実規模ソフトウェアへの適用を目的とした プログラム依存グラフに基づくコードクローン検出法

肥後芳樹[†] 楠本真二[†]

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，全ての面において他の検出手法よりも優れているものはない．プログラム依存グラフを用いた検出の長所は非連続コードクローンを検出できることである．しかし，その反面，連続コードクローンについては，他の検出技術に比べて検出能力が低い．また，検出に必要な計算コストが高いため，実規模ソフトウェアに対しては適用が難しいという弱点もある．本論文では，実規模ソフトウェアに対して適用することを目的とした，プログラム依存グラフを用いたコードクローン検出手法を提案する．また，提案手法では，検出方法そのものにも改良を行っており，既存の手法では検出できないコードクローンも検出可能である．提案した手法を複数のソフトウェアに対して適用し，その有効性を確認した．

Code Clone Detection with Program Dependency Graph for Actual Software Systems

YOSHIKI HIGO[†] and SHINJI KUSUMOTO[†]

At present, there are various kinds of code clone detection techniques. Each detection technique has merits and demerits, and none of them is superior to any other techniques in every way. PDG-based detection is suitable to detect non-continuous code clones meanwhile other detection techniques are not suited to detect them. However, there is a tendency that it cannot detect continuous code clones unlike line-based or token-based techniques. Moreover, PDG-based detection is time-consuming, and it is difficult to apply it to actual software systems. This paper proposes a new PDG-based detection method, which can be applied to actual software systems. Also, the proposed method improves a traditional detection algorithm, and it can detect code clones that cannot be detected by existing PDG-based detection methods. We confirmed the usefulness of the proposed method by applying it to multiple real software systems.

1. はじめに

近年，ソフトウェア工学の研究対象の 1 つとしてコードクローンが注目を集めており，これまでにコードクローンに関係するさまざまな研究がなされている[1),10)]．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．コードクローンは，コピーアンドペーストや定型処理などのさまざまな理由によりソースコード中に作り込まれる[3)]．

コードクローン検出ツールは，対象のソースコードを入力として受け取り，その中に含まれるコードクローンの位置情報を出力する．しかし，コードクローンの厳密で普遍的な定義は存在せず，各手法が独自にコードクローンの定義を持ち，その定義に基づいて検出を行う．そのため同じソースコードから検出を行った場合でも検出結果はツール毎に異なる．

既存の検出手法は用いている技術により，行単位での検出，字句単位での検出，抽象構文木を用いた検出，プログラム依存グラフを用いた検出，メトリクスを用いた検出に大別される[6)]．各検出技術は一長一短であり全ての面において他の技術よりも優れているものはない[4),5)]．コードクローン検出を行う状況に応じて，適切な検出技術を選択することが重要である．

プログラム依存グラフ（Program Dependency Graph, 以降 PDG）を用いた検出の長所と短所を示す[4),5),7),8)]．

長所 非連続コードクローンを検出することができる．非連続コードクローンとは，1 つのコードクローンを構成する要素（プログラムの文や式など）が必ずしもソースコード上で連続していないコードクローンを指す．

短所 行単位，字句単位，および抽象構文木を用いた検出に比べると連続コードクローンの検出能力が劣る．また，検出に必要な計算コストが高いため，

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

実規模のソフトウェアに対しては適用が難しい．

本論文では、実規模ソフトウェアへの適用を目的とした，計算コストを抑えた，PDG を用いたコードクローン検出手法を提案する．また，提案手法は検出方法そのものにも工夫を行っており，従来の PDG を用いた手法では検出できないコードクローンも検出可能である．

2. 関 連 研 究

Komondoor らは初めて PDG を用いたコードクローン検出を行った[7]．彼らの手法では，ソースコード中の文と条件式が PDG の頂点である．全ての頂点からハッシュ値が算出され，ハッシュ値が等しい頂点が同じグループに分類される．グループ内の全ての頂点のペアから，それらを含む同型部分グラフが求められ，それらがコードクローンとされる．同型部分グラフの特定には主にバックワードスライスが用いられ，フォワードスライスは，条件式の頂点が同型部分グラフに追加された時のみ用いられる．文献 7) の実験では，PDG を用いた手法の高い検出能力が示されたが，約 11,000 行のソフトウェアからの検出に約 90 分を要するなど，その実用性の低さも顕著であった．

