ソフトウェアエンジニアリングシンポジウム **2014**
IPSJ/SIGSE Software Engineering Symposium (SES2014)

修正履歴情報を利用した コミット分割支援手法の提案

切貫 弘之[†1,a)]　堀田 圭佑[†1,b)]　肥後 芳樹[†1,c)]　楠本 真二[†1,d)]

概要：一般的に，版管理の 1 つのコミットでは複数のタスクを含む修正を行うべきではないといわれている．このような修正をもつれた修正と呼ぶ．もつれた修正を行うことは，開発者にとってデメリットが多い．例えば，複数のタスクを行うコミットが存在する場合，そのコミットに含まれるある単一のタスクのみをマージする，あるいは取り消すことができない．また，もつれた修正の存在はリポジトリマイニングの妨げとなる．なぜなら，リポジトリマイニングの手法の多くはもつれた修正がないことを前提としているからである．そこで，本研究では，過去のコミットの情報を利用することで，開発者がもつれた修正である可能性が高いコミットを行おうとした際にそれを開発者に知らせ，そのコミットを複数のコミットに分割する手法を提案する．コミットを分割する前に，まず分割候補を開発者に提示し，開発者はそのままコミットを行うか，提示された候補に従って分割されたコミットを適用するかを選択できる．このような支援を行うことによって，開発者がもつれた修正を行うことを防ぐことができる．

キーワード：バージョン管理システム, tangled changes, ソフトウェアリポジトリ

1. はじめに

近年，ソフトウェアリポジトリマイニング（以降，マイニング）に関する研究が注目され，さかんに行われている [5], [10]. ソフトウェアリポジトリ（以降，リポジトリ）にはソフトウェアの開発プロジェクトに関連する様々な情報が蓄積されている．マイニング技術を用いることで，リポジトリからソフトウェア開発に有益な情報を得ることが可能である.[6], [11], [15], [19]．マイニングによって得られた情報は将来的なソフトウェアの保守や開発に活かすことができる．

マイニングの対象となるリポジトリには様々なものがあるが，特にソースコードリポジトリを対象とする研究がさかんである．Subversion, CSV, Git に代表されるバージョン管理システムを用いて開発を行う際に作成されたソースコードリポジトリはしばしばマイニングの対象となる．ソースコードリポジトリを対象としたマイニングの研究分野は多岐にわたっており，プログラム中の欠陥に関するもの [2], [6]，ソースコードの修正に関するもの [1], [3], [4], [10], [18], [20]，コードクローンに関するものなどがある [8], [17].

バージョン管理システムを利用した開発では，一度のコミットでは複数のタスクに関する修正を行わないことを求められている [12]. しかし，このことが広く受け入れられているにも関わらず，開発者はしばしば複数のタスクに関する修正を一度にコミットすることがある．例えば，Murphy-Hill らは，リファクタリングを行った修正は他のタスクを行った修正と同時にコミットされることが多いことを明らかにしている [16]．このような複数のタスクを伴った修正を本論文ではもつれた修正と呼ぶこととする [7].

もつれた修正の存在は，ソフトウェア開発を効率よく進める上での妨げとなる [5], [7]．そのような例を以下に示す．

- もつれた修正をマージする際に手動での操作を必要とする場合がある．修正をマージするということは，あるブランチで行われた修正を他のブランチに適用することを意味する．ある修正をマージする際，マージ作業の対象としているコミットで単一のタスクのみが行われているならば，バージョン管理システムのマージ機能を用いるだけで目的を達成できる．しかし，マージ作業の対象としているコミットで複数のタスクが行われている場合，その中の特定のタスクに関する修正

[†1] 現在，大阪大学
Presently with Osaka University
a) h-kirink@ist.osaka-u.ac.jp
b) k-hotta@ist.osaka-u.ac.jp
c) higo@ist.osaka-u.ac.jp
d) kusumoto@ist.osaka-u.ac.jp

のみを適用することは難しい．そのような場合，下記のいずれかの方法をとらなければならない．
  – バージョン管理システムのマージ機能を用いず，手動でマージ操作を行う
  – バージョン管理システムのマージ機能を用いた後，手動で一部の修正を取り消す

  しかし，いずれの場合も手動による操作が必要となってしまう．
- もつれた修正では，そこで行われた特定のタスクのみを取り消すことができない．バージョン管理システムの取り消し機能を用いた場合，そのコミットで行われた全ての修正が取り消されてしまう．
- もつれた修正が存在すると，差分情報を用いたコードレビューが困難になる．なぜなら，それぞれのタスクに関する修正が，ソースコードの差分のどの部分に対応するのかが分かりづらくなるためである．

これらに加えて，もつれた修正の存在は，マイニングの妨げにもなる．なぜなら，ほとんどのマイニング手法はもつれた修正がないことを前提としているからである．

evolutionary coupling を用いた手法はその良い例である [18]．ファイルやメソッド等のソフトウェアのモジュールが修正された時，それと同時に修正されるべきものを特定するために evolutionary coupling が用いられる．多くの手法では，同じコミットで修正されたモジュールが evolutionary coupling を持つとみなされている．evolutionary coupling を用いる手法では，同時に修正されている複数のモジュールがある場合，その間に何らかの関係があると解釈されるため，本来関係のないモジュールの修正が同時にコミットされることは想定していない．従って，evolutionary coupling を用いた手法は明らかにもつれた修正がないことを前提としている．

既存のマイニング手法では，多くのファイルや多くの行が修正されているコミット（以降，ラージコミット）を取り除く前処理がしばしば行われている．なぜなら，ラージコミットでは，もつれた修正や，マイニングの際に考慮する必要のないコメントやフォーマットの修正などが行われることが多いからである [9]．ラージコミットを除去することで，もつれた修正が分析結果に与える悪影響をある程度取り除くことができる．しかし，そのような前処理はもつれた修正を取り除くという意味では決して完全ではない [14]．

- ラージコミットの中にはもつれた修正を行っていないものも存在する．従って，全てのラージコミットを取り除くことは意図せずそのようなコミットまで取り除いてしまうことになる．
- 規模の小さなコミットの中にももつれた修正が存在する．ラージコミットを対象とする前処理ではそのような場合に対応できない．

