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.

ソースコード中の変数間のデータ依存関係を用いたコミットの分割

切貫　弘之† 　堀田　圭佑† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{h-kirink,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　1 つの修正に対して 1 つのコミットが行われるべきであるが，開発者はしばしば、複数の修正を含んだコミットを行ってしまう．このようなコミットは，リポジトリ分析の精度を下げることが知られている．この研究では，まず，ソースコード中の変数間のデータ依存関係を用いて，修正された箇所をいくつかのグループにまとめる．そして，1 つのグループを 1 つの修正とみなすことで，そのようなコミットを解消できることを示す．

キーワード　プログラム依存グラフ，コミット，版管理システム

partitioning commits using data dependency among variables in source code

Hiroyuki KIRINUKI†, Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871, Japan
E-mail: †{h-kirink,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Although developers should do one commit for every task, they often commit multiple tasks at the same time. It has been found that those commits compromise results of mining software repositories. In this research, we propose a technique to split multiple modifications included in a single commit. The technique utilizes dependencies among variables in source code. We confirmed that some commits were split appropriately by using the proposed technique.

Key words　program dependency graph, commit, version control system

1. ま え が き

リポジトリとは．ソフトウェア開発に関わる様々な情報を蓄積したデータ群のことである．ソフトウェア開発を行う上で有益な情報を得るためにリポジトリを分析することはよく行われており，それに関する研究も盛んである．[1]〜[5]．リポジトリ分析では，複数の修正を含んだ大規模なコミットを事前に分析対象から除外することが多い [6]．なぜなら，そのような大規模なコミットはリポジトリ分析の結果に悪影響を及ぼすためである．そのことは既存研究により明らかになっている [7]．文献 [7] によれば，Herzig らが分析対象としたソフトウェアのバグ修正の最大 20%は大規模なコミットの中に含まれており，それによって少なくとも 16.6%のファイルが誤ってバグレポートと関連付けられていた．しかし，大規模なコミットを分析対象から除外することで，除外されたコミットに含まれていた重要な情報が分析結果に反映されない恐れがある．

文献 [7] ではこのような大規模なコミットを解消する手法が提案されている，Herzig らの手法では，メソッドの呼び出し関係やデータ依存などを考慮して 1 つの大きな修正を複数の修正のまとまりにしている．しかし，Herzig らの手法はメソッド定義とメソッド呼出の追加・削除以外の修正を考慮していないので，適切にコミットを分割するまでには至っていない．さらにコミットを分割する個数を指定する必要があるため，あらかじめいくつの修正が一度にコミットされているのかを知っている必要がある．

本研究では，ソースコードに加えられた修正を変数間のデータ依存関係を用いて複数の修正に分割し，それに基づいてコミットを再構築する手法を提案する．提案手法をオープンソースソフトウェアに対して適用したところ，1 つの大きな修正を別々にコミットすべき複数の小さな修正に分割できる場合があることが分かった．本論文が提案する手法は，既存手法よりも多くの修正を考慮しており，ユーザが分割の個数を指定しなくても良い点で優れている．

本研究の貢献は，既存手法よりも多くの修正を考慮して大規模なコミットを分割できることを示したことである．

以下，本稿は次のように構成されている．2. では，プログラム依存グラフについて説明する．3. では提案手法を説明する．4. では，本研究で行った実験について述べる．5. では，実験結

修正前	修正後

int x;
int y;
int z;
x = 10;
y = 20;
if(x == 10){
  y = x + y;
}
r = z + 1;
print(y);
print(r);
	int x;
	int y;
	int z;
	x = 20;
	y = 30;
	if(x == 10){
	  y = x * y;
	}
	s = z + 1;
	print('a');
	print(s);

図 1　修正前後のソースコード

果についての考察を述べる．6. では，本研究における今後の課題を述べる．最後に 7. で本稿をまとめる．

2. プログラム依存グラフ

プログラム依存グラフ（Program Dependency Graph: PDG）とは，ソースコード中の文をノードとし，文の間のデータ依存や制御依存をエッジとして表した有向グラフである [8]．一般的に，PDG はメソッド単位で構築される．また，文の間の依存関係はデータ依存のみを用いることとする．