Krinke は細粒度 PDG を用いたコードクローン検出手法を提案している[8]．細粒度 PDG では，頂点は抽象構文木の頂点とほぼ 1 対 1 で対応しており，通常の PDG に比べて頂点数がはるかに多くなる．そのため，単純に検出処理を行った場合は，Komondoor らの手法[7] に比べてさらに大きな計算コストを必要とする．この問題に対処するため，Krinke は k-limited search という手法を提案している．k-limited search では，同型部分グラフはスライス基点の k ホップ内でのみ探索されるため計算コストを抑えることができる．しかし，最も大きな同型部分グラフが特定されない可能性がある．Krinke の手法では，フォワードスライスが用いられており，計算コストを抑えるために，スライス基点となる頂点は条件式の頂点に限定されている．文献 8) の実験では，k-limited search により計算コストを大幅に抑えれることが示された．k が 20 以下の場合は，検出処理は数分以内で完了したが，k が 25 以上の場合は，約 25,000 行のソフトウェアからの検出に約 46 時間を要するなど，大きなコードクローンを検出するには，高い計算コストが必要であった．

Komondoor らの手法[7] が高い計算コストを必要とするのは，ハッシュ値の等しい頂点の全ての組み合わせについて同型部分グラフの検出を行っているからであり，Krinke の手法[8] が高い計算コストを必要とするのは，同型部分グラフを探索するグラフが巨大であるからである．このように手法ごとに高い計算コストが必要な処理は異なる．用いる PDG に合わせて，適切な計算コストを抑える工夫を行うことにより，PDG を用いた検出は実用的な手法になりうると著者らは考えた．

3. プログラム依存グラフ

本節では，提案手法で用いる PDG の構築方法について述べる．提案手法では，まず制御フローグラフ（Control Flow Graph, 以降 CFG）が構築され，それを用いて PDG が構築される．以降，3.1 節と 3.2 節ではそれぞれ，CFG と PDG の構築方法について説明する．

3.1 制御フローグラフの構築

CFG はソースコードの表現方法の 1 つであり，プログラムの開始から終了までの全ての実行経路を表す．CFG はコンパイラ最適化やプログラム静的解析の分野でしばしば用いられる．本論文では，CFG の頂点と辺は下記のとおりである．

頂点 文と条件節の式．例えば，図 1(a) では，文「`System.out.println(i)`」,式「`i < 10`」,「`int i = 0`」,「`i++`」が頂点となる．

辺 要素 A の実行直後に要素 B が実行される可能性がある場合，A の頂点から B の頂点へ辺を引く．

本論文では，CFG は各メソッド（と各コンストラクタ）に対して作成される．CFG は，メソッド内の構造において，その内側から順に構築される．つまり，メソッド内の最も深いブロック（ブロックとは，if 文や while 文などを表す）の CFG がまず構築され，それを用いてその外側のブロックの CFG が構築される．内側のブロックの CFG を組み合わせてその外側のブロックの CFG を構築する処理は，外側のブロックがメソッドになるまで繰り返えされる．図 1(b) は図 1(a) に示すソースコードから構築される CFG を表している．

3.2 プログラム依存グラフの構築

PDG は CFG を基にして，下記の方法で構築される．

頂点の生成 1 つの CFG の頂点から 1 つの PDG の頂点が生成される．つまり CFG と PDG の頂点数は完全に一致する．

データ依存辺の生成 CFG において，変数を定義または変数に対して代入を行っている各頂点に対して，その変数を参照している頂点を制御フローをたどりながら検出する．そして，PDG において，定義している頂点から参照している頂点に向けてデータ依存辺を引く．

制御依存辺の生成 条件式を表す頂点から，ソースコード上においてその内部に存在するプログラ

ム要素の頂点に対して制御依存辺を引く．
図 1(c) は図 1(b) の CFG から構築される PDG である．

4. PDG を用いたコードクローン検出法

本節では，3 節で述べた PDG を用いてコードクローンを検出する手法を提案する．提案手法は以下の 4 つの STEP からなる．

STEP1 PDG 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．ハッシュ値が計算される前に，変数やリテラルはその型に変換されるため，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，それらは同じハッシュ値を持つ．