そこで本論文では，もつれた修正によって引き起こされる問題を避けるため，開発者がもつれた修正を行うことを事前に防ぐ手法を提案する．コミットする前に，潜在的なもつれた修正の存在を開発者に気づかせることができるという点で，著者らの手法は有用である．著者らの過去の研究では，過去に一度行われた修正を含むような修正はもつれた修正である可能性が高いという考えに基づき，コミットを分割する手法を提案している [13]．本研究では，これに経験則を追加することで，より適切な分割を行えるようにし，さらに手法の精度の定量的な評価を行っている．

本研究の貢献は以下のとおりである．

- 開発者が行おうとした，もつれていると思われる修正を検知し，それらを複数の修正に分割する方法を提示する手法を提案した．
- 提案手法の有効性を評価した結果，もつれた修正である可能性が高いコミットに対して提示された分割候補の中で，不適切でないと思われるものの割合は 75%から 92%であった．

2. 研究の動機

ソフトウェア開発のプロジェクトにおいて，もつれた修正がしばしば行われ，そのことは様々な問題をはらんでいることを 1 章で説明した．コミットログが，行われた修正の全てを説明しておらず，もつれた修正が行われていることがコミットログから判断できない場合も多い．

図 1 は，オープンソースソフトウェア JEdit で，もつれていると思われる修正が行われた例である．コミットログでは“fix latex mode”と書かれており，バグ修正と推測されるが，これは曖昧な表現である．実際にこのコミットで行われている修正は次の 2 つである．

- メソッド_closeBuffer() で変数 view の null チェックが追加された
- メソッド newView() で非推奨 API が推奨 API に変更された

もし，後者の修正が，バグ修正の一部として行われていないならば，これらの 2 つの修正は互いに関係がなく，別々のコミットに分けて行われるべきである．もし，このリポジトリに対し，メソッド間の evolutionary coupling を特定するためのマイニングを行ったならば，本来は関係が無い_closeBuffer() と newView() の間に何らかの関係があるとみなされてしまう．よって，このようなもつれた修正を未然に防ぐことは重要な意味を持つ．

本研究では，もつれていると思われる修正を開発者が行おうとした時にそれを検知し，コミットの分割候補を提示する手法を提案する．

提案手法では，リポジトリ中の修正履歴情報を用いることで，もつれた修正である可能性が高いコミットを検出する．異なるコミットで類似した修正が繰り返し行われる場合があることを Higo らは確認している [21]．ある単一の

```
1681    public static void _closeBuffer(View view, Buffer buffer)
1682    {
1683      if(buffer.isClosed())
1684      {
1685        // can happen if the user presses C+w twice real
1686        // quick and the buffer has unsaved changes
1687        return;
1688      }
1689
1690      PerspectiveManager.setPerspectiveDirty(true);
1691
1692      if(!buffer.isNewFile())
1693      {
1694 -      view.getEditPane().saveCaretInfo();
       +      if(view != null)
       +        view.getEditPane().saveCaretInfo();
1695        Integer _caret =  (Integer)buffer.getProperty(Buffer.CARET);
1696        int caret = (_caret == null ? 0 : _caret.intValue());

……

2127    public static View newView(View view, Buffer buffer,
          View.ViewConfig config)
……

2184        EditBus.send(new ViewUpdate(newView,ViewUpdate.CREATED));
2185
2186 -      newView.show();
       +      newView.setVisivle(true);
2187
2188        // show tip of the day
2189        if(newView == viewsFirst)
2190        {
2191          newView.getTextArea().requestFocus();
```

nullチェックの追加

推奨APIを使用する変更

図 1 もつれていると思われる修正の例

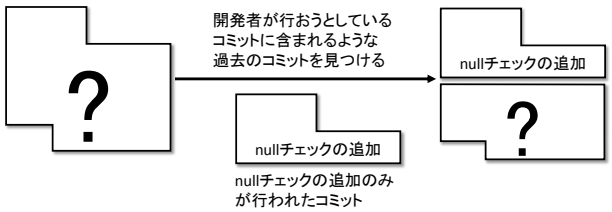

図 2 手法の軸となる考え方

タスクに関する修正 A が過去に行われ，コミットされていたとする．このとき，修正 A と，そのタスクに含まれない他の修正を同時に行うコミットはもつれた修正を行っている可能性が高い．

図 2 は手法の軸となる考え方を示すものである．開発者がバグ修正として「null チェックの追加」を含むコミットを行おうとしているとする．また，過去に「null チェックの追加」という修正のみが行われたコミットが行われていたとする．もしそのコミットがもつれた修正を行っていないならば，「null チェックの追加」という修正は 1 つのタスクであると考えられる．ここで，行おうとしているコミットが「null チェックの追加」とバグ修正ではないその他の修正を行っているとする．この場合，「null チェックの追加」を含むコミットは「null チェックの追加」を行うコミットと，それ以外の修正を行うコミットに分割すべきであると思われる．したがって，過去の修正を含むようなコミットはもつれた修正である可能性が高いと考えられる．このような過去の修正を含むような修正を本論文では**包含修正**と呼ぶこととする．

3. 提案手法

この章では，提案手法の具体的な手順を説明する．

3.1 概要

図 3 は手法のおおまかな流れを示している．手法は，大きく修正履歴情報の抽出とコミットの分割候補の提示に分けられる．修正履歴情報の抽出の手順を簡単に説明する．リポジトリを入力とし，修正履歴情報が蓄えられたデータベースを作成する．この時，修正履歴情報の抽出に用いたリポジトリの中に，もつれた修正を行ったコミットが存在する可能性がある．コミットの分割候補を開発者に提示するために用いる修正履歴は，もつれた修正でないという前提があるため，データベース中の包含修正を解消することで，もつれた修正の数を減らす処理を行う．

提案手法では，修正履歴情報の抽出を事前に行っておき，開発者がリポジトリに修正をコミットする際に，コミットの分割候補の提示を行う．以下の手順に分けて，3.2，3.3 節で詳しく説明する．