2.1 データ依存

ソースコード中の文 s_1，s_2 に関して，以下の条件を満たす時，変数 v に関して文 s_1 から s_2 の間にデータ依存関係が存在する．

- s_1 は v を定義する．
- s_2 は v を参照する．
- s_1 から s_2 への実行経路の中に，v の再定義が起こらない経路が少なくとも 1 つ存在する．

ただし，提案手法では，“文 s は変数 v を定義する”とは，以下のいずれかの動作を表す．

- 文 s において，変数 v に対して代入処理が行われている．
- 文 s において，変数 v が指すオブジェクトの状態が変更されている．

3. 提 案 手 法

3.1 概　　要

指定したコミットにおける前後のリビジョンのメソッドを入力とし，そのコミットに含まれる修正を複数の修正に分割する．

提案手法の手順は，次の 2 ステップである．

ステップ 1　修正前後の文をデータ依存関係によってまとめる

ステップ 2　ステップ 1 でまとめた文を用いて，1 つのコミットに含まれる修正を複数の修正に分割する

提案手法を用いて，修正前のメソッドにこれらの分割された修正を 1 つずつ適用していくことで，分割された修正と同じ数の新たな複数のリビジョンを出力できる．全ての分割された修正が適用された後のリビジョンは，元のリビジョンと実行時の動作が一致する．

次に，これらの手順を具体例を用いながら詳細に説明する．

3.2 具　体　例

具体例として，図 1 のようなソースコードの修正を考える．図 2 が図 1 の PDG である．図 2 において，修正前の PDG にのみ存在するノードは B（Before），修正後の PDG にのみ存在するノードは A（After），修正前の PDG と修正後の PDG において共に存在するノードは C（Common）とタグ付けされている．また，実線のエッジはデータ依存，点線のエッジは制御依存である．

3.3 ステップ 1：修正前後の文をデータ依存関係によってまとめる

修正前の PDG にのみ存在するノードの集合を N_{before}，修正後の PDG にのみ存在するノードの集合を N_{after} とする．

まず，N_{before} に含まれるあるノードを a とする．a を出発ノードとし，N_{before} に含まれるノードのみを辿って到達可能なノードの集合 P_a を定義する．ここで，P_a は a を含む．P_a について，以下が成立する．

$$\forall a_1, a_2 \in P_a \rightarrow P_{a_1} = P_{a_2}$$

N_{before} に含まれる全てのノードについて，このような集合を抽出し，新たに $P_i (1 \leqq i \leqq l)$ とする．等しい集合には同じ番号 i をつけるとすると，以下の条件を満たす．なお，集合 A，B が等しいとは，$A \subset B$ かつ $B \subset A$，つまり， A の要素と B の要素が過不足無く一致することである．

$$i \neq j \rightarrow P_i \cap P_j = \phi$$

$$\bigcup_{i=1}^{l} P_i = N_{before}$$

このように，全ての修正が行われたメソッドについてデータ依存関係のあるノードの集合 $P_i (1 \leqq i \leqq l)$ を抽出する．修正後の PDG 中のノードについても同様に集合 $Q_j (1 \leqq j \leqq m)$ を抽出する．

具体例では，図 3 のように，ある B または A のノードから，B または A のノードのみを辿って到達可能な全てのノードが 1 つのまとまりとみなされるので，P_1，P_2，Q_1，Q_2，Q_3 がこのステップで得られる．

