コードクローン検出に必要な計算コストの削減を目的としたプログラム依存グラフ頂点集約法の提案

肥 後 芳 樹† 楠 本 真 二†

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の長所は，非連続コードクローンを検出できる点である．しかし，連続コードクローンの検出能力は，行単位での検出や字句単位での検出など，他の検出技術に比べて劣っている．また，コードクローンの検出に高い計算コストを必要とし，実規模ソフトウェアについては，適用が難しいという問題点もある．前者の問題については，著者らは，プログラム依存グラフに実行依存という新しい依存関係を導入し，実験により連続コードクローンの検出能力が向上していることを確認した．本論文では，後者の問題点を改善するために，実行依存つきプログラム依存グラフに対して頂点の集約を行う手法を提案する．提案手法を用いることにより，プログラム依存グラフの規模が小さくなるため，検出に必要な計算コストを抑えることができる．

Merging Nodes on Program Dependency Graph for Reducing Computational Cost of Code Clone Detection

Yoshiki Higo† and Shinji Kusumoto†

The advantage of PDG-based code clone detection is that it has a good capability to detect non-contiguous code clones meanwhile it is less suited to detect contiguous code clones than other detection techniques like line-based or token-based. Also, it is unrealistic to apply it to practical size software because it requires high computational cost for detection. In order to resolve the former problem, the authors previously proposed to add a new dependencey, **execution dependency** into PDG. We confirmed that the introduction of execution dependency improved the detection capability by several case studies. The present paper focuses on the latter problem. We propose a method merging multiple nodes as a single node on PDGs. The merging reduces the scale of PDGs, so that less compurational cost is required for detection.

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．コードクローンはコピーアンドペーストや定型処理，プログラミング言語の制限などの理由によりソースコード中に作り込まれる[3]．例えば，Kim らは，開発者は 1 時間あたり約 4 回のコード片単位でのコピーアンドペーストを行なっていると報告している[11]．

近年，ソフトウェア開発や保守を支援するためにコードクローン情報を利用することが注目されており，これまでにさまざまな利用方法が提案されている[1),8),15]．しかし，コードクローンをソースコード中から手作業で見つけること，また見つけたコードクローンをドキュメント等を用いて管理することは現実的ではない．そのため，コードクローン情報が必要な場合に，ツールを用いた自動的な検出が行なわれる[8]．

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なう．そのため同じソースコードから検出を行な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[8]．各検出技術は一長一短であり，全ての面において他の検出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なう状況に応じて，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を用いた検出の長所は，他の検出技術では検出することが難しい**非連続コードクローン**を検出できることである[4),5),12),13]．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

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

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．しかし，その一方で連続コードクローンの検出能力に関しては，他の検出技術に劣る．この問題を解決するために，著者らは，プログラム依存グラフに実行依存という新しい依存関係を導入することにより，連続コードクローンの検出能力を高めた手法を提案した[10]．

また，プログラム依存グラフを用いた検出には，計算コストが高く実規模ソフトウェアに対しては適用が難しいという短所もある．本論文では，この問題を改善するための手法として，プログラム依存グラフにおいて，頂点を集約する方法を提案する．この提案手法では，実行依存において到達可能な複数の頂点を 1 つに集約するため，実行依存をたどることによる計算コストの増加を抑えることができる．また，頂点数が減るため，集約した頂点を基点とするプログラムスライシングを行なう回数が減り，計算コストが削減できる．

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（Program Dependency Graph，以降 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向グラフである．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．以下に，PDG の 2 つの依存関係について説明する．

