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.

コードクローン検出のためのソースコードに対する前処理

佐飛　祐介†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1-5
E-mail: †{y-sabi,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　一般的に，コードクローンはソフトウェアの保守性を低下させる原因であると考えられており，今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．しかし，既存のコードクローン検出手法が採用しているほとんどのアルゴリズムでは，プログラム文の順序が異なるコード片をコードクローンとして検出できない．そのようなコード片に対しては，文の順序を統一することで既存の検出手法においても検出可能になると考えられるが，文の順序統一を行っている既存手法は存在しない．そこで本研究では，文の順序を統一することでコードクローン検出の結果がどのように変化するのかを調査した．文の順序統一には，本研究のために拡張を行ったプログラム依存グラフを用いた．調査では，文の順序統一前後におけるコードクローン検出の結果を比較した．その結果，文の順序統一によってコードクローンの検出数はほとんど変化しないことがわかった．また，文の順序統一によって新しく検出されたコードクローンは存在せず，検出されなくなったクローンはソフトウェア保守の観点から検出されるべきクローンであった．このことより，順序入れ替わりクローンを検出するために文の順序統一を行う有効性は少ないと考えられる．
キーワード　コードクローン，プログラム依存グラフ，順序入れ替わりコードクローン

1. ま え が き

コードクローン（以降，クローンと表記する）とは，ソースコード中に存在する互いに類似もしくは一致しているコード片のことである．クローンの主な発生要因はコピーアンドペーストである [1]．一般的に，クローンはソフトウェアの保守性を低下させる原因であると考えられている [2]．例えば，あるコード片にバグが存在していた場合，そのバグを修正するだけでなく，そのコード片のクローンに対しても同様の修正を検討する必要がある．そのため，クローンがソフトウェア中のどこに存在しているか，またどの程度存在しているかを把握するのはソフトウェアの保守において重要であり，今までに多くのクローン検出手法が提案されている [3]．

既存のクローン検出手法には Parameterized-Matching アルゴリズムを用いたもの [4] や接尾辞配列アルゴリズムを用いたもの [5] など，様々な種類のアルゴリズムを用いたものが存在し，それらの種類によって検出されるクローンは異なる．

既存のクローン検出手法が用いているほとんどのアルゴリズムでは，順序入れ替わりクローンは検出の対象範囲外となっている．順序入れ替わりクローンとはプログラム文の順序が異なるクローンである．図 1 にその例を示す．図 1 では，クローン片 A の 81–88 行目とクローン片 B の 96–103 行目が順序入れ替わりクローンとなっている．クローン片 A ではクローン片の最終行である 88 行目にて変数が宣言されているが，クローン片 B では同様の変数がクローン片の 1 行目である 96 行目で宣言されており，プログラム文の順序が異なることがわかる．プログラム依存グラフ（Program Dependency Graph，以降 PDG と表記する）を用いている Komondoor らの手法 [6] は順序入れ替わりクローンを検出可能である．しかし，彼らの手法ではクローン検出に長い時間を要する．また，彼らの手法は文がソースコード上で連続していないクローンを検出可能である．そのようなクローンの中には，開発者にとって有益でないクローンが多く含まれる [7]．

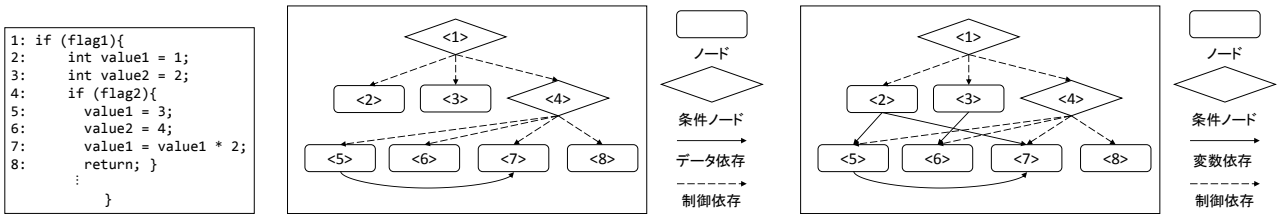
筆者らはプログラム文の順序を一定の規則に則って統一することで，既存のクローン検出手法でも順序入れ替わりクローンを検出することができるようになると考えた．しかし，今までにプログラム文の順序を統一するアルゴリズムを採用しているクローン検出手法は存在しない．