STEP2 同じグループに属する頂点のペア ($r1$,$r2$) を含む同型部分グラフを検出する．同型部分グラフの検出にはフォワードスライスとバックワードスライス[11] の 2 つを用いる．スライス基点は $r1$ と $r2$ であり，2 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型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，たどった頂点は同型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 ($p1, p2$) が異なるハッシュ値を持つ場合．
- ($p1, p2$) のハッシュ値は等しいが，$r1$ のグラフ（または $r2$ のグラフ）がすでに $p1$（または $p2$）を含んでいる場合（無限ループを回避するための処理）．
- ($p1, p2$) のハッシュ値は等しいが，$r1$ のグラフ（または $r2$ のグラフ）が $p2$（または $p1$）を含んでいる場合（2 つの同型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（2 つの同型部分グラフ）が本手法において検出されるクローンペアである．

STEP3 クローンペア ($s1$, $s2$) が他のクローンペア ($s1'$, $s2'$) に含まれている場合（$s1 \subseteq s1' \cap s2 \subseteq s2'$），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．

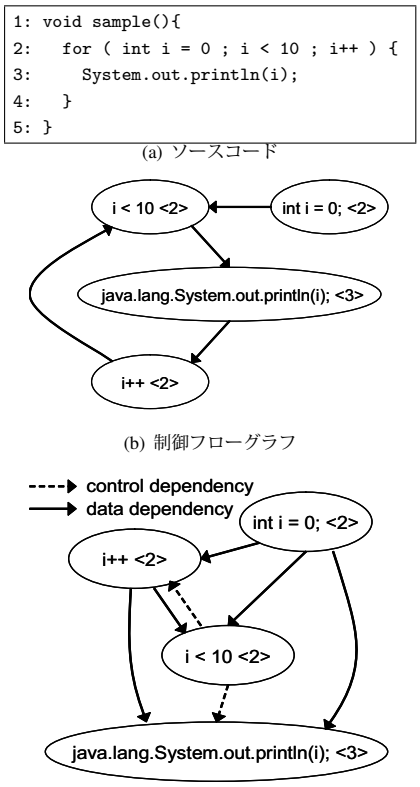


図 1 制御フローグラフとプログラム依存グラフの例

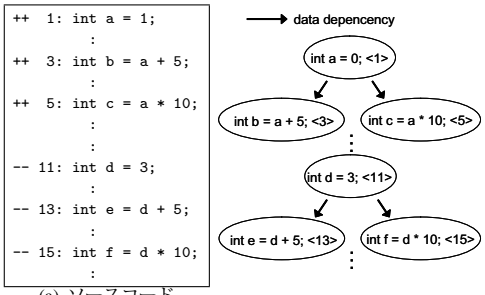
STEP4 同じグラフを持つクローンペアからクローンセットを形成する．例えば，2 つのクローンペア ($s1$, $s2$), ($s2$, $s3$) があった場合，クローンセット $\{s1, s2, s3\}$ が形成される．

4.1 フォワードスライスとバックワードスライスの両方をコードクローン検出に用いる理由

2 節で紹介した先行研究は，フォワードスライスとバックワードスライスのどちらか 1 つしか用いていないのに対して，提案手法では両方を用いる．その理由は，バックワードスライスを用いて検出される一部の同型部分グラフはフォワードスライスでは検出されず，またその逆もあるからである．

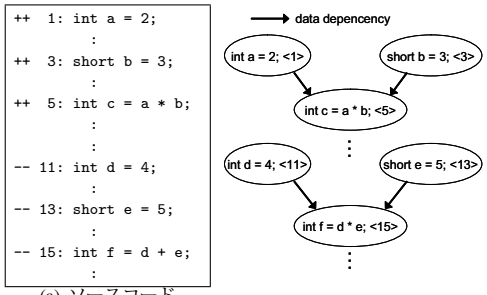
図 2 と図 3 では，++で始まる 3 行と−−で始まる 3 行がクローンペアである．図 2 では，各同型部分グラフにおいて 1 つの変数が 2 つの文で参照されている．そのため，1 つの頂点から 2 つの頂点に向けてデータ依存辺が存在する．例えば，コードクローン<1,3,5>☆では，文<1> → 文<3>と文<1> → 文<5>の 2 つのデータ依存辺が存在する．この例のような場合，どの頂点