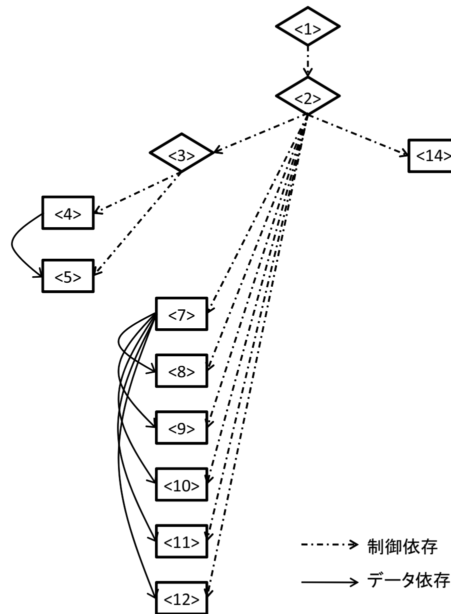
データ依存　文 s で変数 v を定義し，文 t で変数 v を参照しており，文 s から文 t への経路のうち，変数 v を再定義しないものがある場合，文 s から文 t にデータ依存があるという．

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．

図 1 は，簡単な PDG の例を表している．if 文の条件式からその内部に存在している文へ制御依存があり，また text などの変数の定義・参照を行なっている文の間にはデータ依存があるのがわかる．以降，本論文では，説明のための PDG としてこの例を用いる．

```
 1: String sample1(){
 2:   if(this.trueOrFalse()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text = new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     return text.toString();
13:   }else{
14:     return "";
15:   }
16: }
```

(a) ソースコード



(b) PDG

図 1　ソースコードと PDG の例

2.2 PDG を用いたコードクローン検出手順

PDG を用いたコードクローン検出手順[7] のうち，本論文で関連する STEP1 と STEP2 について記述する．

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れば，同じハッシュ値を持つ．

STEP2　プログラムスライシングを行い，同型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 $(r1, r2)$ であり，2 つのスライシングは同期して行なわれる．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，新たにたどった頂点のペア $(p1, p2)$ 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．

```
 1: void sample2(){
 2:   while(this.goOrStop()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text = new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     System.out.println(text.toString());
13:   }
14: }
```

図 2　比較対象ソースコード

- $(p1, p2)$ 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．これは無限ループを回避するための処理である．
- $(p1, p2)$ 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である．

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）がコードクローンのペアである．

2.3　PDG を用いたコードクローン検出の問題点

PDG を用いたコードクローン検出は，連続コードクローンの検出能力が低い．例えば，図 1(a) と図 2 からコードクローン検出を行なう場合を考える．図 1(a) の 2 行目から始まる if 文の内部の処理と，図 2 の 2 行目から始まる while 文の内部の処理は，12 行目の文を除いて重複している．しかし，既存手法を用いてコードクローン検出を行なった場合は，3 行目から 5 行目までが 1 つのコードクローンとして検出され，7 行目から 11 行目までは異なるコードクローンとして検出される．この理由は，3 行目から始まる if 文の処理と 7 行目から始まる変数 text に対する処理は，PDG 上では 2 行目の条件式の頂点を介して接続しているが，図 1(a) と図 2 では，条件式が等しくないため，プログラムスライシングがその頂点で止まるからである．

PDG を用いた検出では，計算コストが高いことも短所である．具体的には以下の 2 つの処理において高い計算コストを必要とする．

- 同型部分グラフの検出は，同じグループに属する頂点の全てのペアに対して行われる．あるグループに n 個の頂点を含む場合は，n(n-1)/2 個のペアに対して同型部分グラフの検出が行われる．実規模ソフトウェアでは，1 つのグループに何百，何千という頂点が含まれる場合があり，膨大なペア数に対して検出処理を行わなければならない．
- 頂点のペアから同型部分グラフを検出する事自体が，NP 完全な難しい問題である．