そこで，本研究では文の順序統一を行うことにより，クローン検出結果がどのように変化するかを調査した．この調査では，以下の 2 つの調査項目に回答する．

RQ-1　文の順序統一によって，クローンの検出数はどのように変化するのか．

RQ-2　文の順序統一によって，新しく検出されたもしくは検出されなくなったクローンが存在する場合，それらのクローンはソフトウェア保守の観点から検出されるべきクローンか．

本研究では，文の順序統一に PDG を用いた．文の順序統一では，同じ変数を使用している文の順序を保ちながら，文の順序を一定の規則に則って並べ替える．ここで，PDG において依存関係が存在する文を順序を保つ文としている．従来の PDG で用いられているデータ依存を用いた場合，同じ変数を参照している文間に依存関係が存在しないため，そのような文の順序が入れ替わってしまう可能性がある．そこで本研究では PDG に対して拡張を行った．この拡張を行った PDG は 2 種類の依



(a) ソースコード (b) PDG (c) 拡張 PDG

図 2: PDG の例

```
81:    Date startDate = null;
82:    if(sinceDays > 0) {
83:        Calendar cal = Calendar.getInstance();
84:        cal.setTime(new Date());
85:        cal.add(Calendar.DATE, -1 * sinceDays);
86:        startDate = cal.getTime();
87:   }
88:   List<SubscriptionEntry> results = new ……
```

クローン片A

```
96:   List<WeblogEntryWrapper> results = new ……
97:   Date startDate = null;
98:   if(sinceDays > 0) {
99:       Calendar cal = Calendar.getInstance();
100:      cal.setTime(new Date());
101:      cal.add(Calendar.DATE, -1 * sinceDays);
102:      startDate = cal.getTime();
103:  }
```

クローン片B

図 1: 順序入れ替わりクローンの例

存関係をもつ．1 つ目の依存関係は，従来の PDG と同じ定義をもつ制御依存である．2 つ目の依存関係は，2 つの文に同じ変数を使用している場合に存在する変数依存である．制御依存は同じブロックにあるソースコードを取得するために使用され，変数依存は同じ変数を使用している文の順序を保つために使用される．

上記の 2 つの調査項目に回答するため，8 つのオープンソースプロジェクトに対して実験を行った．調査項目への回答は以下のとおりである．

- 文の順序統一によって，新しく検出されたクローンの数と検出されなくなったクローンの数はほぼ同数であり，クローンの検出数はほとんど変化しなかった．
- 文の順序統一を行うことでクローン片内が並べ替えられ，新しく検出されたクローンは存在しなかった．クローン片内が並べ替えられ検出されなくなったクローンのほとんどは，同じ開発者によって同じコミットにて変更されたものであり，コードの記述からもコピーアンドペーストが原因で発生したクローンと考えられる．検出されなくなったクローンはコピーアンドペーストが原因で発生したクローンであるため，1 つのクローン片に変更を加えたとき，他のクローン片にも同様の変更が必要となる可能性が高い．このことより，検出されなくなったクローンはソフトウェアの保守性を低下させるようなクローンであり，検出されるべきクローンであると考えられる．

このことより，順序入れ替わりクローンを検出するために文の順序統一を行う有効性は少ないと考えられる．

```
1: StringBuilder result = new StringBuilder();
2: public String getResponse() …… {
3:      result.setLength(0);
4:      String line = reader.readLine();
5:      if (line != null) {
6:          result.append(line.substring(0, 3));
7:          result.append(" "); }
```

並べ替えてはいけない2つの文

図 3: 順序を並べ替えていけない文の例

2. 定　　義

この章では，本論文で使用する語句，技術を説明する．

2.1 クローンのタイプ

クローンには 3 つのタイプが存在する．それぞれの定義を以下に示す．

Type-1: 空白やタブの有無，括弧の位置などのコーディングスタイルに依存する箇所を除いて，完全に一致するクローン．

Type-2: 変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるクローン．

Type-3: Type-2 における変更に加えて，文の挿入や削除，変更が行われたクローン．

クローンとして検出されたコード片の組のことを “クローンペア” という．

2.2 プログラム依存グラフ (PDG)