3.4 ステップ 2：ステップ 1 でまとめた文を用いて，1 つのコミットに含まれる修正を複数の修正に分割する

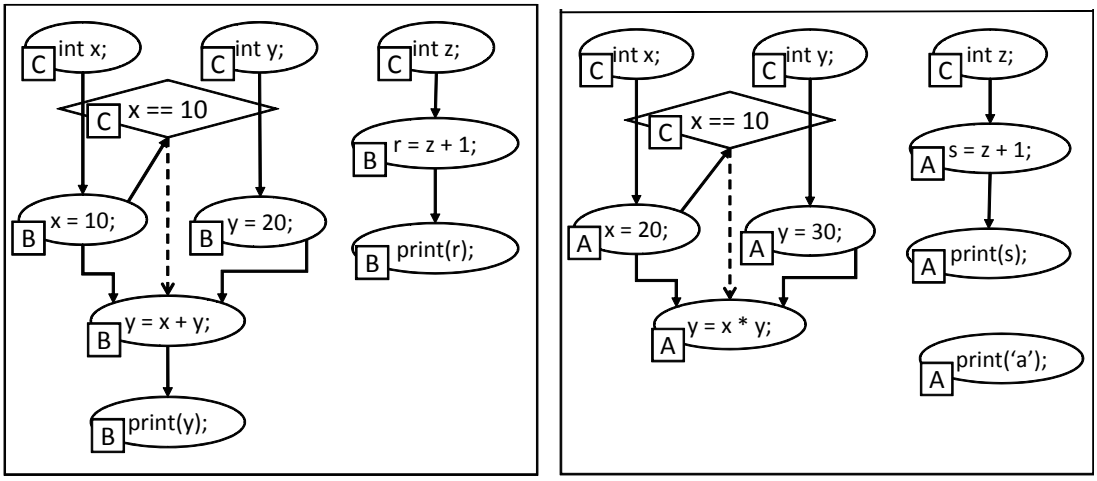
修正前の PDG と修正後の PDG において共に存在するノードの集合を N_{common} とする．N_{common} の要素であり，かつ P_i に含まれるノードと隣接しているノードの集合を N_{Pi}（$\subset N_{common}$），同じく N_{common} の要素であり，かつ Q_j に含まれるノードと隣接しているノードの集合を N_{Qj}（$\subset N_{common}$）とする．任意の N_{Pi} と N_{Qj} が，以下の場合のどれかに当てはまる．

（1）　N_{Pi}=$N_{Qj} \wedge N_{Pi} \neq \phi \wedge N_{Qj} \neq \phi$

（2）　N_{Pi}=ϕ

（3）　N_{Qj}=ϕ

(1) の場合，P_i が Q_j に変更されたとみなす．(2) の場合，P_i が修正により削除されたとみなす．(3) の場合，Q_j が修正により追加されたとみなす．これにより，1 つのコミットの中に含まれる修正が，複数の修正（変更・追加・削除）に分割される．