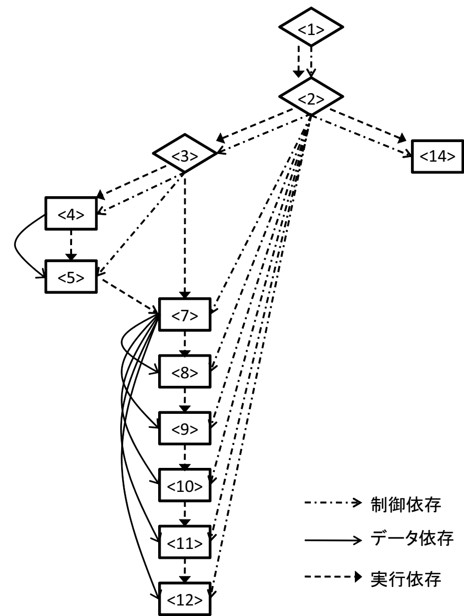


図 3　実行依存を付加した PDG

2.4　改良手法の概要

連続コードクローンの検出能力が低い問題を改善するため，著者らは PDG に新しい依存関係，実行依存を導入することを提案した[10]．実行依存はプログラム要素間の実行の順序を表す依存関係である．PDG 上の 2 つの頂点において，そのうちの 1 つの頂点（頂点 A とする）のプログラム要素が，他方の頂点のプログラム要素（頂点 B とする）が実行された直後に実行される可能性がある場合は，頂点 B から頂点 A に向かって実行依存辺が引かれる．実行依存を導入することにより，データ依存や制御依存がない場合でもソースコード上で隣接したプログラム要素をたどることが可能となり，連続コードクローンの検出能力が上がる．

図 3 は，図 1(b) に実行依存を導入した PDG を表している．実行依存を導入することにより，3 行目からの if 文の処理と 7 行目からの変数 text に対する処理が PDG 上で直接接続されるため，3 行目から 11 行目までが 1 つの連続コードクローンとして検出される．

3. 提 案 手 法

提案手法は，PDG の頂点数を削減することにより，コードクローン検出に必要な計算量を削減する．また，検出能力の向上も考慮に入れ，これまでの研究によりコードクローン検出に悪影響を与えるとされているソースコード上の繰り返し処理部分[9]，を削減の対象とする．ソースコード上の繰り返し処理とは，図 1(a) の 8 行目から 11 行目のような，文単位で同様の処理を繰り返し行っている部分を指す．このような処理部分から検出されるコードクローンは，人間が必要とするコードクローン情報となることはほとんどないことがこれまでに示されている[9]．例えば，8 行目の頂点と 10 行目の頂点から，フォワードスライスを用いて実行依存をたどることにより，8,9 行目と 10,11 行目が同一処理として検出されるが，そのようなコードクローンを人間が必要とすることはないだろう．8 行目から 11 行目までの処理をそれぞれ異なる頂点とするのではなく，まとめて 1 つの頂点として PDG を構築することにより，上記のような必要の無いコードクローンの検出処理を抑えつつ，(PDG の頂点数が少なくなるため）計算コストも削減も実現することができる．

具体的には，提案手法では，下記の全ての条件を満たす頂点群が 1 つの頂点として集約される．

条件 1　文 s から文 t へ，実行依存辺のみをたどることにより到達可能な経路 $s, u_1, u_2, \cdots, u_k, t$（以降，経路 R）が存在する．

条件 2　経路 R に存在する全ての頂点（頂点 s と t を含む）は，実行依存の入力辺と出力辺を 1 つずつ持つ（if 文や while 文などの条件式，およびそれらが終わった後の合流点を含まない）．

条件 3　経路 R 上に存在する全ての頂点は同一ハッシュ値を持つ．

条件 4　経路 R を包含するいかなる経路も，条件 2 と 3 を同時に満たさない．

経路 R に含まれる頂点群を 1 つに集約した頂点を m とする．以降，頂点 m を始点･終点とする，データ依存辺，制御依存辺，実行依存辺について説明する．

データ依存辺　頂点 p を終点とするデータ依存辺の始点となっている頂点の集合を $data_{to}(p)$，経路 R に含まれる頂点の集合を P とすると，集約された頂点 m へのデータ依存辺を持つ頂点の集合は以下の式により定義される．

$$data_{to}(m) = \bigcup_{p \in P} data_{to}(p) \cap \overline{P} \qquad (1)$$

同様に，頂点 p を始点とするデータ依存辺の終点となっている頂点の集合を $data_{from}(p)$ とすると，頂点 m からのデータ依存辺を持つ頂点の集合は以下の式により定義される．

$$data_{from}(m) = \bigcup_{p \in P} data_{from}(p) \cap \overline{P} \qquad (2)$$

