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.

相関ルールマイニングを用いたソースコードの修正候補の推薦

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　開発者がソースコードに対して修正を行う場合，修正が必要な全ての箇所を把握している必要がある．しかし，開発者は修正が必要な全ての箇所を必ずしも把握しているわけではないため，行うべき修正の一部が行われない恐れがある．そのような問題を解決するため，開発者がファイルやメソッドに修正を加えた際に，それと同時に修正を行うべきファイルやメソッドを推薦する研究が行われている．しかし，これらの研究には，どの行にどのような修正を行うのかという具体的な情報については提示することができないという問題がある．そこで，本研究では，開発者が修正を行った際にそれと同時に行われるべき修正の内容を提示する手法を提案する．この手法では，同じコミットで行われやすい修正の組があると仮定し，プログラム文レベルで修正間の相関ルールを作成する．そしてそのルールを参照することで，開発者が行った修正から同時に行われるべき修正の内容を候補として推薦する．

キーワード　相関ルール，ソースコードリポジトリ，リポジトリマイニング，修正漏れ

Recommending Source Code Modifications by Using Association Rule Mining

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

1. ま え が き

機能追加やバグ修正などのためにソースコードに対して修正を行う際，しばしば，複数の箇所に同時に修正を行わなければならない場合がある．そのような場合，開発者は修正が必要な全ての箇所を把握している必要がある．しかし，開発者は修正が必要な全ての箇所を必ずしも把握しているわけではないため，行うべき修正の一部が行われない恐れがある．行うべき修正の一部が行われないことは，バグを発生させる原因になりうる [1]．このような修正漏れを防ぐため，様々な研究がなされている．

代表的なものとしてロジカルカップリングを用いた研究がある [2]〜[4]．ロジカルカップリングとは，プロジェクト中のあるモジュールと別のモジュールが同時に変更されやすいという論理的な結び付きのことである．ロジカルカップリングは修正漏れの防止以外にも，ライブラリの再利用パターンの発見などの目的に用いられている [5]．修正漏れの防止を目的とした既存研究では，ファイル間やメソッド間でのロジカルカップリングが用いられることが多い．これらの研究により，例えば，あるメソッドが修正された際に，同時に修正されやすいメソッドを開発者に提示することができる．しかし，これらの研究には，どの行にどのような修正を行うのかという具体的な情報については提示することができないという問題がある．したがって，提示されたメソッドにおいて行うべき修正の内容を開発者が把握していない場合，メソッドレベルの推薦では修正を行うのに不十分である恐れがある．

また，修正漏れを防ぐための手段として，grep によるキーワード検索やコードクローンの検出といったものがよく用いられる [6]．これらの方法は，修正が必要な類似したコード片を発見するのには効果的であるが，そうでないコード片に対する修正には対応することができない．ソフトウェア開発においては類似していないコード片が同時に修正されることも多く，そのような修正においても修正漏れを防ぐ必要がある．

そこで，本研究では，開発者が修正を行った際にそれと同時に行われるべき修正の内容を提示する手法を提案する．この手法では，同じコミットで行われやすい修正の組があると仮定し，プログラム文レベルで修正間の相関ルールを作成する．そしてそのルールを参照することで，開発者が行った修正から同時に

行われるべき修正の内容を候補として推薦する．提案手法を使うことで，従来手法では提示することのできなかった具体的な修正内容を提示することができ，さらに類似していないコード片に対する修正にも対応することができる．

本研究では，提案手法を実装し，手法の有効性を評価するための実験を行った．実験では，手法の定量的な評価を行うために，行うべき修正の一部が欠けているコミットを作り，それらについて欠けている修正を推薦できるかを調査した．調査の結果，提案手法は高い精度で欠けている修正を提示することができた．さらに，提案手法を用いて実際の開発で起こった修正漏れを発見できるかを調査したところ，自動的に検出された修正漏れ候補の約半数が実際に修正漏れであった．