☆ 1，3，5 行目のプログラム要素からなるコードクローン



(a) ソースコード　(b) プログラム依存グラフ

図 2　バックワードスライスでは特定できない同型部分グラフの例



(a) ソースコード　(b) プログラム依存グラフ

図 3　フォワードスライスでは特定できない同型部分グラフの例

をスライス基点としてもバックワードスライスでは 3 つの頂点からなる同型部分グラフを特定できないが，フォワードスライスを用いることにより特定できる．

図 3 では，各同型部分グラフにおいて，2 つの変数が 1 つの文で参照されている．そのため，2 つの頂点から，1 つの頂点に向けてデータ依存辺が存在する．例えば，コードクローン<1,3,5>では，文<1> → 文<5>と文<3> → 文<5>の 2 つのデータ依存辺が存在する．この例のような場合，どの頂点をスライス基点としてもフォワードスライスでは 3 つの頂点からなる同型部分グラフを特定できないが，バックワードスライスを用いることにより特定できる．このように 2 つのスライスを用いることで，1 つのスライスでは特定できない同型部分グラフを特定できるようになる．

4.2　計算コストの削減

PDG を用いたコードクローン検出の計算コストが高いのは 2 つの理由が考えられる．1 つ目の理由は，スライス基点として使用される頂点のペア数が非常に多いことである．2 つ目の理由は，同型の部分グラフを特定することそのものが，NP 完全な，難しい問題であるためである．本論文では，理由 1 の側面から計算コストを削減する手法を提案する．その理由は，理由 2 を改善することは容易ではなく，また，本論文で用いる PDG はメソッド単位で作成されるため，あまりに巨大なグラフが作成されることはないためである．

検出処理の STEP2 において，n 個の頂点が同じハッシュ値を持つ場合，それらのペアをスライス基点とする組み合わせの数は $n(n-1)/2$ となる．実規模ソフトウェアの場合，同じハッシュ値を持つ頂点の数は，数千あるいはそれ以上になる場合があり，全ての組み合わせに対して同型部分グラフを特定する処理は非常に高い計算コストを必要とする．しかし，数千もの頂点が同じハッシュ値を持つ理由は，それらがプログラムで頻繁に表れる処理であるからであり（例えば，変数宣言 `int i;` と `int j;` や，リターン文 `return a;` と `return b;`），それらがコピーアンドペーストにより生成されたわけではない．そこで，本論文では，同じハッシュ値を持つ頂点数の閾値を設け，その閾値よりも多くの数の頂点数が同じハッシュ値を持つ場合は，それらはスライス基点としては用いない．

5. 適 用 実 験

提案手法を Java 言語を用いて実装し[2]，適用実験を行った．この実験では，さまざまな規模のソフトウェアを対象とした (表 1 参照)．この表では，「行数」は空白行やコメント行を含んだ総行数，「頂点数」は実装したツールにおいて生成された PDG の頂点数を表す．「前処理時間」は，4 節の STEP1 を行うまでに必要な全ての処理，つまり，ソースコードの読み込み，意味解析，そして PDG の構築に要した合計時間を表す．また，この実験では，計算コスト削減手法の閾値として 50,100,200,300,500,1000 の 6 つを用い，5 つ以上のプログラム要素からなるコードクローンを検出した．

5.1　両方のスライスを用いた効果

どちらか一方のスライスを用いた場合に検出されるコードクローンが，両方のスライスを用いて検出されるコードクローンに対して，どの程度の網羅率があるのかを下記の式を用いて調査した．

$$coverage(r1, r2) = 100 \times \frac{\mid S(r1) \cap S(r2) \mid}{\mid S(r1) \mid} (1)$$

なお，各変数は下記のとおりとする．

r1: 両方のスライスを用いた場合の検出結果

r2: どちらか一方のスライスを用いた場合の検出結果

S(r): 検出結果 r に含まれるプログラムの要素

調査結果を表 2 に示す．なお，JDK において閾値なしの場合は，両方のスライスを用いた検出処理が一晩では終了しなかったため，途中で処理を中断した．

閾値 50 の場合では，網羅率は約 37% ～ 61% と低く，両方のスライスを用いることにより，コードクローンとして検出されるプログラム要素数が約 2 倍になっ

表 1 対象ソフトウェア

ソフトウェア	バージョン	行数	ファイル数	頂点数	前処理時間 (秒)
XDoclet	1.2.3	14,927	58	3,936	5.02
FreeMind	8.1.0	38,522	197	13,726	12.41
JHotDraw	7.1	78,528	411	29,504	27.10
Ant	1.7.1	188,570	787	52,333	39.82
Tomcat	6.0.18	308,176	1,053	94,234	65.48
JDK	1.6.0_13	2,023,518	7,154	518,710	458.29