制御依存辺　条件 2 より，P に含まれる全ての頂点は，同一の頂点からの制御依存辺を持つ．提案手法では，頂点 m も同一の頂点からの制御依存辺をもつとする．頂点 p を終点とする制御依存辺の始点となっている頂点の集合を $control_{to}(p)$ とすると，下記の式により定義される．

$$control_{to}(m) = control_{to}(s) \qquad (3)$$

また，条件 2 より，P に含まれる頂点を始点とする制御依存辺は存在しない．提案手法では，頂点 m を始点とする制御依存辺も存在しないとする．よって，頂点 m を始点とする制御依存辺の終点となっている頂点の集合 $control_{from}(m)$ は，下記の式により定義される．

$$control_{from}(m) = \emptyset \qquad (4)$$

実行依存辺　頂点 m を終点･始点とする実行依存辺をもつ頂点の集合 $execute_{to}(m)$，$execute_{from}(m)$ は，それぞれ以下の式で定義される．

$$execute_{to}(m) = execute_{to}(s) \qquad (5)$$

$$execute_{from}(m) = execute_{from}(t) \qquad (6)$$

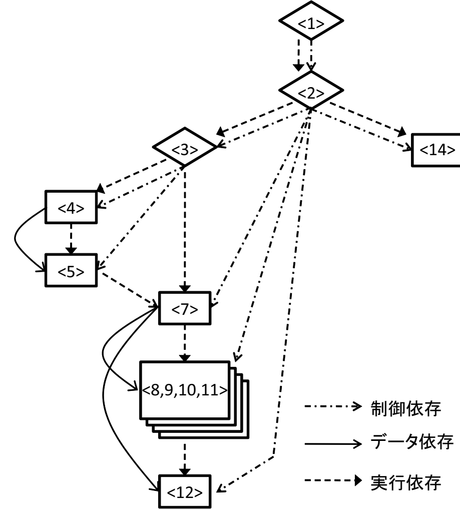


図 4　提案手法を用いて頂点集約を行なった PDG

図 3 の PDG に対して，提案手法の頂点集約を行なった PDG を図 4 に示す．この例では，図 1(a) に表すソースコードの 8 行目から 11 行目までの 4 つ頂点が，集約に必要な 4 つの条件をすべて満たしているため，1 つの頂点として集約が行われている．

提案手法の頂点集約を行うことにより，下記の 2 つの計算コストを削減することができる．

- 集約した頂点がプログラムスライシング上に現れる場合，頂点をたどる計算コストを抑えることができる．例えば，図 1(a) と図 2 において，各ソースコードの 7 行目の文をスライス基点としてスライシングを行なう場合，頂点を集約しない場合は，8 行目から 11 行目までの頂点をたどるのに，PDG 上で 4 ホップを必要とするが，頂点集約を行うことにより，1 ホップでたどることができる．
- 集約した頂点がスライス基点となる場合，集約しない場合に比べて頂点の総ペア数が減るため，行なうプログラムスライシングの数を削減することができる．例えば，頂点集約を行なわない場合は，図 1(a) と図 2 では，各ソースコードの 8 行目から 11 行目に存在する計 8 個の文はすべて同一ハッシュ値を持つため，${}_8C_2 = 28$ 個のペアについてスライシングが行なわれる．しかし，それらのペアのうちの多くは，ソースコード上で隣接した文であり，そのような頂点のペアを基点としてコードクローンを検出する必要はないと著者は考える．一方頂点集約を行なった場合は，各ソースコードの 8 行目から 11 行目の 4 つの頂点が 1 つに集約されるため，同一ハッシュ値を持つ頂点数は 2 つとなり，そのペアに対してのみプログラムスライシングが行われることになる．