2. 相関ルール

相関ルールとは，ある事象 x の下である事象 y が発生するという関係を表し．一般的に $x \Rightarrow y$ のように表される [7]．事象 x を条件部 (または前提部) と呼び，事象 y を帰結部 (または結論部) と呼ぶ．相関ルールは，同時に起こった事象の集合から作成される．各事象をアイテム，同時に起こった事象の集合をトランザクションと呼ぶ．

各ルールを評価するための指標として，支持度・確信度が一般的に用いられる．各指標について説明する．ここでは，アイテム x を含むトランザクションの数を $freq(x)$，2 つのアイテム x と y を含むトランザクションの数を $freq(x,y)$ と表現する．また，全トランザクション数を N とする．

支持度　ルール $x \Rightarrow y$ の支持度とは，トランザクションが 2 つのアイテム x と y を含む頻度，つまり，事象 x と事象 y が同時に起こる頻度を表す指標である．支持度が高いほど x と y の組み合わせがトランザクション中に存在する確率が高く，より重要なルールであることが期待される．支持度が低いルールは再現される機会が少なく，また x と y がトランザクション中に偶然存在したために作成された可能性が高いといえる．支持度は以下の式で表される．

$$supp(x \Rightarrow y) = freq(x,y)$$

確信度　ルール $x \Rightarrow y$ の確信度とは，アイテム x を含むトランザクションがアイテム y を含む確率，つまり，事象 x が起こった時に同時に事象 y が起こる確率のことである．確信度は以下の式で表される．確信度は 0 以上 1 以下の実数値である．

$$conf(x \Rightarrow y) = \frac{freq(x,y)}{freq(y)}$$

3. 提 案 手 法

本研究では，バージョン管理されているシステムにおいて，ソースコードに修正を加えた際にそれと同時に行うべき修正の内容を提示する手法を提案する．手法の手順は大きく以下の 3 つに分けられる．

A. 修正履歴情報のデータベース化
B. 相関ルールの作成
C. 修正候補の提示

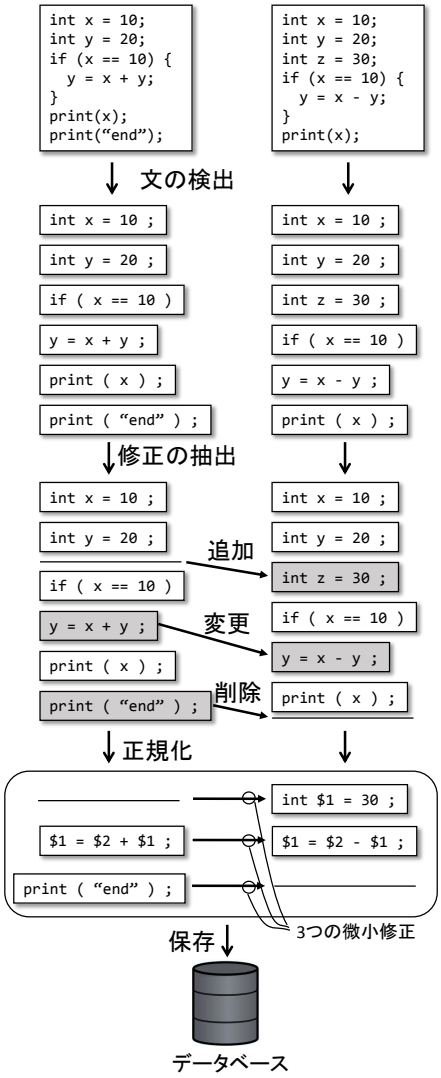


図 1　修正の抽出

修正履歴情報とは，過去のコミットにおいて，ソースコードにどのような修正が行われたかについての情報である．例えば，リビジョン r から $r+1$ の間で，“x = x + 1;” という文が追加されたといったような情報がそれにあたる．修正履歴情報を抽出する処理では，リポジトリを入力として与え，各コミットで行われた修正を抽出し，正規化を行った後データベースに格納する．相関ルールを作成する処理では，データベース中の修正履歴情報を入力として与え，修正間の相関ルールを出力する．修正候補を提示する処理では，修正が行われたファイルを入力として与えることで，それと同時に行うべき修正の内容が正規化された状態で出力される．なお，提示された修正候補は入力として与えたファイルにおいて行われるべき修正であるとは限らない．以降の説明では，リポジトリに対して n 回のコミットが既に行われているとする．