(a) 修正前　　(b) 修正後

図 2　修正前後の PDG

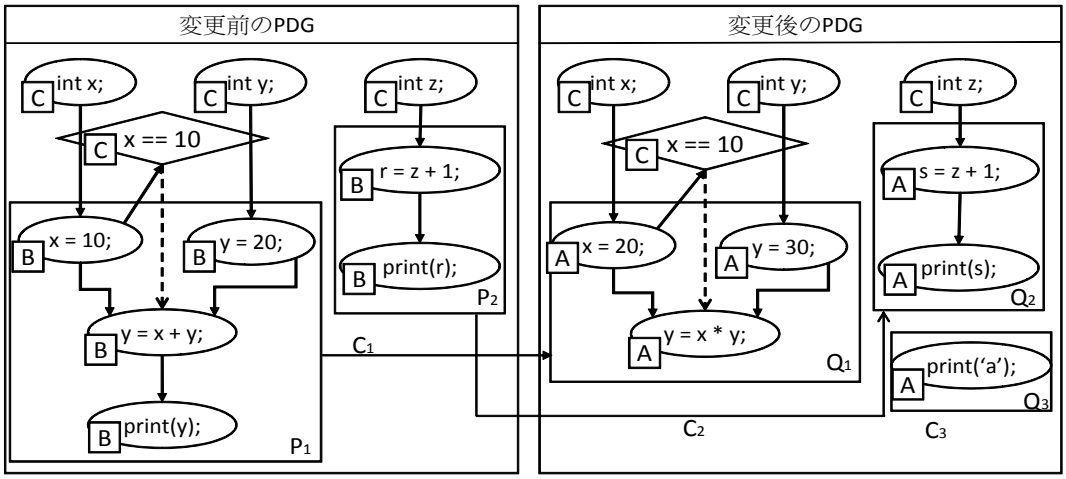


図 3　修正された文の対応関係

ここで，1 つのコミットの中に含まれる修正が n 個の修正に分割されたとし，各々の修正を $C_i(1 \leqq i \leqq n)$ と定義する．

具体例では，図 3 より，P_1 の要素と隣接する点線のノードは “int x”と “int y”のノードである．よって，N_{P1}={“int x”, “int y”} と定義される．同じく，N_{P2}={“int z”}，N_{Q1}={“int x”, “int y”}，N_{Q2}={“int z”}，$N_{Q3} = \phi$ である．N_{P1}=N_{Q1}，N_{P2}=N_{Q2} より，P_1 が Q_1 に，P_2 が Q_2 に変更されたとみなされる．また，$N_{Q3} = \phi$ より，Q_3 は新たに追加されたとみなされる．P_1 から Q_1 への変更を C_1，P_2 から Q_2 への変更を C_2，Q_3 の追加を C_3 とする．これにより，1 つのコミットの中に含まれる修正が，より細かな修正 C_1，C_2，C_3 に分割された．

3.5　分割された修正によってコミットを再構築する

あるコミットの前後のリビジョン r_k と r_{k+1} について，r_k が修正前，r_{k+1} が修正後であったとする．ステップ 2 で得られた修正 $C_i(1 \leqq i \leqq n)$ を，修正前のリビジョンに順番に適用していく．リビジョン r_k に C_1 を適用したものを r'_1，さらに C_2 を適用したものを r'_2 として，全ての修正を適用するまで繰り返す．これを繰り返すと、より分割された修正が加えられた複数のリビジョン $r'_i(1 \leqq i \leqq n)$ が生成される．

修正前	C1適用後	C2適用後	C3適用後

int x;
int y;
int z;
x = 10;
y = 20;
if(x == 10){
  y = x + y;
}
r = z + 1;
print(y);
print(r);
	int x;
	int y;
	int z;
	x = 20;
	y = 30;
	if(x == 10){
	  y = x * y;
	}
	r = z + 1;
	
	print(r);
		int x;
		int y;
		int z;
		x = 20;
		y = 30;
		if(x == 10){
		  y = x * y;
		}
		s = z + 1;
		
		print(s);
			int x;
			int y;
			int z;
			x = 20;
			y = 30;
			if(x == 10){
			  y = x * y;
			}
			s = z + 1;
			print('a');
			print(s);

図 4　ソースコードの変遷

C_k が P_i から Q_j への変更であるとき，C_k を適用するとは，修正前のソースコードから P_i に含まれる全ての文を削除し，Q_j に含まれる全ての文を追加することである．C_k が追加または削除のみであるときは，それぞれ追加・削除のみを行う．

具体例では，修正前のソースコードに，図 3 で示される修正 C_1，C_2，C_3 を順番に適用する．このとき，ソースコードの変遷は図 4 のようになる．

4. 実　　験

本研究では，提案手法により，1 つのコミットに含まれる修正が，複数のより細かな修正に分割できる場合があることを確認するための評価実験を行った．実験の対象はオープンソースソフトウェアの JRuby とした．JRuby についての情報を表 1 に示す．

4.1 実 験 方 法

実験は以下の手順で行った．なお，以下の作業は全て手作業で行っている．

手順 1　実験対象のリポジトリから，複数の修正が含まれているコミットを抽出する．

手順 2　コミットの中から，複数の修正が含まれているメソッドを選択する．

手順 3　修正前後のメソッドの PDG を作成する．

手順 4　提案手法を適用し，分割したコミットを目視で確認する．

4.2 実 験 結 果

例として，JRuby の RubyZlib クラス内の initialize メソッドに対して提案手法を適用した．initialize メソッドの修正前後のソースコードは図 5 に示す．このメソッドに対して，メンバ変数 countingStream の代入と，line の初期値を 0 から 1 にするという修正が行われている．まず，修正前後のソースコードから，PDG を作成した（図 6）．図 6 について，楕円のノードは initialize メソッド内の文であり，矩形のノードはメンバ変数がメソッド内で定義された場合に生成されるノードである．