4. 適用実験

提案手法をコードクローン検出ツール Scorpio[7),10)] に追加実装し，提案した頂点集約法の効果を調査するため，実験を行なった．この実験では，頂点集約が計算コストの削減量とコードクローンの検出結果に与えた影響について，下記の項目を用いて評価した．

計算コストの削減量　コードクローン検出に必要な計算コストを定量的に評価するため，頂点比較回数を調査した．頂点比較回数とは，検出処理の STEP2 で同型部分グラフを検出するために，2 つのプログラムスライシングを同時に実行し，たどった頂点のハッシュ値を比較した回数である．頂点比較回数が多いほど同型部分グラフの検出に必要なハッシュ値の比較回数が多いことになり，より高い計算コストが必要であったことがわかる．

検出結果に与えた影響　コードクローン検出に与えた影響を定量的に評価するため，この実験では，Bellon らが作成したコードクローンの正解集合（検出すべきコードクローンの集合）[2),4)] を用いた．コードクローンの検出結果と正解集合を用いて再現率を算出することにより，提案手法が検出結果に与えた影響を調査した．

この実験では，Bellon らが作成したコードクローンの正解集合を用いるため，正解集合の作成元となった 4 つのソフトウェア（表 1）を用いた．以降，これらのソフトウェアは表 1 で示す省略名でよぶ．

また，この実験では，データ依存辺と制御依存辺からなる従来の PDG を traditional，実行依存辺を追加した PDG を execution，提案手法を用いて頂点集約を行なった PDG を merged とよぶ．

4.1 コードクローンの正解集合

この実験では，2) で公開されているデータを検出すべきコードクローン群とした．このコードクローン群の情報は下記の手順で作成された．

(1) 文献 4) の著者（Stefan Bellon）が検出対象ソフトウェアを決め，6 人の研究者にコードクローン検出を依頼（Bellon はコードクローン検出ツールの開発者ではないため，中立的な立場で検出対象ソフトウェアを選んだといえる.）．依頼された各研究者は，コードクローン検出ツールの開発者である．

(2) 各研究者は，自身が開発した検出ツールを用いて，対象ソフトウェアから重複コードを検出．所定のフォーマットを用いて，検出した重複コードの位置情報を Bellon に送付．

(3) Bellon は各開発者から送られた重複コードの全ペア（325,935 個）のうちの 2%を乱数を用いて選択し，それらが本当にコードクローンであるかを手作業により判定．ツールが検出した重複コードがそのままコードクローンと判断される場合もあるが，Bellon が必要に応じて重複コードを加工して☆コードクローンと判断した場合もある．また，コードクローンではないと判定された重複コードもあった．この結果 4 つの対象ソフトウェアから 4,789 個のコードクローン

表 1　対象ソフトウェア

ソフトウェア	省略名	行数
netbeans-javadoc	netbeans	14,360
eclipse-ant	ant	34,744
eclipse-jdtcore	jdtcore	147,634
j2sdk1.4.0-javax-swing	swing	204,037