3.1 A. 修正履歴情報のデータベース化

リポジトリから修正履歴情報を抽出し，データベース化する処理について説明する．リビジョン r から $r+1 (1 \leqq r \leqq n-1)$ 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．図 1 は連続する 2 つのリビジョンのソースコード

から修正を抽出し，データベースに保存する過程を表したものである．

この手順は次の 4 つのステップで構成されている．

STEP-A1: ソースコード中の文の検出

STEP-A2: リビジョン間で行われた修正の抽出

STEP-A3: 識別子の正規化

STEP-A4: 正規化された修正情報の保存

以降では，それぞれのステップについて詳細に説明する．

STEP-A1: ソースコード中の文の検出

リビジョン r から $r+1(1 \leqq r \leqq n-1)$ 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．ソースコード中のコメントについては無視する．

STEP-A2: リビジョン間で行われた修正の抽出

修正前後の 2 つのソースファイルを最長共通部分列アルゴリズムを用いて比較し，修正を抽出する．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．

提案手法では，系列がソースファイルに対応し，系列の要素が文に対応している．つまり，修正前後のソースファイルをそれぞれ文の列とみなして最長共通部分列アルゴリズムを用いている．ここで，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．最長共通部分列の長さを m としたとき，最長共通部分列の各要素を $common_k(1 \leqq k \leqq m)$ とする．また，修正前の文の列において最長共通部分列の要素でない文 $before_k(0 \leqq k \leqq m)$ を以下のように定義する．

$before_0$: $common_1$ の前に来る文の列

$before_k(1 \leqq k \leqq m-1)$: $common_k$ と $common_{k+1}$ 間の文の列

$before_m$: $common_m$ の次に来る文の列

修正後の文の列についても同様に $after_k(0 \leqq k \leqq m)$ を定義する．なお，$before_k$，$after_k$ は空でもよい．このとき，2 つのソースファイル間で行われた修正を複数の小さな修正であるとみなし，それらを変更・追加・削除の 3 種類に分類する．本論文では，このような文の列の変更・追加・削除を**微小修正**と定義する．変更・追加・削除の定義を以下に示す．

$before_k$ から $after_k$ への変更: $before_k \neq \emptyset \wedge after_k \neq \emptyset$

$after_k$ の追加: $before_k = \emptyset \wedge after_k \neq \emptyset$

$before_k$ の削除: $before_k \neq \emptyset \wedge after_k = \emptyset$

STEP-A3: 識別子の正規化

このステップでは STEP-A2 で得られた微小修正中の識別子に対し正規化を行う．正規化とは，ここでは識別子を特殊な文字列に置換することを表す．正規化を行うことで，識別子名のみが異なる文を同じ文として扱うことができるようになる．本研究では変数名と定数名に対し正規化を行う．また，ここでは正規化処理として Parameterized Matching という方法を用いる．Parameterized Matching は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という正規化手法である．

STEP-A4: 正規化された微小情報の保存

最後に，STEP-A3 で正規化された微小修正を修正が行われたリビジョン番号とともにデータベースに保存する．ここまでの操作によって，リビジョン r から $r+1(1 \leqq r \leqq n-1)$ の間で行われた微小修正がデータベースに保存される．

STEP-A1〜STEP-A4 の操作をリポジトリ中の全てのリビジョン間に対して繰り返し行うことで，リビジョン 1 から n までに行われた全ての微小修正をデータベースに保存する．