PDG で用いられている 2 種類の依存関係を以下に示す [8]．

データ依存

- 変数 v が文 s で定義されている．
- 変数 v が文 t で参照されている．
- s から t の間に v を再定義しない経路が存在する．

以上の 3 つの条件を満たすとき，文 s と文 t 間にデータ依存が存在する．

制御依存

- 文 s が条件文である．
- 文 t は文 s の後に実行される可能性がある．
- s の実行結果によって t が実行されるかが決まる．

以上の 3 つの条件を満たすとき，文 s と文 t 間に制御依存が存在する．

図 2(b) に簡単な PDG の例を示す．変数 $value1$ が定義されている 5 行目から同じ変数が参照されている 7 行目へとデータ依

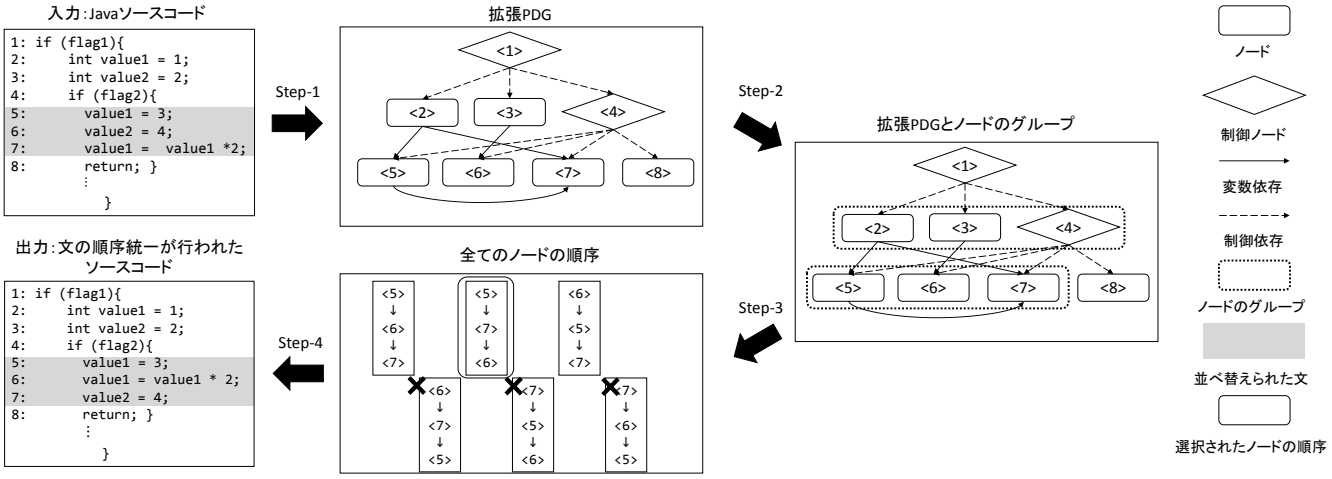


図 4: 文の順序統一の例

存が存在していることがわかる．また，if 文の条件文からその内部に存在する文に制御依存が存在していることがわかる．

2.3 順序入れ替わりクローン

順序入れ替わりクローンとは，図 1 に示すようなプログラム文の順序が異なるクローンのことである．

Komondoor らは順序入れ替わりクローンを検出可能なクローン検出手法を提案している [6]．Komondoor らの手法では，検出対象となるソースコードを PDG に変換し，同型部分グラフを検出することで順序入れ替わりクローンの検出を可能としている．しかし，同型部分グラフを検出する処理は長い時間を要する処理であり，そのためクローンの検出に長い時間を要する．また，Komondoor らの手法では文がソースコード上で連続していないクローンが検出できる．そのようなクローンの中には，開発者にとって有益でないクローンが多く含まれる [7]．

2.4 拡張 PDG

本研究では従来の PDG を拡張した 2 つの依存関係を持つ PDG を用いている．以降，本研究で用いる拡張を行った PDG のことを “拡張 PDG” と表記する．1 つ目の依存関係は従来の PDG と同じ定義を持つ制御依存である．2 つ目の依存関係は変数依存である．変数依存の定義を以下に示す．

変数依存

- 変数 v が文 s で定義もしくは参照されている．
- 変数 v が文 t で定義もしくは参照されている．
- s から t の間に v を再定義しない経路が存在する．

以上の 3 つの条件を満たすとき，文 s と文 t 間に変数依存が存在する．

図 2(c) に拡張 PDG の例を示す．変数 $value1$ が定義されている 2 行目から同じ変数が再定義されている 5 行目へと変数依存が存在することがわかる．