☆ ここでの加工とは，ツールが検出した重複コードから一部分を取り除くことや，重複コードの周辺コードもコードクローンに含めることを指す．

表 2　各 PDG の規模と頂点比較回数の調査結果

ソフトウェア	PDG	頂点数	辺数			頂点比較回数	
			データ依存	制御依存	実行依存	絶対値	増減率
netbeans	traditional	6,557	4,700	5,626	0	110,422,894	100.0%
	execution	6,557	4,700	5,626	6,144	103,408,483	93.6%
	merged	6,060	4,362	5,131	5,647	40,211,133	36.4%
ant	traditional	12,505	11,269	10,423	0	10,437,444	100.0%
	execution	12,505	11,269	10,423	12,379	11,396,350	109.2%
	merged	12,126	10,998	10,073	12,002	9,916,301	95.0%
jdtcore	traditional	77,493	91,617	64,701	0	1,424,793,100	100.0%
	execution	77,493	91,617	64,701	77,980	1,480,543,317	103.9%
	merged	73,885	88,443	61,263	74,595	1,048,732,585	73.6%
swing	traditional	82,824	75,560	68,310	0	528,247,797	100.0%
	execution	82,824	75,550	68,310	78,110	533,917,155	101.1%
	merged	78,783	73,026	64,370	74,050	409,070,104	77.4%

のペアが抽出された☆．

以降，この実験では，ツールが検出した重複コードを**クローン候補**，Bellon が抽出したコードクローンを**正解クローン**と呼ぶ．なお，正解クローンは，Bellon により以下の三種類に分類されている．

Type1　空白やタブを除いて表面上全く同一なコードクローンのペア，

Type2　変数名やメソッド名など，ユーザ定義の識別子が異なるコードクローンのペア，

Type3　Type1 や Type2 が許容する変更に加え，文単位や式単位で異なる部分が存在するコードクローンのペア．

Type1 と Type2 は連続コードクローンであり，Type3 は非連続コードクローンである．

4.2　評 価 基 準

この実験では，クローン候補が正解クローンかどうかを good 値と ok 値を用いて判断し，再現率を算出する．文献 4) と同様，good 値と ok 値の閾値として 0.7 を用いた． good 値と ok 値，および再現率の定義は付録に示す．なお，この実験で用いた正解クローンは，コードクローンの母集団（対象ソフトウェアに含まれるすべてのコードクローン）ではない．そのため，下記の点に留意されたい．

- 再現率の絶対値そのものは意味のある値ではない☆☆．この値はあくまでも複数の検出結果を相対的に比較するためのものである．
- 正解クローンの母集団が不明なため，適合率は算出できていない．

4.3　計算コストの削減量に関する調査結果

表 2 に，頂点比較回数の調査結果を示す．この表の増減率は，tradtional PDG を基準とした，各 PDG の頂点比較回数の割合を表している．execution PDG の場合，増減率は約 93∼109%であり，3 つのソフトウェアで頂点比較回数が増加していた．一方，netbeans では評点比較回数が減少してた．これは，execution PDG が tradtional PDG に比べて効率よくコードクローンを検出できる場合があることを示している．

merged PDG の場合，増減率は約 36∼95%であり，全てのソフトウェアで頂点比較回数が削減されていることが確認された．特に ant 以外の 3 つのソフトウェアは削減率が高い．頂点数や辺数は数%しか減少していないことを考慮すると，提案手法を使うことにより，検出のボトルネックとなっている部分を集約しているといえるだろう．

4.4　検出結果に与えた影響に関する調査

各 PDG を用いたコードクローンの検出結果を表 3 に示す．クローン候補数については，merged PDG が traditional PDG や execution PDG に比べて多いソフトウェアも少ないソフトウェアもあり，傾向はつかめなかった．検出された正解クローン数については，jdtcore においては，execution PDG の検出数が traditional PDG に比べて非常に大きく，実行依存辺導入の効果がもっとも顕著に現われている．他のソフトウェアについても，ほとんどの場合について微増している．swing の good 値を用いた評価の場合のみ，Type3 のコードクローンを 1 つ見落としてしまう場合があった．

merged PDG と execution PDG の正解クローンの検出数を比較すると，提案手法を用いることにより見落

☆ Bellon らの正解集合は Java 言語の 4 つのソフトウェアと C 言語の 4 つのソフトウェアに対して作成された．本論文ではこのうち，Java 言語に対して作成された 2,207 個のコードクローンのペアを用いた．