表 2 1 つのスライスを用いた検出結果の，2 つのスライスを用いた検出結果に対する網羅率

ソフトウェア	用いたスライス	2 つのスライスを用いた検出結果に対する網羅率						
		閾値 50	閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	閾値なし
XDoclet	フォワード	48.34 %	62.12 %	62.12 %	62.12 %	62.12 %	78.20 %	78.29 %
	バックワード	55.51 %	56.66 %	56.66 %	56.66 %	56.66 %	56.60 %	56.60 %
FreeMind	フォワード	40.78 %	42.83 %	48.84 %	48.84 %	48.84 %	48.84 %	68.90 %
	バックワード	55.74 %	54.18 %	55.48 %	55.48 %	55.48 %	55.48 %	55.45 %
JHotDraw	フォワード	61.50 %	63.04 %	64.88 %	65.51 %	65.51 %	65.52 %	78.96 %
	バックワード	50.55 %	53.57 %	57.45 %	57.61 %	57.61 %	57.62 %	57.62 %
Ant	フォワード	45.61 %	49.04 %	53.95 %	54.10 %	55.44 %	64.60 %	77.92 %
	バックワード	51.01 %	53.32 %	54.31 %	55.22 %	55.78 %	60.96 %	61.19 %
Tomcat	フォワード	47.32 %	53.12 %	55.54 %	57.12 %	58.54 %	58.96 %	82.17 %
	バックワード	53.18 %	53.91 %	55.08 %	55.89 %	56.63 %	59.04 %	62.41 %
JDK	フォワード	37.19 %	39.69 %	42.35 %	43.74 %	46.72 %	47.63 %	-
	バックワード	51.86 %	54.85 %	56.77 %	57.42 %	58.27 %	60.57 %	-

ていることがわかる．閾値が上がるにつれて網羅率も上昇するが，閾値なしの場合でも網羅率は約 55% ～ 82%に留まっている．

閾値が低い場合に網羅率が低くなるのは，次の理由からである．例えば，バックワードスライスしか用いない場合は，PDG を逆順にたどることしかできないため，逆順のみで同型部分グラフ上の全ての頂点をたどることができる頂点がスライス基点になっていない場合は，網羅率は低くなってしまう．閾値を用いることにより，このような「すべての頂点を 1 つのスライスのみでたどれる頂点」がスライス基点から除かれてしまい，網羅率が低くなってしまったと考えられる．また，閾値なしの場合でも網羅率が 100%になっていないのは，4.1 節で紹介したような，そもそも 1 つのスライスのみでは完全に特定できない同型部分グラフが存在したためである．閾値なしの場合の網羅率が約 55% ～ 82%ということは，残りの 18% ～ 45%はそのような同型部分グラフの頂点であったことを示している．

また，表 3 は各スライスを用いた場合の検出時間を表している．ここでの検出時間は，4 節で説明した STEP1 ～ STEP4，つまり PDG 頂点のハッシュ値の計算から，クローンセットの検出までに要した時間である．表 3 より，両方のスライスを用いた場合は，1 つのスライスを用いた場合に比べて多くの時間を必要としていることがわかる．この原因は，(1) 両方のスライスを用いることにより同型グラフの探索範囲が増加したため，また，(2) それにより検出されるクローンペアの数が増え，STEP3（不必要なクローンペアの除去）および STEP4（クローンセットの形成）により多くの時間が必要になったため，である．

5.2 計算コスト削減手法の効果

計算コスト削減手法の効果について調査を行った．この調査では全てのコードクローン検出は両方のスライスを用いて行った．表 4 は各閾値における，スライス基点として使用された頂点のペア数（括弧内の数値は閾値なしの場合に対する割合）を表している．この表より，閾値なしの場合は，膨大な数のペアがスライス基点として使用されたことがわかる．閾値を用いることにより，スライス基点になるペア数が閾値なしの場合に比べて，かなり削減されている．削減率は規模の大きいソフトウェアほど大きく，例えば，JDK の閾値 50 の場合は，閾値なしの場合に比べて約 0.01%のペアのみがスライス基点として用いられている．この頂点数の削減の効果が，表 3 において，検出時間の短縮というかたちで現れている．