(1) 修正履歴情報の抽出
(2) コミットの分割候補の提示

(2) で提示された分割候補に従うかどうかは開発者が決定する．以降の説明では，既に n リビジョンがリポジトリ中に存在しており，開発者が新たなコミットを行う準備ができているものとする．

3.2 修正履歴情報の抽出

リビジョン r から $r+1(1 \leq r \leq n-1)$ の間の修正をソースファイルごとに抽出し，全てデータベースに登録する．その後，データベース中の包含修正を解消する．リビジョン間で行われた修正を構成する 1 つ以上の小さな修正のことを本論文では**微小修正**と呼ぶこととする．

図 4 は連続する 2 つのリビジョン間から微小修正を抽出する過程を表したものである．図 4 の例では，連続する 2 つのリビジョン間から 3 つの微小修正が抽出されている．この手順は次の 5 つのステップで構成されている．

STEP 1-1: リビジョン r から $r+1(1 \leq r \leq n-1)$ の間で修正があったソースファイルを構文解析し，ソースコード中の文を検出する．

STEP 1-2: 修正前後の 2 つのソースファイルをそれぞれ文の列とみなして比較し，最長共通部分列を特定した後，それを用いて微小修正を抽出する．

STEP 1-3: 変数名と定数リテラルを特殊文字に置換する．この際，同じ識別子は同じ特殊文字に置換し，異なる識別子は異なる特殊文字に置換する．クラス名やメソッド名などのその他のユーザ定義名は置換しない．正規化を行うことで，修正を比較する際に変数名のみが異なる文を同じ文として扱うことができる．なお，修正の比較は，修正された文の文字列を比較することで行う．

STEP 1-4: ここまでの操作で得られたファイルごとの修正をデータベースに登録する．

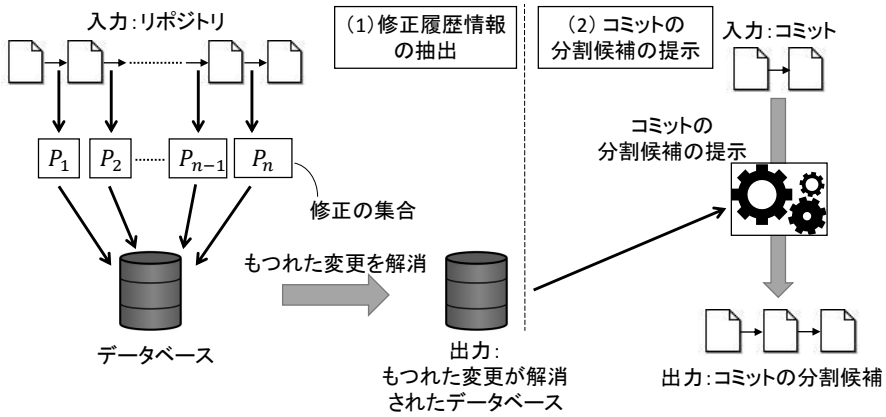

図 3　提案手法の概要

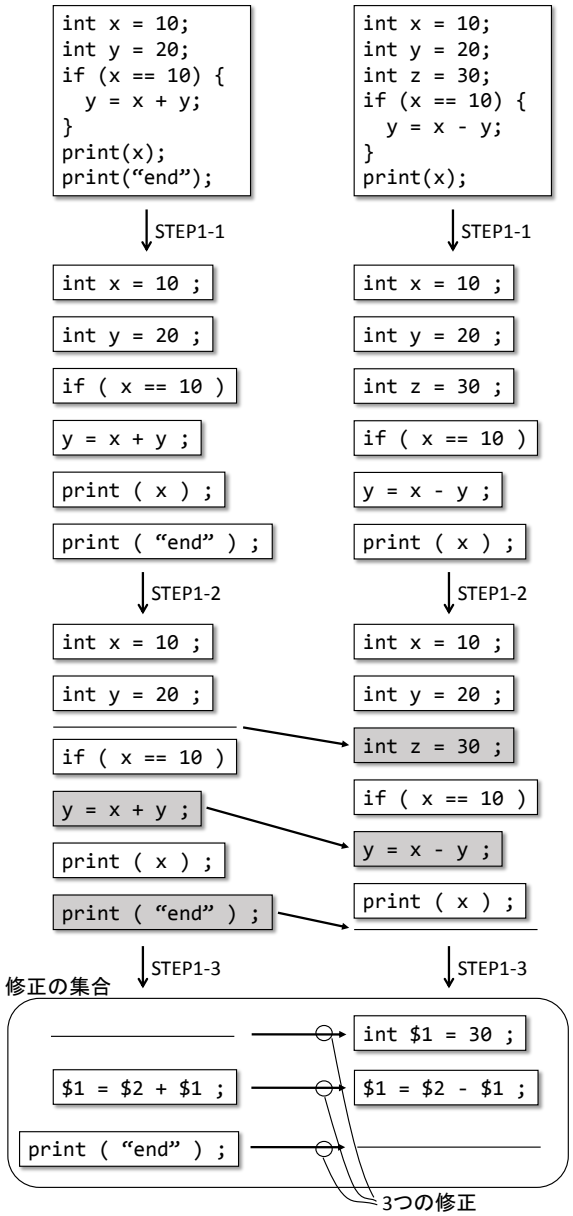

図 4　2 つのリビジョン間の修正の抽出

STEP 1-5: データベース中の包含修正を検知し，そのような修正を複数の修正に分割した上でデータベースに再登録する．STEP1-5 の詳細については，本章の最後で説明する．STEP1-1 から STEP1-5 までが行われた後のデータベースのイメージを図 5 に示す．

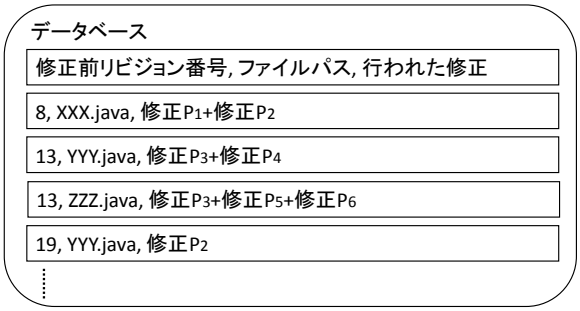

図 5　データベースのイメージ

3.2.1　最長共通部分列を用いた微小修正の抽出

STEP1-2 の最長共通部分列を用いた微小修正の抽出について説明する．最長共通部分列とは，2 つの系列の部分列のうちで最長のもののことである．例えば，(A,B,C,D,E) と (B,F,C,E,G) という 2 つの系列の最長共通部分列は (B,C,E) である．