手法を適用した結果，修正をより細かな修正 C_1，C_2 に分割できた．C_1 は，メンバ変数 countingStream を定義する修正である．この修正の理由は，BufferedInputStream の引数が入れ子になっているため，また，別のメソッドでこの変数を使用するためであると思われる．C_2 は，line の初期値を 0 から 1 に変更している．よってこれらの修正 C_1，C_2 は互いに関係がなく，別々にコミットすべきものである．

5. 考　　察

今回，様々なメソッドに対して手法を適用し，1 つのコミットに含まれる修正を適切に分割できるパターンや分割できないパターンの傾向を見つけることができた．元の修正を適切に分割するとは，複数のコミットにすべきところを認識し，元の修正を，変更・追加・削除の 3 種類のより細かな修正に分割することである．今回の実験では，3 種類の修正について，変更とみなすべきものが，追加・削除とみなされている例があった．これは，変更箇所の対応付けの精度が低いためであるので，正しく対応付けることで，より適切に元の修正を分割できる．反対に，追加・削除とみなすべきものを変更とみなしている例はほとんど見当たらなかった．

表 1　JRuby に関する情報

リビジョン数	11,134
最終的なコードの行数	101,799
最初のコミットの日付	2001/09/11
最後のコミットの日付	2012/08/04

5.1 修正を適切に分割できるパターン

修正を適切に分割できるパターンは，変数名・定数の変更である．変数名や定数の変更では，PDG のデータ依存関係が変化しないため，必ず変更前後で対応付けすることができる．

5.2 修正を適切に分割できないパターン

修正を適切に分割できないパターンとして，以下の 2 つがあった．

（1） 1 つのコミットに含まれるべき修正が分割されてしまっているパターン

（2） P_i から Q_j への変更を対応付けることができないパターン

（1）のパターンについて，アルゴリズムの変更のような大規模な修正の場合は，複数の修正に分割されやすい傾向にあった．

次に，(2）のパターンについて，図 7 では，print メソッドの引数が y から x に変更されている．ここで，x と y には同じ値が代入されている．手法の性質上，図 7 のような変更の場合，プログラムの動作が変わらない小規模な変更であっても，正しく変更箇所の対応付けができず，追加または削除とみなされてしまった．また，変更前後の PDG においてどちらか一方にのみ存在するノードと，修正前後の PDG において共に存在するノードとの間にデータ依存がない場合，必ず追加または削除とみなされるため，変更を対応付けることができない．

6. 今後の課題

今回の実験によって明らかになった改善すべき点について以下で述べる．

6.1 分割の精度の向上

提案手法では，修正が過度に分割される問題があった．これは，図 8 のように本来 1 つの修正として扱うべきものが別々の修正とみなされることが原因である．また，変更箇所の対応付けをすべきところができていない問題について，今回の手法では，3.4 で述べられている N_{P_i} のようなノードの集合を考えた．図 7 のような問題を解決するためには，N_{common} の要素であり，かつ P_i に含まれるノードと隣接しているノード以外のノードについても考慮する必要がある．

6.2 コミットの再構築

分割された修正を適用する際に，文を追加する位置についての課題がある．プログラムの動作を変えないような文の追加を行うには，ソースコード中の文の前後関係に気をつけなければならない．これは，ソースコード中の文の行番号を利用することで適切な位置に文を追加できると考えられる．また，ソースコード中の if や while などの語は PDG に反映されないので，制御依存を考慮して文の追加・削除を行う必要がある．