次に，閾値を用いた場合に検出されるコードクローンは，閾値を用いない場合に対してどの程度の網羅率があるのかを，式 (1) を用いて調査した．この調査では各変数は以下のものを用いた．

r1: 閾値を用いない場合の検出結果

表 3 各スライスを用いた検出時間

ソフトウェア	用いたスライス	検出時間（秒）						
		閾値 50	閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	閾値なし
XDoclet	両方	1.04	1.55	1.55	1.56	1.54	3.53	3.49
	フォワード	0.39	0.55	0.58	0.62	0.55	2.97	2.93
	バックワード	0.33	0.48	0.48	0.50	0.51	0.46	0.47
FreeMind	両方	1.97	3.25	4.80	4.75	4.56	4.78	29.41
	フォワード	1.05	1.63	2.21	1.91	1.78	2.19	26.83
	バックワード	0.57	0.82	0.97	1.03	1.41	1.43	4.05
JHotDraw	両方	22.39	23.03	43.97	158.85	160.03	161.61	293.70
	フォワード	2.10	9.50	10.01	11.71	12.80	11.49	144.39
	バックワード	1.43	4.48	10.14	10.69	12.38	10.69	24.41
Ant	両方	8.39	7.35	12.92	20.37	20.05	61.80	336.82
	フォワード	2.68	4.88	8.52	9.98	9.35	35.32	312.94
	バックワード	3.02	2.99	4.92	6.24	7.26	16.35	47.32
Tomcat	両方	31.50	65.52	180.90	193.23	213.71	255.83	1407.27
	フォワード	5.68	12.04	14.64	14.87	21.96	26.89	1083.77
	バックワード	5.52	6.13	9.44	9.16	16.81	26.30	146.06
JDK	両方	1910.55	6288.34	9209.92	9207.60	9496.63	9681.43	-
	フォワード	47.55	189.55	192.67	219.37	268.36	325.11	-
	バックワード	47.88	75.02	102.45	105.42	131.37	155.52	-

表 4 スライス基点として使用される頂点数（括弧内の数値は閾値なしの場合に対する割合）

ソフトウェア	スライス基点として使用される頂点数						
	閾値 50	閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	閾値なし
XDoclet	12,878 (5.03 %)	25,795 (10.07 %)	25,795 (10.07 %)	25,795 (10.07 %)	25,795 (10.07 %)	255,976 (100.00 %)	255,976 -
FreeMind	24,483 (0.67 %)	61,684 (1.70 %)	130,150 (3.60 %)	130,150 (3.60 %)	130,150 (3.60 %)	130,150 (3.60 %)	3,610,991 -
JHotDraw	80,140 (0.53 %)	133,343 (0.89 %)	298,116 (1.99 %)	371,290 (2.48 %)	371,290 (2.48 %)	371,290 (2.48 %)	14,916,211 -
Ant	111,655 (0.31 %)	205,290 (0.57 %)	427,300 (1.19 %)	716,265 (2.00 %)	998,170 (2.79 %)	2,932,741 (8.21 %)	35,699,966 -
Tomcat	200,853 (0.17 %)	364,189 (0.32 %)	619,853 (0.54 %)	994,548 (0.87 %)	1,419,842 (1.25 %)	4,462,410 (3.93 %)	113,311,371 -
JDK	982,448 (0.01 %)	2,106,308 (0.03 %)	3,963,608 (0.07 %)	4,991,945 (0.09 %)	8,408,394 (0.15 %)	14,108,189 (0.25 %)	5,543,539,373 -

表 5 閾値なしの検出結果に対する網羅率

ソフトウェア	閾値なしの検出結果に対する網羅率					
	閾値 50	閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000
XDoclet	98.01 %	99.90 %	99.90 %	99.90 %	99.90 %	100.00 %
FreeMind	91.17 %	97.63 %	99.94 %	99.94 %	99.94 %	99.94 %
JHotDraw	96.37 %	99.63 %	99.89 %	99.97 %	99.97 %	99.97 %
Ant	91.48 %	94.59 %	97.71 %	98.50 %	99.04 %	99.88 %
Tomcat	90.60 %	94.92 %	96.54 %	97.63 %	98.94 %	99.35 %

r2: 閾値を用いた場合の検出結果

S(r): 検出結果 r に含まれるプログラムの要素

表 5 は，各ソフトウェアと各閾値における網羅率をまとめたものである．閾値を用いた場合，最低でも約 90%のプログラム要素がコードクローンとして検出されていることがわかる．閾値を用いた場合のスライス基点数が用いなかった場合の高々数%だったことを考慮すると，この網羅率の値は非常に高いといえる．

5.3 他の手法との比較

既存の実用的な検出手法と比較して，どの程度検出されるコードクローンに違いがあるのかを調査した．比較対象として，字句単位の検出ツールである CCFinder を選択した．その理由は，(1)CCFinder は広く用いられているツールであること，(2) 実用性の