提案手法では，修正前後のソースコードをそれぞれ文の列とみなして最長共通部分列をとっている．最長共通部分列の要素である文を修正されなかった文とみなし，要素でない文を修正された文とみなす．例えば，以下のような文の並び X,Y があるとする．

$$X = (A, B, C, D, E, F, G)$$
$$Y = (A, P, C, Q, R, D, E, G)$$

もし，X が修正前，Y が修正後であるならば，下記の 3 つの微小修正が抽出される．

- B からの P への変更
- Q と R の追加
- F の削除

図 6 はその様子を表したものである．微小修正は，それぞれ変更・追加・削除のいずれかにあてはまる．

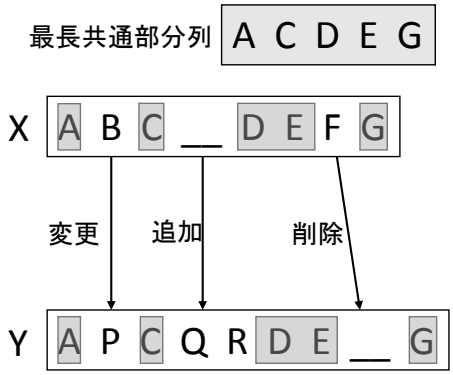

図 6　最長共通部分列を用いた修正の検出

3.2.2　データベース中の包含修正の解消

STEP1-5 のデータベース中の包含修正を解消する処理について説明する．包含修正を，複数の修正に分割することで，包含修正を解消する．修正 P と Q がデータベース中にあるとする．P と Q の要素数はそれぞれ a,b とする．m_x,n_y を微小修正とすると，以下のように表せる．

$$P = \{m_x \mid 1 \leq x \leq a\}$$
$$Q = \{n_y \mid 1 \leq y \leq b\}$$

このとき，$Q \subset P$ ならば P は包含修正であるとみなす．微小修正の比較は STEP1-3 で正規化された後の文字列を比較することで行う．

データベース中の包含修正を解消するアルゴリズムを Algorithm1 に示す．ここではデータベース中の修正の数を n としている．まず，P を修正 Q と $P-Q$ に分割する．次に，同じ変数が関わる文の修正が 1 つの集合にまとまるように $P-Q$ から Q に微小修正を移動する．なぜなら，同じ変数が関わる文の修正は同じタスクである可能性が高いと考えられるためである．また，同じ変数が関わる文の修正をまとめない場合，修正を適用する際コンパイルエラーが発生する場合がある．例えば，Q で変数 x の宣言文が削除される修正が行われており，$P-Q$ で x の値を出力する文が削除される修正が行われていたとする．このとき，もしも Q の修正を $P-Q$ よりも先に適用した場合，コンパイルエラーが発生する．

同じ変数が関わる文の修正を 1 つにまとめるアルゴリズムを Algorithm2 に示している．Algorithm2 では，P が Q と $P-Q$ に分割された時に，同じ変数が関わる文の修正が 1 つにまとまるように微小修正を Q に追加している．ここで新たにできた Q' を用いて，P を Q' と $P-Q'$ に分割する．

Algorithm 1 包含修正の解消

Input: データベース
Output: 包含修正を解消したデータベース
LABEL: START
データベース中の修正を微小修正の数について昇順でソート
for $i = 0$ **to** n **do**
  $Q \leftarrow DB[i]$
  for $j = i+1$ **to** n **do**
    $P \leftarrow DB[j]$
    if $Q \subset P$ **then**
      Algorithm2 : 同じ変数に関する修正を 1 つにまとめる (入力 : P,Q，出力 : Q')
      if $P - Q' \neq \phi$ **then**
        データベースから P を取り除き，$Q', P-Q'$ を追加する
      end if
      goto START
    end if
  end for
end for

Algorithm 2 同じ変数に関する微小修正を 1 つにまとめる

Input: 修正 $P, Q(Q \subset P)$
Output: 修正 $Q'(Q' \subset P)$
$Q' \leftarrow Q$
LABEL: START
$v_{Q'} \leftarrow Q'$ 中の修正に関わる変数の集合
for all $m \in P$ **do**
  $v_m \leftarrow$ 修正 m に関わる変数の集合
  if $v_{Q'} \cap v_m \neq \phi$ **then**
    m を Q' に追加する
    goto START
  end if
end for

3.3　コミットの分割候補の提示

3.2 節の STEP1-2，STEP1-3 と同様に開発者が行おうとしているコミットから修正を抽出し，抽出した修正に含まれるような修正がデータベース中に存在しないかを探す．そのような修正がなければ，開発者がもつれのない修正をしているとみなし，開発者はそのままコミットする．そのような修正があれば，開発者がもつれた修正を行っている可能性が高いとみなし，コミットの分割候補を提示する．開発者はコミットの分割候補を見て，それに従うかどうかを決める．アルゴリズムを Algorithm3 に示す．

この手順は下記の 3 つのステップで構成されている．開発者が包含修正をコミットした場合の手順の詳細を図 7 の例を用いながら説明する．図 7 では説明のため，正規化を行う前の文を表示している．

STEP2-1: 開発者が行おうとしているコミットから抽出した，あるファイルの修正を P とする．ここで，データベース中に $Q \subset P$ となるような修正 Q が存在する場合，P は包含修正である．図 7 の例では，Q で行われた微小修正が全て P でも行われているため，P は包含修正である．

