return $1;
36:   }
```

図 4　図 3 のソースコードに対して文の分解を行なった結果

3.3 文の分解

Java などの高級言語では，複数の処理を一つの文にまとめて記述することが可能である．たとえば，図 1 の 33 行目から始まるメソッドは，図 3 のように記述することが可能である．しかし，return 文がこのように記述されていた場合，5 行目の文とハッシュ値が等しくならないため，それらの部分をコードクローンとして検出することができない．また，return 文のオペランドに複数の変数が存在するため，3.2 で提案した返り値を介したデータ依存関係を構築することができない．この問題を解決するため，本稿では，手順 1 の MDG 構築時に，全ての文を三番地コードに分解する．この分解により，上記の return 文の例は，図 4 のコードに変換される．この例では，a と b の和を一時的に保存するための変数\$1 が定義され，その変数が return 文のオペランドに指定されている．このように分解することにより，下記の処理が保証される．

- メソッドの返り値が void でない限り，return 文は変数をオペランドとして持つ．
- メソッドの返り値を利用している場合，その値はまず変数に代入される．

よって，元のソースコード上でどのように記述されていても，その処理内容が同一であれば，その部分は SDG において同形部分グラフとなる．同様に，全てのメソッド呼び出しでは実引数が変数参照であることが保証される．

3.4 既存研究のシステム依存グラフとの違い

既存研究では SDG の構築にあたり実引数を表す頂点と仮引

数を表す頂点間にデータ依存辺を引いている [3], [6]．そのため，同様の処理が一つのメソッド内に閉じて存在している場合と複数のメソッドに分散して存在している場合にコードクローンとして検出することができない．それに対して本稿で提案する構築法では，メソッド呼び出し元と呼び出されたメソッド間でデータ依存関係がシームレスに構築されるため，検出することができる．例えば，図 2 で表す SDG から頂点を四つ以上含むコードクローンを検出した場合は，下記のクローンセット S_1 と S_2 が検出される．

$S_1 = \{\{3,4,5,6\},\{15,16,19,34\}\}$,

$S_2 = \{\{9,10,11,12\},\{15,16,19,40\}\}$

4. SDG のスライシング

前節で提案した SDG に対して単純に全てのデータ依存辺をたどると，実際には起こりえないデータフロー上からコードクローン検出を行なう場合があり，計算コストと検出精度の両面からみて好ましくない．たとえば，図 2 において，データ依存辺をたどることにより，<15>→<40>→<26>とスライスを得ることができるが，このデータフローは実際には起こりえない．このような無駄なプログラムスライスの実行を避けるため，提案手法では，メソッド呼び出しの遷移状況をスタックに保存し，スタックの状態に応じてスライシング実行の有無を決定する．スタックの初期設定，更新は下記のように行なわれる．

- スライスの開始時，スタックは空である．
- メソッド呼び出し元からメソッド呼び出し先へとデータ依存辺をたどる場合は，その基点となっている頂点をスタックに push する．例えば，図 2 において，<15>→<34>へとたどる場合は，頂点<18>がスタックに push される．
- メソッド呼び出し先からメソッド呼び出し元へとデータ依存辺をたどる場合は，その基点となっている頂点がスタックの最上部にある頂点と一致するかどうかを調べ，一致する場合は遷移し，スタックを pop する．一致しない場合やスタックが空の場合は遷移しない．例えば，図 2 において，頂点<15>からスライスを開始して，頂点<34>へとたどったとする．この時スタックの最上部は頂点<18>となっている．この状態の時，頂点<19>にはたどるが，頂点<26>にはたどらない．この理由は，頂点<26>へとたどる際の基点となっている頂点は<25>であるが，この頂点はスタックの最上部にある頂点とは異なるからである．

このようにスライスを行なうことによって，無駄なスライスのコストを減らしつつ，検出の精度を高めることができる．

5. 関 連 研 究

Horwitz らや Sinha らは，ソフトウェア全体からプログラム依存グラフを作成する方法とそのグラフにおいてプログラムスライスを用いる手法を提案している [6], [10]．彼らの手法では，メソッド呼び出し文と呼び出されたメソッドの入り口ノードを特殊な依存辺で結び，メソッド呼び出しにおけるデータ依存を表現するために，実引数を表すノードと仮引数を表すノード間にデータ依存辺を引く．かれらの提案している SDG を用いてコードクローン検出を行なえば，複数のメソッドにまたがったコードクローンは検出可能である．しかし図 1 で示したような，一方が一つのメソッド内で完結した処理であり，他方が複数のメソッドにまたがる処理となっていた場合は，コードクローンとして検出することができない．

吉田らは，複数のメソッドにまたがって存在しているコードクローンのパターンを分類し，そのパターンに応じて集約する方法を提案している [11]．彼らの手法ではまずメソッド内で閉じたコードクローンが検出される．そして，そのコードクローンをシステムのクラス階層上での位置関係から，いくつかのパターンに分類している．つまり，各メソッド内の重複コードは，それぞれが単体でコードクローンとして検出されうる規模でなければならない．

字句単位や行単位の検出手法 [2], [7] を用いることによって，連続して定義されている複数のメソッドを一つのコードクローンとして検出することは可能であるが，このようなコードクローンは，単に連続して定義されているという関係が成り立っているだけであり，コードクローンに含まれるメソッドが共通の処理を行なっているや，呼び出し関係を持っているということは保証されないため，人間が見て有益と感じるコードクローンとなることはほとんどない．

6. お わ り に

本稿では，複数にまたがったコードクローンを検出することを目的としたプログラム依存グラフの提案とそれを用いたスライシング手法の提案を行なった．本稿で提案した手法を用いれば，処理が一つのメソッド内で閉じていても，複数のメソッドに分散していても，処理として重複していれば，コードクローンとして検出化膿である．今後はツールの実装と評価を行なう予定である．本手法において検出されるコードクローンは，複数のメソッドやファイルにまたがっているため，ユーザが理解しやすいかたちでコードクローンを可視化することが重用である．

謝辞

本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究 (A)(課題番号：21240002) および (C)(課題番号：20500033)，文部科学省科学研究費補助金若手研究 (B)(課題番号：22700031) の助成を得た．

文　　献

[1] Clone Detection Literature. http://www.cis.uab.edu/tairasr/clones/literature/.

[2] Simian. http://www.redhillconsulting.com.au/products/simian/.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Watten. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] 肥後, 楠本. プログラム依存グラフを用いたコードクローン検出法の改良と評価. 情報処理学会論文誌, 51(12):xxxx–xxxx, Dec.

2010.

[5] 肥後, 楠本, 井上. コードクローン検出とその関連技術. 電子情報通信学会論文誌 D, J91-D(6):1465–1481, June 2008.

[6] S. Horwitz, T. Reps, and D. Binkly. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan 1990.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[8] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proc. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pp. 155–169, Jan. 2000.

[9] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University, 2007.

[10] S. Sinha, M. J. Harrold, and G. Rothermel. System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow, May 1999.

[11] 吉田, 肥後, 楠本, 井上. コードクローン間の依存関係に基づくリファクタリング支援. 情報処理学会論文誌, 48(3):1431–1442, Mar. 2007.