☆☆ 正解クローンの母集合を用いていないため，再現率の絶対値が検出能力を正しく表しているとは限らない．

表 3　検出された正解クローン数

ソフトウェア	正解クローン数			PDG	クローン候補数	検出された正解クローン数						再現率	
	Type1	Type2	Type3			Type1		Type2		Type3		good	ok
						good	ok	good	ok	good	ok		
netbeans	6	33	16	traditional	18,236	2	5	8	21	8	14	0.327	0.727
				execution	18,636	2	5	9	22	8	14	0.345	0.745
				merged	10,657	2	5	10	22	8	14	0.364	0.745
ant	4	24	2	traditional	1,279	0	3	6	10	1	1	0.233	0.467
				execution	1,553	0	3	6	10	1	1	0.233	0.467
				merged	1,378	0	3	6	10	1	1	0.233	0.467
jdtcore	120	866	359	traditional	150,750	31	98	195	483	84	165	0.230	0.555
				execution	248,488	33	101	314	619	130	214	0.355	0.694
				merged	268,266	36	102	322	624	130	214	0.363	0.699
swing	145	595	37	traditional	63,558	31	54	71	359	13	22	0.148	0.560
				execution	67,636	31	55	72	360	12	22	0.148	0.560
				merged	31,114	31	56	72	360	13	23	0.149	0.565

としてしまう場合は皆無であることがわかる．また，jdtcore 以外の 3 つのソフトウェアにおいても正解クローンの検出数が微増している．これは，連続した同一処理部分全体をまとめてコードクローンとすることにより，正解クローンとの good 値，ok 値が閾値を上回った（人間がコードクローンと判断する部分に近づいた）場合があったことを示しており，提案手法は，検出結果の再現率向上にも役立つといえる．

4.5　実験結果の妥当性について留意すべき点

本実験では，Bellon らが作成したコードクローンの正解集合を用いて提案手法の有効性について調査を行なった．しかし，この正解集合は対象ソフトウェアに含まれるすべてのコードクローンではない．そのため，全ての正解クローンを対象にして同様の実験を行なった場合は，提案手法を用いることにより，検出されなくなってしまう正解クローンが現われる可能性がある．

また，全ての正解クローンが得られていないため，提案手法を用いることによる適合率の変化については調査ができていない．コードクローン検出手法の適合率とは，検出結果のどの程度が正解クローンなのかを表す指標であり，検出手法の能力を評価する上で非常に重要な尺度である．再現率と適合率の両方の変化を見ることにより初めて公正な評価となる．

5.　お わ り に

本論文では，プログラム依存グラフを用いたコードクローン検出に必要な計算コストの削減を目的とした頂点集約手法を提案した．提案手法は，著者らがこれまでに提案している実行依存つきプログラム依存グラフ[10] を拡張するものである．提案手法はプログラム依存グラフの実行依存辺に着目し，辺の始点と終点が一定の条件を満たす場合は 1 つの頂点として集約を行なう．頂点数が削減されるため，プログラムスライシングにおいて頂点をたどるコスト，またプログラムスライシングを行なう回数がそれぞれ削減できる．提案手法を検出ツール Scorpio[7),10)] に追加実装し，オープンソースのソフトウェアに対して実験を行なった．この実験により，必要なコードクローン情報をまったく減らすこと無く，計算コストを最大で約 36%にまで削減できることが確認された．今後は，実行依存辺だけでなく，他の要素に着目しさらに計算コストを削減する手法を考案する予定である．また，近年，プログラム依存グラフ以外の技術を用いた非連続コードクローンを検出する手法も提案されている[6),14)]．これらの手法と本手法を用いた比較も行う予定である．

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．

参　考　文　献

1) : Clone Detection Literature, http://www.cis.uab.edu/tairasr/clones/literature/.
2) : Detection of Software Clones, http://bauhaus-stuttgart.de/clones/.
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 En-*

gineering, 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) Gabel, M., Jiang, L. and Su, Z.: Scalable Detection of Semantic Clones, *Proc. of the 30th International Conference on Software Engineering*, pp. 321–330 (2008).