STEP2-2: 修正 P を Q と $P-Q$ に分割する．次に，Algorithm2 を用いて，同じ変数が関わる文の修正が 1 つ

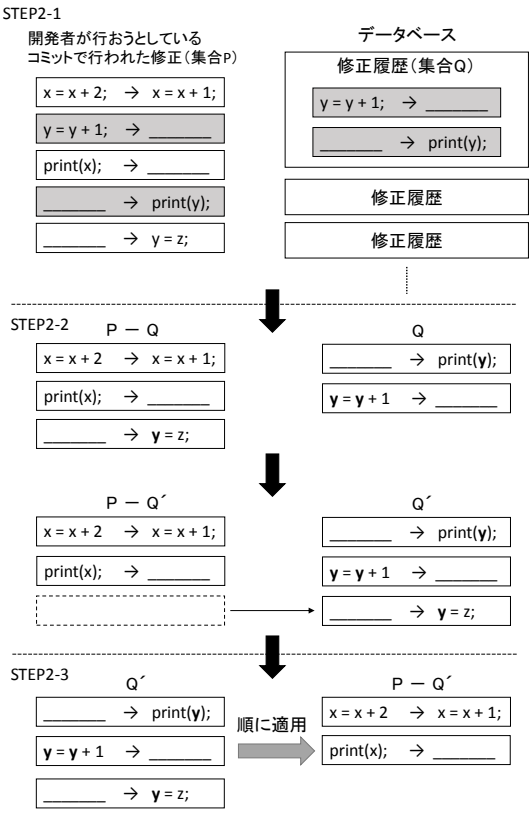

図 7 包含修正を分割する手順の具体例

の集合にまとまるように $P-Q$ から Q に微小修正を移動する．これにより微小修正が追加された Q を新たに Q' とする．図 7 では，Q と $P-Q$ 共に変数 y が出現する文の微小修正を要素に持っているので，変数 y が出現する文の微小修正を $P-Q$ から Q に移動する．Q に微小修正が追加されたものを Q' とする．

STEP2-3: 最新リビジョンに Q' を適用したものを分割候補として提示し，開発者はそれに従うかどうかを決定する．分割候補に従う場合，最新リビジョンに Q' を適用したものをコミットする．次に $P-Q'$ を新たな P として，STEP2-1 から同様に処理を行う．$P-Q'$ が包含修正でない場合は，Q' を適用した後のリビジョンに $P-Q'$ を適用して終了する．これを繰り返し行うことで，開発者が行おうとしているコミットが複数のコミットに分割される．図 7 は，$P-Q'$ が包含修正でなかった場合の例である．

開発者が一度に複数のファイルに修正を行ってコミットする場合がある．複数のファイルが一度のコミットで修正されている場合は，個別に同様の処理を行い，ファイルごとの分割された修正を組み合わせてコミットを行えるようにする．

4. 実験

提案手法の有用性を確認するため，本研究では以下に示す項目について調査を行った．実験対象のソフトウェアの情報を表 1 に示す．

Algorithm 3 コミットの分割候補の提示

```
Input: 修正 P
Output: コミットの分割候補
  LABEL: START
  for i = 0 to n do
    Q ← DB[i]
    if Q ⊂ P then
      Algorithm2：同じ変数に関する修正を 1 つにまとめる (入力：P, Q，出力：Q')
      最新リビジョンに修正 Q' を適用したものを分割候補として提示
      if 開発者が分割候補に従う then
        他のファイルで行われている修正がある場合，任意に Q' に追加する
        最新リビジョンに Q' を適用してコミット
        P ← P − Q'
        goto START
      end if
    end if
  end for
```

調査項目 1: 包含修正がもつれた修正である割合の調査.

調査項目 2: データベースの規模と検出される包含修正の数および分割精度の変化の調査.

調査項目 3: データベース作成に用いたソフトウェアと，包含修正を検出するソフトウェアが異なる場合の，検出される包含修正の数および分割候補の精度の調査.

4.1 調査の目的

各調査項目について，調査の目的を述べる．

調査項目 1: 提案手法では，検出された包含修正をもつれた修正の候補としている．どのような場合に包含修正がもつれた修正であるのかを調べる．

調査項目 2: 提案手法は，リポジトリに蓄積されている修正履歴情報の数が多ければ多いほど，包含修正の検出数が増加することが予想される．検出数の増加はどの程度であるのか，検出数が増加することによって分割精度がどう変わるのかについて調べることで，実用の際に必要なデータベースの規模を知る．

調査項目 3: 提案手法は，リポジトリにある程度データが蓄積されていることを前提としている．そのため，開発を開始して間もないプロジェクトでは手法を利用することができないという問題がある．そのような場合に，他のプロジェクトのリポジトリから作成したデータベースを代わりに利用できればそのような問題を軽

表 1 実験対象ソフトウェア

	JEdit	ArgoUML
コードの行数	299,530	164,851
リビジョン数	23,395	19,915
開発期間（月）	204	180

減できる．この時の検出数と分割候補の精度を調べることで，他のプロジェクトのリポジトリから作成したデータベースで代用可能かどうかを判断する．

4.2 調査項目 1

調査項目 1 について，調査方法と結果について述べる．調査対象にはオープンソースソフトウェアの ArgoUML を用いた．

まず，リポジトリのリビジョン 1 から 5,000 までを用いてデータベースを作成した．次に，リビジョン r から $r+1(5001 \leq r < 6000)$ に対応するコミットに対し提案手法を用いたところ，41 のコミットで包含修正が行われているとみなされた．

提案手法では，開発者が行おうとしているコミットがある修正 A を含んでおり，その修正 A がデータベース中に存在する場合に，開発者が行おうとしているコミットを修正 A とその他の修正に分割したものを候補をとして提示する．それぞれについて提示された分割候補が適切であるかどうかを目視で評価した．分割候補を以下の 3 種類に分類する．

True 修正 A が，その他の修正と関係のない単一のタスクである

False 単一のタスクであるものが分割されている

Unclear 上記の 2 つに当てはまるかどうかを判断できない

修正 A が単一のタスクであるかどうかは，コミットログや修正に関わっている変数・メソッドから判断した．分割された修正の行数がわずかである場合は，修正の内容を理解しやすいため判断が容易であるが，そうでない場合は，単一のタスクであるかを判断できない場合が多かった．