変数依存を用いた理由を説明する．本研究では，同じ変数を使用している文の順序を保ちながら，文の順序を統一する．PDG において依存関係が存在する文を順序を保つ文としている．そのため，同じ変数を使用している文間に依存関係が存在する変数依存を用いている．

例えば，図 3 のソースコードの 6，7 行目には 2 つのメソッド呼び出し文 “append()” が存在する．これらの文が異なる順序で実行された場合，変数が指しているオブジェクトの中身は異なるものとなる．そのため，この 2 文の順序を並べ替えてはいけない．従来の PDG で用いられているデータ依存を用いた場合，6，7 行目間に依存関係が存在せず，順序が保たれない可能性がある．一方，変数依存を使用した場合，6，7 行目間に依存関係が存在するため，2 文の順序は保たれる．このように，同じ変数を用いた文の順序を保つために変数依存を用いている．

3. 文の順序統一

この章では，文の順序統一の方法について説明する．入力と出力は以下のとおりである．

- 入力…Java ソースコード
- 出力…文の順序統一が行われたソースコード

文の順序統一は以下の 4 つの Step で行われる．

Step-1: 入力されたソースコードから拡張 PDG を生成．
Step-2: 同じブロックに存在するノードのグループを取得．
Step-3: 依存関係を変えないノードの並び順を選択．
Step-4: 拡張 PDG からソースコードを復元．

以下，それぞれの Step について図 4 の例を用いて説明する．図 4 では，図 2(a) のソースコードを入力として与えている．

3.1 Step-1

この Step では，入力ソースコードから拡張 PDG を生成する．

3.2 Step-2

この Step では，ノードで構成されるグループを取得する．それぞれのグループは同じノードからの制御依存を持つノードで構成される．ここで，グループ内のノードはソースコードにおいて同じブロックに存在するプログラム文に対応する．この Step ではブロックの最後に return 文，break 文，continue 文がある場合，それと対応するノードをグループから除外するというヒューリスティックを用いた．

図 4 の例では，ノード <1> はノード <2>，<3>，<4> への制御依存辺を持っており，またノード <4> はノード <5>，<6>，<7>，<8> への制御依存辺を持っている．そのため，