表 6 ツール間における検出されるコードクローンの違い（網羅率 $C1$ と $C2$）

ソフトウェア	網羅率	閾値 50	閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	閾値なし
XDoclet	$C1$	36.07 %	35.46 %	35.46 %	35.46 %	35.46 %	35.46 %	35.46 %
	$C2$	38.58 %	38.71 %	38.71 %	38.71 %	38.71 %	38.71 %	38.71 %
FreeMind	$C1$	40.80 %	41.56 %	42.11 %	42.11 %	42.11 %	42.11 %	42.11 %
	$C2$	18.80 %	20.23 %	21.04 %	21.04 %	21.04 %	21.04 %	21.04 %
JHotDraw	$C1$	70.45 %	71.19 %	71.12 %	71.07 %	71.07 %	71.07 %	71.03 %
	$C2$	44.25 %	46.17 %	46.22 %	46.22 %	46.22 %	46.22 %	46.22 %
Ant	$C1$	46.12 %	45.36 %	44.51 %	44.48 %	44.37 %	44.17 %	44.14 %
	$C2$	32.28 %	32.69 %	33.09 %	33.36 %	33.41 %	33.42 %	33.42 %
Tomcat	$C1$	55.68 %	55.93 %	55.70 %	55.55 %	55.28 %	55.10 %	54.89 %
	$C2$	36.38 %	38.35 %	38.80 %	39.11 %	39.35 %	39.37 %	39.40 %
JDK	$C1$	53.42 %	53.56 %	53.86 %	53.73 %	53.37 %	53.10 %	-
	$C2$	33.30 %	35.54 %	37.19 %	37.27 %	37.38 %	37.55 %	-

高いツールであり，この実験対象全てのソフトウェアを解析可能であること，の 2 つである．なお，この調査では，CCFinder を用いて 30 字句以上のコードクローンを検出した．紙面の都合上 CCFinder の実行時間の詳細については割愛するが，いずれのソフトウェアに対しても提案手法よりも短時間でコードクローンを検出した．なお，この調査でも，提案手法では両方のスライスを用いてコードクローン検出を行った．

CCFinder の検出できなかったコードクローンを提案手法がどの程度検出できたのかを調査するため，式 (1) と下記の変数を用いて網羅率 $C1$ を算出した．

r1: 提案手法の検出結果

r2: CCFinder の検出結果

S(r): 検出結果 r に含まれるプログラムの行☆.

$C1$ の値が低いほど，提案手法は CCFinder が検出できなかったコードクローンを検出できたこと意味する．

また，提案手法が検出できなかったコードクローンをどの程度 CCFinder が検出できたのかを調査するため，下記の変数を用いて網羅率 $C2$ を算出した．

r1: CCFinder の検出結果

r2: 提案手法の検出結果

S(r): 検出結果 r に含まれるプログラムの行.

$C2$ の値が低いほど，CCFinder は提案手法が検出できなかったコードクローンを検出できたことを意味する．

表 6 は，網羅率 $C1$ と $C2$ の値をまとめたものである．この表より，提案手法が検出したコードクローンのうちの約 37% ～ 70%が CCFinder によっても検出されているのがわかる．つまり，残りの約 30% ～ 63%は提案手法よってのみ検出されたコードクローンであり，提案手法を用いたコードクローン検出の価値があることを表している．一方，CCFinder の検出したコードクローンのうちの約 17% ～ 38%が提案手法によっても検出されているのがわかる．つまり，CCFinder の検出した 62% ～ 83%は提案手法は検出できていない．提案手法と CCFinder はそれぞれ，他方が検出できないコードクローンをある程度検出できている．

☆ 提案手法と CCFinder では検出されるコードクローンの単位が異なるため，直接比較することはできない．そのため，検出されたコードクローンが位置する行を用いて網羅率を計算した．

5.4 検出されたコードクローンの例

検出されたコードクローンを 1 つ紹介する．図 5 は JHotDraw から検出されたコードクローンである．++ で始まる行はコードクローンに含まれるプログラム要素である．このクローンセットは，5 個のコードクローンからなり，全てが異なるクラスに存在していた．このコードクローンは，新しいフレーム（JFrame）を作成，サイズや内部コンポーネントの設定，そして，表示を行うという処理から成る．