分割候補の分類は，修正 A の種類ごとに排他的に行った．分類について，以下の様な基準で行った．

- ロギング・・・ロギング用のライブラリを用いてログをとる処理を行っている
- メソッド宣言・呼び出し・・・メソッド宣言文，又はメソッド呼び出し文が追加・削除されている
- 引数の変更・・・引数の変数名や引数の個数が変更されている
- 排他制御・・・synchronized 修飾子が追加されている
- 継承・インターフェース・・・extends 又は implements 修飾子が追加されている

調査の結果は表 2 のようになった．表 2 から，包含修正の約半数がもつれた修正であることと，修正の種類によって，適切に分割しやすいものとそうでないものがあることが分かる．適切に分割されやすいものは，ロギング，引数の変更，文字列リテラルの変更などであった．また，適切に分割されにくいものは，メソッド宣言・呼び出しの変更，break 文の追加・削除などであった．

```
47  public class UMLClassifierComboBoxModel extends UMLComboBoxModel  {
……

55 +    private static final Logger LOG =
56 +        Logger.getLogger(UMLClassifierComboBoxModel.class);
57
58                                                    定数LOGの宣言
……

115    while (iter.hasNext()) {
116        before = currentStr;
117        currentEntry = afterEntry;
118        currentStr = after;
119        afterEntry = (UMLComboBoxEntry) iter.next();
120        after = afterEntry.getShortName();
121 -        if (currentEntry != null)
    -                currentEntry.checkCollision(before,after);
    +        if (currentEntry != null) {
    +            currentEntry.checkCollision(before, after);
    +        }                                          フォーマットの変更
122    }
123
124 -  if (afterEntry != null) afterEntry.checkCollision(currentStr, null);
    +  if (afterEntry != null) {
    +      afterEntry.checkCollision(currentStr, null);
    +  }
……

185        catch (InvocationTargetException ex) {
186 -              cat.error(ex.getTargetException()
    -                      + " is IncovationTargetException in
    -                          UMLClassifierComboBoxModel", ex);
    +              LOG.error(ex.getTargetException()
    +                      + " is IncovationTargetException "
    +                      + "in UMLClassifierComboBoxModel"
    +                      , ex);                       LOGを使用する変更
187        setSelectedItem(null);
188        }
189        catch (Exception e) {
190 -          cat.error("Exception in targetchanged", e);
    +          Log.error("Exception in targetchanged", e);
191            setSelectedItem(null);
192        }
193  }
```

図 8 もつれた修正であるとみなされた例

提案手法が適切に分割候補を提示できた例を図 8 に示す．図 8 では，以下の修正が行われている．

- 定数 LOG の宣言の追加
- ロギングに LOG を使用するようにする変更
- if 文のフォーマットの変更

提案手法によってこのコミットは以下の 2 つに分割された．

(1) 定数 LOG の宣言の追加と，ロギングに LOG を使用するようにする変更
(2) if 文のフォーマットの変更

表 2 分割された修正の種類と分類

修正の種類	T	U	F	合計
メソッド名変更	3	0	1	4
メソッド宣言・呼び出し	0	1	3	4
ロギング	3	0	0	3
排他制御	1	0	0	1
引数の変更	6	2	0	8
変数名の変更	1	0	0	1
変数・定数宣言	2	1	1	4
文字列リテラルの変更	3	0	0	3
クラス名の変更	0	0	1	1
継承・インターフェース	0	1	0	1
break 文の追加・削除	0	0	2	2
try 文の追加・削除	0	3	0	3
分類不可	0	4	0	4
合計	19	13	8	41

T : True, U : Unclear, F : False

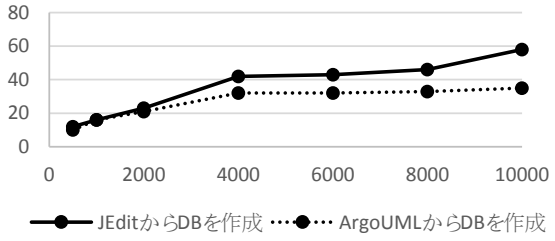

図 9 調査項目 3：データベースの規模と検出数 (JEdit)

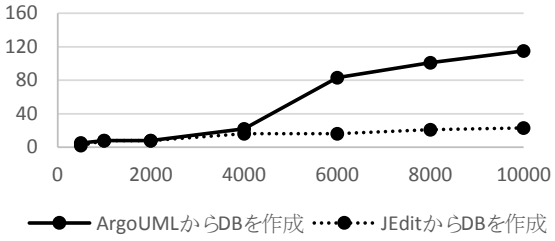

図 10 調査項目 3：データベースの規模と検出数 (ArgoUML)

この 2 つの修正は互いに関係がなく，別々にコミットされるべきであると考えられる．また，定数 LOG の宣言の追加とロギングに LOG を使用するようにする変更は別々にコミットするべきではないと思われる．なぜなら，定数の宣言とその定数の使用は単一のタスクであると考えられ，また，コミットを行う順番によってはコンパイルエラーが発生するためである．

4.3 調査項目 2

調査項目 2 について，調査方法と結果について述べる．調査対象にはオープンソースソフトウェアの ArgoUML を用いた．この調査の手順を以下に示す．

(1) リビジョン 1 から n (n=500, 1000, 2500, 5000) までを用いてデータベースをそれぞれ作成する．
(2) 作成したそれぞれのデータベースを用いて，リビジョン r から $r+1$ ($5001 \leq r < 6000$) に対応するコミットで包含修正を検出する．
(3) 提示された分割候補を，4.2 章で定義した“True”，“False”，“Unclear”に分類する．

その結果を表にしたものが表 3 である．データベースの規模に関わらず，True，Unclear，False の順で多かった．1,000 リビジョン以上の規模で分割候補が True であったものの割合は，37.5%–53.8%の間であった．また，True と Unclear を合わせた割合は，75.0%–92.3%の間であった．