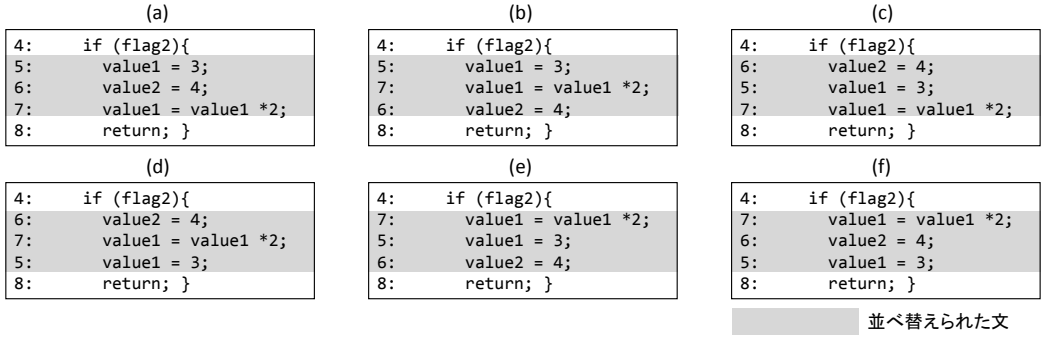


図 5: ノードの順序をそれぞれソースコードで表したもの

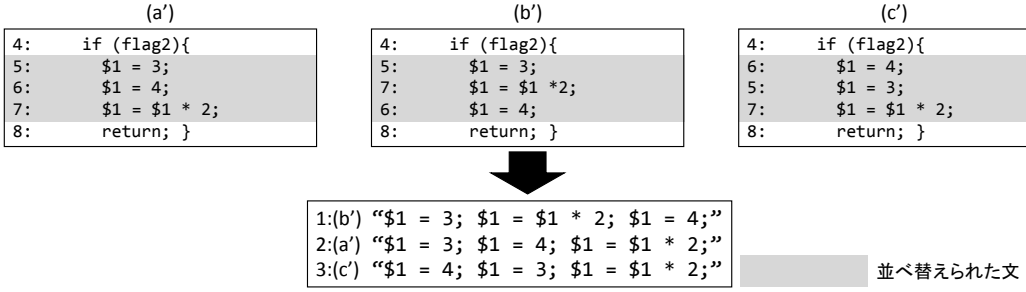


図 6: 変数名の正規化とアルファベット順のソート

ノード <2>，<3>，<4> で構成されるグループとノード <5>，<6>，<7>，<8> で構成されるグループがそれぞれ存在する．次に，ノード <8> は return 文であるため，2 つ目のグループから除外する．ここで，条件ノードはそのブロックに含まれる全てのノードとみなす．つまり，ノード <4> は，ノード <4>，<5>，<6>，<7>，<8> とみなす．以上より，図 4 の例ではノード <2>，<3>，<4> からなるグループと，ノード <5>，<6>，<7> からなるグループが取得できる．

3.3 Step-3

この Step では，Step-2 で取得したグループそれぞれに対して，依存辺の始点と終点となっているノードの組の順序が並べ替えられていないノードの順序を選択する．はじめに，ノードの数を n としたとき，$n!$ 通りのノードの順序を候補として列挙する．次に，依存辺の始点と終点となっているノードの組の順序が 1 つでも並べ替えられている順序を候補から除外する．その次に，候補として残っている順序に対して，ノードに含まれる変数名の正規化を行う．最後に，変数名の正規化を行った文をもとにして，残っている順序をアルファベット順に並べ替え，最初のものを選択する．

図 5 と図 6 の例を用いて説明する．図 5 はノード <5>，<6>，<7> で構成されるグループについて，$3! = 6$ 通りのノードの順序をソースコードで表している．それぞれの順序を (a)–(f) とする．この図において，行番号はノードの ID と対応している．図 4 に示すとおり，ノード <5> とノード <7> の間には変数依存が存在する．(d)，(e)，(f) ではそれらのノードの順序が並べ変わっているため，候補から除外する．次に，残っている順序 ((a)，(b)，(c)) に対して変数名の正規化を行う．正規化を行ったソースコード ((a')，(b')，(c')) を図 6 に示す．それら 3 つの順序をアルファベット順に並べた場合，(b') が最初となる．そのため，(b) の順序を並べ替え後の順序として選択する．つまり，ノード <5>，<6>，<7> はノード <5>，<7>，<6> という順序に並べ替えられる．また，ノード <2>，<3>，<4> で構成されるグループについては，並べ替えは行われない．

3.4 Step-4

ここまでで，それぞれのグループに対してノードの順序が得られた．この Step では，Step-3 の結果をもとにして拡張 PDG からソースコードを生成する．

4. 実　　験

この章では，本研究で行った実験について説明する．

4.1 実験手順

以下 4 つの Step で実験を行った．

Step-A: ソースコードに対して文の順序統一を行う．

Step-B: 元のソースコードと文の順序統一を行ったソースコードそれぞれに対してクローンを検出する．

Step-C: 文の順序統一によって新しく検出されたもしくは検出されなくなったクローンを抽出する．

Step-D: 抽出されたクローンを目視で分類する．

Step-A については 3. で説明しているため，ここでは省略する．以下，Step-B，C，D について説明する．

Step-B では既存のクローン検出手法を用いて，元のソースコードと文の順序統一を行ったソースコードに対してクローンの検出を行う．検出手法には CCFinder [9] を用いた．CCFinder は短時間で Type-2 までのクローンを検出できる手法である．CCFinder のパラメータである最小一致トークン数は 50，最小トークン種類数は 12 とした，これはデフォルトの設定である．

Step-C では，文の順序統一によって新しく検出されたもしくは検出されなくなったクローンを抽出する．これらのクローンを抽出するために，両方の検出結果に含まれているクローンのマッピングを行った．マッピングが行われなかったクローン

のうち，文の順序統一後に検出されたクローンを新しく検出されたクローン，文の順序統一前に検出されたクローンを検出されなくなったクローンとみなしている．マッピングには good 値 [10] を利用した．2 つのクローンペア p_1 と p_2 間の good 値は以下の式で計算される．$p_1.f_1$，$p_1.f_2$，$p_2.f_1$，$p_2.f_2$ はそれぞれのクローンのコード片を表す．

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

2 つのコード片 f_1, f_2 間の値，$overlap$ は以下の式で計算される．$lines(f)$ はコード片 f に含まれる行の集合を表す．

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

2 つのクローンペア間の good 値が閾値を超えた場合，それら 2 つのクローンペア間にマッピングが取られる．ここでは文献 [10] と同様に，閾値として 0.7 を用いた．