6.3 手法の自動化と適用範囲の拡張

手法を定量的に評価するためにも，今回の手法を複数のリポ

```
public IRubyObject initialize(IRubyObject io, Block unusedBlock) {
  realIo = io;
  try {
    this.io = new BufferedInputStream(new
        GZIPInputStream(new IOInputStream(io)));
  } catch (IOException e) {
    Ruby runtime = io.getRuntime();
    RubyClass errorClass = runtime.fastGetModule("Zlib").
        fastGetClass("GzipReader").fastGetClass("Error");
    throw new RaiseException(RubyException.newException(runtime,
        errorClass,e.getMessage()));
  }
  line = 1;
  position= 0;
  return this;
}
```

(a) 修正前

```
public IRubyObject initialize(IRubyObject io, Block unusedBlock) {
  realIo = io;
  try {
    countingStream = new CountingIOInputStream(io);
    this.io = new BufferedInputStream(new
        GZIPInputStream(countingStream));
  } catch (IOException e) {
    Ruby runtime = io.getRuntime();
    RubyClass errorClass = runtime.fastGetModule("Zlib").
        fastGetClass("GzipReader").fastGetClass("Error");
    throw new RaiseException(RubyException.newException(runtime,
        errorClass, e.getMessage()));
  }
  line = 0;
  position = 0;
  return this;
}
```

(b) 修正後

図 5　RibyZlib クラスの initialize メソッド

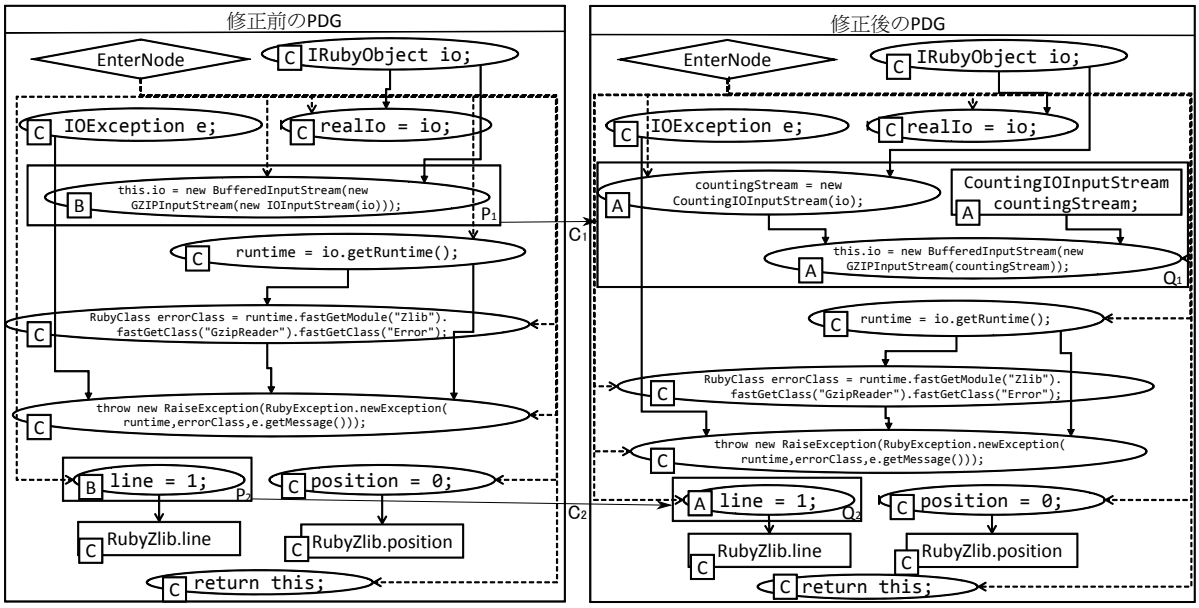


図 6　修正前後の PDG

ジトリに対し自動で行えるようなツールを開発する予定である．今回は手法を自動化するツールが無かったため，1 つのメソッドに対して手法を適用した．今後は適用範囲を修正のあった全メソッドに拡張し，コミットを分割できるようにしたい．