4.4 調査項目 3

調査項目 3 について，調査方法と結果について述べる．調査対象にはオープンソースソフトウェアの JEdit と ArgoUML を用いた．

まず，包含修正が行われているとみなされたコミットの数について調べた．その手順を以下に示す．

表 3 調査項目 2：データベースの規模とコミットの分割候補の精度

リビジョン	True	Unclear	False	合計
1–500	6(75.0%)	1(12.5%)	1(12.5%)	8
1–1,000	7(53.8%)	5(38.5%)	1(7.7%)	13
1–2,500	9(37.5%)	9(37.5%)	6(25.0%)	24
1–5,000	19(41.3%)	18(39.1%)	9(19.6%)	26

(1) 調査対象となる 2 つのソフトウェアについて，リビジョン 1 から n (n=500, 1000, 2000, 4000, 6000, 8000, 10000) までを用いてデータベースをそれぞれ作成する．
(2) 作成したそれぞれのデータベースを用いて，JEdit のリビジョン r から $r+1$ ($10001 \leq r < 11000$) に対応するコミットに対して提案手法を用いる．
(3) 包含修正が行われているとみなされたコミットの数を数える．

その結果をグラフにしたものが図 9 である．

同じく，ArgoUML のリビジョン r から $r+1$ ($10001 \leq r < 11000$) に対応するコミットに対し提案手法を用い，その結果包含修正であるとみなされたコミットの数をグラフにしたものが図 10 である．

図 9,10 について，縦軸は包含修正の数，横軸はデータベース作成に用いたリビジョン数である．JEdit と ArgoUML 双方 4,000 リビジョン以下の規模のデータベースではも包含修正の数の差が小さいが，それより上の規模では，特に ArgoUML で差がでていることが分かった．

次に，データベースの元となるソフトウェアと包含修正を検出するソフトウェアが異なる場合の分割候補の精度について調べた．

JEdit のリビジョン 1 から 5,000 を用いてデータベースを作成し，ArgoUML のリビジョン r から $r+1$ ($5001 \leq r < 6000$) に対応するコミットが，包含修正である時に提示された分割候補を“True”, “Unclear”, “False”に分類したところ，それぞれの割合は 58%, 21%, 21%であった．

また，JEdit のリビジョン 1 から 5,000 を用いてデータベースを作成し，ArgoUML のリビジョン r から $r+1$ ($5,001 \leq r < 6,000$) に対応するコミットが，包含修正とみなされた時に提示された時の分割候補を“True”, “Unclear”, “False”に分類したところ，それぞれの割合は 50%, 44%, 6%であった．

5. 議論

各調査項目の結果に対する考察を述べる．

5.1 調査項目 1 の結果に対する考察

適切に分割されやすい修正の 1 つとして，ロギング処理

の追加があった．ロギング処理は専用のライブラリやクラスを用いて行うため，決まった命令が用いられることが多い．従って，データベース中にロギング処理の追加を行った修正履歴があれば，その後同様の処理を行った時に検出されやすい．

次に，適切に分割できなかった修正について説明する．適切に分割されない原因の 1 つとして，import 文の存在がある．import 文を追加することによって使用可能になるメソッドがあるが，import 文の追加と対応するメソッド呼び出しの追加が別のコミットに分かれる案を提示してしまうことがあった．import 文の追加と，対応するメソッドの呼び出しの追加は同じコミットで行うべきである．同様に，継承・インターフェースの修正についても，それによって使用可能なメソッドが変わる場合があり，継承・インターフェースの修正と対応するメソッド呼び出しが別のコミットに分かれてしまうことがあった．

この問題を解決する方法として，クラスやインターフェースとそれらが持っているメソッドの対応関係を用いることが考えられる．例えば，import 文の追加によって新たに使用可能になるメソッドを記憶しておくことで，そのようなメソッドの呼び出しが同時に追加されている時，import 文の追加とメソッド呼び出しの追加が別々のコミットに分かれないようにすることができる．

適切に分割されない他の原因として，break 文の追加のみが行われるなどの，粒度の小さすぎる修正の存在があった．このようなコミットがあると，break 文の追加を含むようなコミットが全て包含修正となってしまう．しかし，実際は break 文の追加を含むコミットがもつれた修正でない場合も多く，そのような場合に誤検出が発生する．

この問題を解決する方法として，break 文の追加など，単一のタスクとしては粒度が小さすぎる修正については個別に除外するという経験則を用いる方法が考えられる．

5.2 調査項目 2 の結果に対する考察

1,000 リビジョン以上の規模でデータベースを作成した場合，検出数が増加しても分割候補の精度が大きく変化することはなかった．データベースの規模が大きければ大きいほど，検出数が増加するため，より検出漏れを減らすことができる．分割候補の精度はデータベースの規模によらないため，実用の際はなるべく大きな規模のデータベースを用いるべきである．

10,000 リビジョンの規模のデータベースを用いた場合でも，包含修正を検出するのにかかる時間は 1 コミットあたり 0.2 秒程度であった．検出時間はデータベース中の修正履歴の数を n とし，$O(n)$ であるので，規模の大きなデータベースを用いた時の実行時間についての問題はないと考えられる．

5.3 調査項目 3 の結果に対する考察

図 9,10 共に，4,000 リビジョン以下の規模のデータベースでは検出数自体が少ないため，4,000 リビジョンより大きな規模のデータベースを用いた場合の結果について議論を行う．JEdit のリポジトリでの包含修正の検出数について，同じソフトウェアからデータベースを作成した場合の検出数に対して，ArgoUML からデータベースを作成した場合の検出数は平均 31.2%減少している．

一方，ArgoUML のリポジトリでの包含修正の検出数について，同じソフトウェアからデータベースを作成した場合の検出数に対して，JEdit からデータベースを作成した場合の検出数は平均 80.0%減少している．

この結果から，使用するソフトウェアによって，検出数が大きく減る場合とそうでない場合があることが分かった．手法の性質上，あるソフトウェア A 中の包含修正を，他のソフトウェア B から作成したデータベースを用いて検出する場合，ソフトウェア A 特有の修正は検出することができない．ArgoUML は JEdit と比べて，そのソフトウェア特有の修正が多かったため，結果に差が出たと考えられる．その他の原因として，ArgoUML で多くのソースファイルに対して同様の包含修正が行われた事が考えられる．同様の包含修正が複数ある場合，それらの修正が検出されるかどうかで検出数に大きく差が出てしまうため，このような結果になったと考えられる．