本来，good 値は 1 つのソースコードに対してクローン検出手法を用いた際，検出されたクローンが正解集合と同一であるかを判断するための評価方法である．ここでは，複数のソースコードに対して 1 つのクローン検出手法を用い，両方の検出結果に含まれるクローン間にマッピングを取るために使用しており，本来の使用方法とは異なる．このため，本来マッピングが行われるべきクローンペア間にマッピングが行われない可能性がある．しかし，そのような場合でも目視確認の対象が増えるのみで，新しく検出されたもしくは検出されなくなったクローンが抽出されなくなることはない．

Step-D では，Step-C で抽出されたクローンペアに対して目視で分類を行った．目視での分類は本論文の著者 1 名（大学院生）が行った．目視分類では，クローン片内において文の並べ替えが行われているかどうかで分類を行った．これは，クローン片内の文の順序は変化していないにも関わらず，文の順序統一によって行番号がずれたためマッピングが取られなかったクローンと，そうでないクローンを区別するためである．

4.2 実験対象

Apache リポジトリより 2016/9/13 時点で最新の Java プロジェクト 79 個をチェックアウトし，プロジェクトサイズが 1MB 以上のものを 8 個選択した (注1)．表 1 に詳細を示す．

表 1: 実験対象プロジェクト

プロジェクト名	Java ファイル数	LOC
Any23	396	46,957
BVal	350	37,520
Flume	314	48,986
Giraph	252	60,814
Gora	435	31,162
Qpid	118	27,037
Subversion	170	47,917
Wookie	295	48,537

(注1)：プロジェクト内の Java ファイルに対して “generated” の文字列で検索し，該当したファイルから自動生成ファイルとみられるものを目視で除去した．

4.3 実験結果

表 2，3 に実験結果を示す．5. で結果に対する議論と調査項目への回答を行う．表 2 での “変化クローンペア” の欄はマッピングが取られなかったクローンペアの数を表しており，“新規検出” の欄は新しく検出されたクローンペアの数，“消失” の欄は検出されなくなったクローンペアの数を表している．

5. 調査項目への回答

この章では，実験結果を用い調査項目に対して回答する．

5.1 RQ-1

表 2 より，8 つ中 3 つのプロジェクトで検出されたクローンペア数は減少している．しかし，減少したクローンペアの数は多くても 2 組であり，ほとんど変化していない．

RQ-1 に対する回答：文の順序統一によって，新しく検出されたクローンの数と検出されなくなったクローンの数はほぼ同数であり，クローンの検出数はほとんど変化しなかった．

以上より，クローンの検出数を増やす観点において，文の順序統一の有効性は低いと考えられる．

5.2 RQ-2

表 3 に，新しく検出されたもしくは検出されなくなったクローンペアに対する目視分類の結果を示す．

新しく検出されたクローンペアのうち，クローン片内において並べ替えが行われているものは存在しなかった．検出されなくなったクローンペアのうち，クローン片内で並べ替えが行われているもの 4 組に対しては，その原因を調査した．これらのクローンペアが検出されなくなった原因は，片方のクローン片において，クローン片外の文も含めて並べ替えが行われ，クローン片外の文がクローン片内に並べ替えられたからであった．プロジェクトのリポジトリを調査したところ，これらの検出されなくなったクローンのうち 3 組は，同じ開発者によって同じコミットにて変更されたものであり，コードの記述からもコピーアンドペーストで発生したクローンと考えられる．残りの 1 組は，異なる開発者によるコミットであり，コピーアンドペーストで発生したクローンではないと考えられる．コピーアンドペーストが原因で発生したクローンは，1 つのクローン片に変更を加えたとき，他のクローン片にも同様の変更が必要となる可能性が高い．このことより，文の順序統一により検出されなくなったクローンはソフトウェアの保守性を低下させるク

表 2: クローン検出結果

	クローンペア数		変化クローンペア数	
	順序統一前	順序統一後	新規検出	消失
Any23	389	389	0	0
BVal	199	199	2	2
Flume	1,051	1,051	0	0
Giraph	582	582	62	62
Gora	703	701	0	2
Qpid	426	425	132	133
Subversion	708	707	0	1
Wookie	6,090	6,090	0	0