今後，同様の処理を実装する必要がある場合に，このコードクローンを参考にすることで，素早く実装を完了することができる．また，すでに 5 つ重複しているため，Template Method パターンなどを用いて，重複コードを共通化することも考えられる．

図 5 は，図 4(a) のコードクローンの PDG を表している．この図の頂点の識別子は，図 4(a) のプログラム要素の識別子と対応している．このコードクローンは逐次処理のみから成るため，制御依存辺は存在しない．図 5 からわかるように，このコードクローンは，1 つスライスのみを用いた場合には，どの頂点をスライス基点とした場合でも他の全ての頂点をたどることはできない．つまり，これは両方のスライスを用いることによってはじめて検出されるコードクローンである．

6. おわりに

本論文では，PDG を用いたコードクローン検出手法を提案した．また提案手法を実装したツールを用い

```
      cf.setStartConnector(ta.findConnector(
                  Geom.center(ta.getBounds()),cf));
      cf.setEndConnector(tb.findConnector(
                  Geom.center(tb.getBounds()),cf));

a ++  Drawing drawing = new DefaultDrawing();
            :
b ++  JFrame f = new JFrame("My Drawing");
c ++  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
d ++  f.setSize(400,300);

e ++  DrawingView view = new DefaultDrawingView();
f ++  view.setDrawing(drawing);
g ++  f.getContentPane().add(view.getComponent());

i ++  f.show();
            :
```

(a) コードクローン 1

```
          :
++  Drawing drawing = new DefaultDrawing();
          :
++  JFrame f = new JFrame("UltraMini");
++  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
++  f.setSize(600,300);

++  DrawingView view = new DefaultDrawingView();
++  view.setDrawing(drawing);
++  f.getContentPane().add(view.getComponent());

    DrawingEditor editor = new DefaultDrawingEditor();
    editor.add(view);
    editor.setTool(new SelectionTool());

++  f.show();
          :
```

(b) コードクローン 2

図 4 JHotDraw から検出されたコードクローン

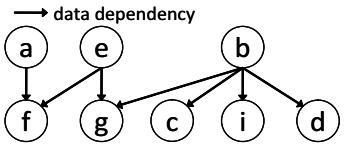


図 5 図 4(a) のコードクローンの PDG

てオープンソースソフトウェアに対して実験を行った．実験により，提案手法の実規模ソフトウェアに対する有用性を確認することができた．

現在は，メソッド単位の PDG を用いてコードクローンを検出しているが，今後は対象システム全体の PDG からの検出を行う予定である．これにより，複数のメソッドにまたがる処理を 1 つのコードクローンとして検出することが可能となり，より有益で興味深いコードクローンの検出が見込まれる．また，複数のソフトウェア間でのコードクローン検出も予定している．著者らが知るかぎり PDG を用いてソフトウェア間のコードクローンを検出している研究はない．Simone らは 7,000 以上のシステムからコードクローンを検出し，検出されたコードクローンの分析を行っているが，彼らは字句単位でのコードクローンを検出しているため，非連続コードクローンは分析の対象外であった[9]．システム間の非連続コードクローンを検出し，分析することで新たな発見があると期待している．

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，文部科学省科学研究費補助金基盤研究 (C)(課題番号:20500033) の助成を得た．

参 考 文 献

1) : Clone Detection Literature, http://www.cis.uab.edu/tairasr/clones/literature/.
2) : Scorpio, http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e/.
3) Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. of International Conference on Software Maintenance 98*, pp. 368–377 (1998).
4) Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 804–818 (2007).
5) Burd, E. and Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance, *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36–43 (2002).
6) 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, 電子情報通信学会論文誌 D, Vol. J91-D, No. 6, pp. 1465–1481 (2008).
7) Komondoor, R. and Horwitz, S.: Semantics-Preserving Procedure Extraction, *Proc. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pp. 155–169 (2000).
8) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. of the 8th Working Conference on Reverse Engineering*, pp.301–309 (2001).
9) Livieri, S., Higo, Y., Matsushita, M. and Inoue, K.: Very-Large Scale Code Clone Analysis and Visualization of Open Source Program Using Distributed CCFinder: D-CCFinder, *Proc. of the 29th International Conference on Software Engineering*, pp.106–115 (2007).
10) Roy, C. K. and Cordy, J. R.: A Survey on Software Clone Detection Research, Technical report, School of Computing, Queen's University (2007).
11) 下村隆夫: プログラムスライシング技術と応用, 共立出版 (1995).