また，提案手法では，2. で述べているとおり，オブジェクトの変更も変数の定義とみなしている．しかし，オブジェクトが変更されたことをソースコードから判断することはとても難しい．ツールを実装する際に，オブジェクトの変更を考慮しないとすると，修正される文の間でデータ依存が存在せず，うまく文をまとめられない場合がある．図 8 のような例ではそれが顕著である．この例では，3 つの文の追加はそれぞれ 1 文ずつの別々の追加とみなされてしまう．今回の手法では，3.3 で述べられている通り，N_{before} に含まれるノードのみを辿って到達可能なノードの集合を用いた．これを，全ノードを辿って到達可能なノードの集合を用いることでこの問題を解決できる可能性がある．

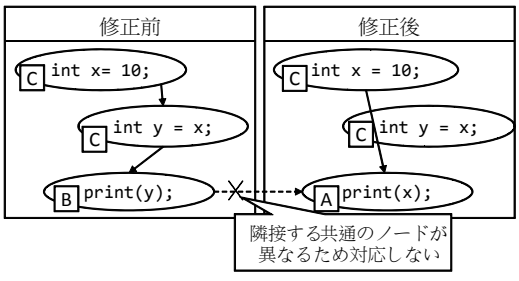


図 7　変更箇所の対応付けができない例

6.4　メソッド間での変更の対応付け

複数のメソッドに対して，変数名の変更など 1 つのコミットに含まれるべき修正が行われることがある．提案手法を複数のメソッドに対して適用する場合に，これらの修正が異なるコミットにならないようにすべきである．よって，複数のメソッドに対するこのような修正を 1 つのコミットと判定できるようなアルゴリズムを考案する必要がある．

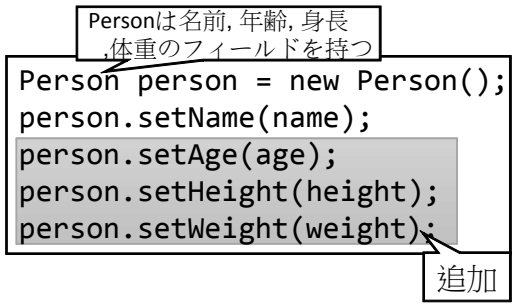


図 8　修正が不適切に分割されてしまう例

7. ま と め

本研究では，1 つのコミットに含まれる修正を複数の修正に分割する手法を提案し，オープンソースソフトウェアに対して実験を行なった．実験の結果，1 つのコミットに含まれる修正を，異なるコミットにすべき複数の修正に分割できることが分かった．今後は，手法を自動化するツールを実装して，より多くのソフトウェアに対する調査を行い，手法の定量的な評価を行えるようにする予定である．

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：25220003)，挑戦的萌芽研究 (課題番号：24650011)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の助成を得た．

文　献

[1] 松下　誠，“ソフトウェア工学の新潮流 (1) リポジトリマイニング,” ソフトウェアエンジニアリング最前線 2009，pp.21–24，Sep. 2009.

[2] 林晋平，佐伯元司，“リファクタリング支援に用いる知識抽出のためのソフトウェアリポジトリの解析,” 電子情報通信学会技術研究報告，vol.106，no.16，pp.1–6，2006.

[3] M. Askari and R. Holt, “Information Theoretic Evaluation of Change Prediction Models for Large-scale Software,” Proc. of the 3rd International Workshop on Mining Software Repositories, pp.126–132, May 2006.

[4] H. Kagdi, S. Yusuf, and J.I. Maletic, “Mining Sequences of Changed-Files from Version Histories,” Proc. of the 3rd International Workshop on Mining Software Repositories, pp.47–53, May 2006.

[5] H. Kagdi, J.I. Maletic, and B. Sharif, “Mining Software Repositories for Traceability Links,” Proc. of the 15th International Conference on Program Comprehension, pp.145–154, June 2007.

[6] A. Hindle, D.M. German, and R. Holt, “What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits,” Proc. of the 5th International Working Conference on Mining Software Repositories, pp.99–108, May 2008.

[7] K. Herzig and A. Zeller, “The Impact of Tangled Code Changes,” Proc. of the 10th International Workshop on Mining Software Repositories, pp.121–130, May 2013.

[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, pp.319–349, 1987.