また，データベースの元となるソフトウェアと包含修正を検出するソフトウェアが異なる場合，検出数は減少するが，分割候補の精度は変わらないことが分かった．従って，開発プロジェクトが始まって間もない時期に十分な規模のデータベースが作成できない場合，検出数は減るものの，他のソフトウェアから作成したデータベースで代用することが可能であると考えられる．

また，他のソフトウェアからデータベースを作成する際，包含修正を検出するソフトウェアとより近いソフトウェアを用いたほうがより多くの包含修正を検出しやすいと考えられる．従って，同じ機能を持っていたり，同じライブラリを使用しているソフトウェアからデータベースを作成すべきである．

6. 妥当性への脅威

本研究の実験では，対象としたオープンソースソフトウェアは 2 つで，さらに限られたリビジョンに対し手法を適用した．よって，他のリビジョンまたは，他のソフトウェアを実験対象とすることで包含修正の検出数や分割候補の精度が変わる恐れがある．さらに多くのソフトウェアに対し実験を行うことで，より一般的な結果を得られると考えられる．

また，実験では，提案手法によって提示された分割候補が適切かどうかの分類は著者らが目視で行っている．分割

候補の分類は判断基準を設定して行ったが，主観による部分を完全に取り除くことはできないため，人によって評価結果が変わる恐れがある．従って，複数の被験者を用意したり，実際にコミットした開発者に実験を行ってもらうことでより客観的に判定できると思われる．

7. おわりに

本論文では，開発者がもつれた修正を行うことを防ぐ手段として，包含修正がコミットされた場合にそれを検知する手法を提案した．この手法は，包含修正がもつれた修正である可能性が高いという考えに基づいている．提案手法を用いることで，開発者はもつれた修正をコミットする前にそのことに気づくことができる上，提示された分割候補に従ってコミットすることでもつれた修正を減らすことができる．

著者らは，JEdit と ArgoUML という 2 種類のオープンソースソフトウェアに対し，提案手法の有用性を評価する実験を行った．実験の結果，修正の種類によって適切に分割しやすいものとそうでないものがあることが分かった．また，包含修正を検知するソフトウェアとは別のソフトウェアから作成したデータベースを用いた場合も，提案手法を利用できることが分かった．今後，今回適切に分割できなかった種類の修正について適切に分割できるような経験則を導入し，手法を改善する予定である．

また，本研究では提案手法を実装したプロトタイプを用いて実験を行ったが，提案手法を Eclipse のプラグインとして実装し，実用可能なツールを作成する予定である．

謝辞 本論文は，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)，及び文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の支援を受けて行われた．

参考文献

[1] Askari, M. and Holt, R.: Information Theoretic Evaluation of Change Prediction Models for Large-scale Software, *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pp. 126–132 (2006).

[2] Boogerd, C. and Moonen, L.: Evaluating the relation between coding standard violations and faultswithin and across software versions, *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pp. 41–50 (2009).

[3] Canfora, G., Cerulo, L. and Penta, M. D.: Identifying Changed Source Code Lines from Version Repositories, *Proceedings of the Fourth International Workshop on Mining Software Repositories*, pp. 14– (2007).

[4] Giger, E., Pinzger, M. and Gall, H.: Can we predict types of code changes? An empirical analysis, *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pp. 217–226 (2012).

[5] Hassan, A.: The road ahead for Mining Software Repositories, *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pp. 48–57 (2008).

[6] Hata, H., Mizuno, O. and Kikuno, T.: Bug Prediction Based on Fine-grained Module Histories, *Proceedings of the 34th International Conference on Software Engineering*, pp. 200–210 (2012).

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

[8] Higo, Y., Hotta, K. and Kusumoto, S.: Enhancement of CRD-based Clone Tracking, *Proceedings of the 2013 International Workshop on Principles of Software Evolution*, pp. 28–37 (2013).

[9] Hindle, A., German, D. M. and Holt, R.: What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits, *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 99–108 (2008).

[10] H.Kagdi, H., Maletic, J. I. and Sharif, B.: Mining Software Repositories for Traceability Links, *Proc. of ICPC2007*, pp. 145–154 (2007).

[11] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B. and Hassan, A. E.: Revisiting Common Bug Prediction Findings Using Effort-aware Models, *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pp. 1–10 (2010).

[12] KDE_TechBase: Commit Policy, (online), available from ⟨https://techbase.kde.org/Policies/Commit_Policy⟩ (accessed 2014-07-23).

[13] Kirinuki, H., Higo, Y., Hotta, K. and Kusumoto, S.: Hey! Are You Committing Tangled Changes?, *Proceedings of the 22Nd International Conference on Program Comprehension*, pp. 262–265 (2014).

[14] Kusunoki, N., Hotta, K., Higo, Y. and Kusumoto, S.: How Much Do Code Repositories Include Peripheral Modifications?, *Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 19–24 (2013).

[15] Moser, R., Pedrycz, W. and Succi, G.: A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction, *Proceedings of the 30th International Conference on Software Engineering*, pp. 181–190 (2008).

[16] Murphy-Hill, E., Parnin, C. and Black, A. P.: How We Refactor, and How We Know It, *Proceedings of the 31st International Conference on Software Engineering*, pp. 287–297 (2009).

[17] Rahman, F., Bird, C. and Devanbu, P.: Clones: What is That Smell?, *Empirical Softw. Engg.*, Vol. 17, No. 4-5, pp. 503–530 (2012).

[18] Ying, A. T. T., Murphy, G. C., Ng, R. and Chu-Carroll, M. C.: Predicting Source Code Changes by Mining Change History, *IEEE Trans. Softw. Eng.*, Vol. 30, No. 9, pp. 574–586 (2004).

[19] Zhang, H.: An investigation of the relationships between lines of code and defects, *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 274–283 (2009).

[20] Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A.: Mining Version Histories to Guide Software Changes, *Proceedings of the 26th International Conference on Software Engineering*, pp. 563–572 (2004).

[21] 肥後芳樹，楠本真二：コード修正履歴情報を用いた修正漏れの自動検出，情報処理学会論文誌，Vol. 54, No. 5, pp. 1686–1696 (2013).