7) 肥後芳樹, 楠本真二: 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, ソフトウェアエンジニアリング最前線 2009（ソフトウェアエンジニアリングシンポジウム 2009 予稿集）, pp. 97–104 (2009).

8) 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).

9) Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and Implementation for Investigating Code Clones in a Software System, *Information and Software Technology*, Vol. 49, No. 9-10, pp. 985–998 (2007).

10) Higo, Y. and Kusumoto, S.: Significant and Scalable Code Clone Detection with Program Dependency Graph, *Proc. of the 16th Working Conference on Reverse Engineering*, pp. 315–316 (2009).

11) Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proc. of 2004 International Symposium on Empirical Software Engineering*, pp. 83–92 (2004).

12) 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).

13) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. of the 8th Working Conference on Reverse Engineering*, pp.301–309 (2001).

14) Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *Proc. of the 16th Intenational Conference on Program Comprehension*, pp. 172–181 (2008).

15) Roy, C. K. and Cordy, J. R.: A Survey on Software Clone Detection Research, Technical report, School of Computing, Queen's University (2007).

付　録

定義 1　2 つのコード片（f_1 と f_2）の重なりの程度を以下の式で定義する．なお，$lines(f)$ はコード片 f に含まれる行の集合を表す．

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|} \quad (7)$$

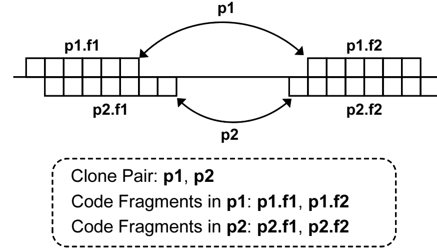


図 5　good 値と ok 値の算出例

定義 2　あるコード片（f_1）が他のコード片（f_2）に含まれている程度を以下の式で定義する．

$$contain(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|} \quad (8)$$

定義 3　2 つのクローンペア（p_1 と p_2）の good 値は以下の式で定義される．

$$good(p_1, p_2) = min(overlap(p_1.f_1, p_2.f_1), overlap(p_1.f_2, p_2.f_2)) \quad (9)$$

図 5 には 2 つのクローンペア（p_1 と p_2）が存在している．この場合，good 値は，

$$good(p_1, p_2) = min(\frac{5}{8}, \frac{6}{8}) = \frac{5}{8}$$

となる．good 値の閾値が 0.7 の場合，$\frac{5}{8} \leq 0.7$ であるため，2 つのクローンペアは一致しない．

定義 4　2 つのクローンペア（p_1 と p_2）の ok 値は以下の式で定義される．

$$\begin{aligned} ok(p_1, p_2) = \\ & min(max(contain(p_1.f_1, p_2.f_1), \\ & \quad contain(p_2.f_1, p_1.f_1)), \\ & \quad max(contain(p_1.f_2, p_2.f_2), \\ & \quad contain(p_2.f_2, p_1.f_2))) \end{aligned} \quad (10)$$

図 5 の例を用いた場合，ok 値は，

$$ok(p_1, p_2) = min(max(\frac{5}{6}, \frac{5}{7}), max(\frac{6}{6}, \frac{6}{8})) = \frac{5}{6}$$

となる．good 値の閾値が 0.7 の場合，$0.7 \leq \frac{5}{6}$ であるため，2 つのクローンペアは一致する．

定義 5　コードクローンの正解集合を S_{refs}，ok 値を用いてコードクローン検出結果 R から抽出した正解クローンの集合を $S_{ok}(R)$，good 値を用いた正解集合を $S_{good}(R)$ としたとき，R の再現率はそれぞれ以下の式で定義される．

$$Recall_{ok}(R) = \frac{|S_{ok}(R)|}{|S_{refs}|} \quad (11)$$

$$Recall_{good}(R) = \frac{|S_{good}(R)|}{|S_{refs}|} \quad (12)$$