表 3: 新しく検出されたもしくは検出されなくなったクローンペア

	新しく検出されたクローンペア		合計	検出されなくなったクローンペア		合計
	クローン片内の並べ替え無	クローン片内の並べ替え有		クローン片内の並べ替え無	クローン片内の並べ替え有	
Any23	0	0	0	0	0	0
BVal	2	0	2	2	0	2
Flume	0	0	0	0	0	0
Giraph	62	0	62	62	0	62
Gora	0	0	0	0	2	2
Qpid	132	0	132	132	1	133
Subversion	0	0	0	0	1	1
Wookie	0	0	0	0	0	0
全体	196	0	196	196	4	200

ローンであり，検出されるべきクローンであると考えられる．

また，クローン片内の並べ替えが行われていなかったクローンペアに関しては目視によるマッピングを行い，新しく検出されたクローンペアのうちクローン片内の並べ替えがなかったものと，検出されなくなったクローンペアのうちクローン片内の並べ替えがなかったものが 1 対 1 で対応することを確認した．

このことより，RQ-2 に対する回答は以下のとおりである．

RQ-2 に対する回答：文の順序統一を行うことでクローン片内が並べ替えられ，新しく検出されたクローンは存在しなかった．クローン片内が並べ替えられ検出されなくなったクローンのほとんどは，コピーアンドペーストが原因で発生したクローンと考えられる．これらのクローンはソフトウェアの保守性を低下させるクローンであり，検出されるべきクローンであった．

文の順序統一によって全ての順序入れ替わりクローンが検出できるようになるわけではないが，新しく検出されたクローンの中に順序入れ替わりクローンは存在しなかった．このことより，今後のクローン検出手法において，順序入れ替わりクローンを検出可能にする利点は少ないと考えられる．

6. 妥当性への脅威

対象プロジェクト：本研究では Apache リポジトリの 8 つのプロジェクトを対象としている．しかし，他のプロジェクトに対して実験を行った場合，異なる結果が得られる可能性がある．

クローン検出手法：本研究では CCFinder を検出手法として用いた．異なる手法を用いた場合，検出されるクローンが変化するため，得られる結果が異なる可能性がある．しかし，Type-3 を検出可能な手法を用いた場合，文の順序統一を行わなくても順序入れ替わりクローンの一部がクローンとして検出される可能性がある．この場合，文の順序統一によって新しく検出された順序入れ替わりクローンの数が実際よりも少なくなってしまう可能性がある．そのため，本研究では Type-2 までのクローン検出が可能な CCFinder を用いた．

7. あ と が き

本研究では，拡張を行った PDG を用いて文の順序統一を行い，クローン検出結果がどのように変化するかを調査した．調査では，文の順序統一前後のクローン検出結果を比較した．その結果，ほとんどのプロジェクトでクローンの検出数は変化しなかった．また，新しく検出されたもしくは検出されなくなったクローンの分類を行った．その結果，文の順序統一によって新しく検出されるクローンは存在せず，検出されなくなったクローンはソフトウェア保守の観点から検出されるべきクローンであった．このことより，順序入れ替わりクローンを検出するために文の順序統一を行う利点は少ないと考えられる．

今後の課題としては，大規模なプロジェクトに対する追加実験や，他のクローン検出ツールを用いた追加実験があげられる．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：JP25220003）の助成を得て行われた．

文　献

[1] C.K. Roy, J.R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming, vol.74, no.7, pp.470–495, 2009.

[2] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, “Can I Clone This Piece of Code Here?,” Proc. of the 27th International Conference on Automated Software Engineering, pp.170–179, 2012.

[3] 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.91-D-1，no.6，pp.1465–1481，2008.

[4] 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.

[5] H.A. Basit and S. Jarzabek, “Detecting Higher-level Similarity Patterns in Programs,” Proc. of the 10th European Software Engineering Conference, pp.156–165, 2005.

[6] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication in Source Code,” Proc. of the 8th International Symposium on Static Analysis, pp.40–56, 2001.

[7] 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法の改善と評価，” 情報処理学会論文誌，vol.51，no.12，pp.2149–2168，2010.

[8] J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The Program Dependence Graph and Its Use in Optimization,” ACM Transactions on Programming Languages and Systems, vol.9, no.3, pp.319–349, 1987.

[9] 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, vol.28, no.7, pp.654–670, 2002.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” IEEE Transactions on Software Engineering, vol.33, no.9, pp.577–591, 2007.