3.2 B. 相関ルールの作成

提案手法では，微小修正間の相関ルールを作成する．相関ルールは条件部と帰結部から成り立ち，条件部と帰結部に微小修正が 1 つずつ対応する．相関ルールは 1 つのコミットで行われた全ての微小修正間について作成される．よって，コミットで行われた微小修正の数を a とすると，そのコミットから作成される相関ルールの数は ${}_aP_2$ となる．コミットで行われた微小修正の数が 1 つだけの場合はルールが作成できないため，そのようなコミットは無視する．

各相関ルールは評価指標として，支持度と確信度を持つ．

3.3 C. 修正候補の提示

本節では，開発者に対して修正候補を提示する処理について説明する．ここでは，開発者は最新リビジョンに対して修正を行っており，修正を行ったソースコードがコンパイル可能な状態であるとする．修正候補を提示する処理は以下の 2 つのステップで構成されている．

STEP-B1: 最新リビジョンに対して行われた修正の抽出

STEP-B2: 提示する修正の決定

以降では，それぞれのステップについて詳細に説明する．

STEP-B1: 最新リビジョンに対して行われた修正の抽出

3.1 節の STEP-A1〜STEP-A3 と同じ手順で最新リビジョンに対して行われた微小修正を抽出する．まず，3.1 節の STEP-A1 と同様に，最新リビジョンのソースコードと最新リビジョンに対して修正が行われた後のソースコードに対して構文解析を行い文を特定する．次に，STEP-A2 と同様の方法で，最新リビジョンに対して行われた微小修正を抽出する．最後に，STEP-A3 と同様に抽出された微小修正中の識別子に対して正規化を行う．ここまでの手順により，最新リビジョンに対して行われた微小修正を正規化された状態で抽出できる．

STEP-B2: 提示する修正の決定

最新リビジョンに対して行われた各微小修正に対し，相関ルールに基づいてそれと同時に行われやすい微小修正の内容を候補として提示する．開発者は，提案手法を用いて修正候補の提示を行う前に，パラメータとして最低支持度と最低確信度を設定する必要がある．ここで，設定した閾値以上の支持度と確信度を持つ相関ルールのみが有効なルールとなる．閾値に高い値を設定した場合，ルールの数が減るが，正しいと思われるルールの比率が高くなる．一方，閾値に低い値を設定した場合，ルールの数が増えるが，正しいと思われるルールの比率が低くなる．

ここで，最新リビジョンに対して行われた微小修正の集合を X，有効な相関ルールの集合を R とする．このとき，開発者に提示される微小修正の集合 Y は以下のように表される．

$$Y = \{y | x \Rightarrow y \in R, x \in X\}$$

つまり，開発者に提示される微小修正の集合 Y は，最新リビジョンに対して行われた微小修正を条件部に持つようなルールにおける帰結部の集合である．提示される修正候補は各種評価指標を用いて並べ替えを行うことができる．並べ替えを行った場合，一般的に上位に提示される候補のほうが信頼性が高いと考えられる．並べ替えには以下の 3 種類の評価指標を用いる．これらは全て，値がより大きい方がより信頼度が高いと考えられる．

支持度　修正候補が提示される元となった相関ルールの支持度を示す．そのようなルールが複数ある場合，それぞれのルールの支持度のうち最大のものを用いる．

確信度　修正候補が提示される元となった相関ルールの確信度を示す．そのようなルールが複数ある場合，それぞれのルールの確信度のうち最大のものを用いる．

ルール数　修正候補が提示される元となった相関ルールの個数を示す．

4. 評 価 実 験

本研究では，提案手法の有効性を評価するために複数のオープンソース・ソフトウェアを対象として実験を行った．本章では実験内容とその結果について述べる．実験対象としたソフトウェアの一覧と，実験に使用したリビジョンのうちソースファイルに修正が行われたリビジョン数を表 1 に記述する．行った調査の内容は以下の通りである．

調査項目 1: 修正候補の推薦精度の調査

行われるべき修正の一部が欠けているコミットに対して，提案手法を用いることで欠けている修正を推薦することができるかを調査する．また，複数の修正候補が推薦された時に正しい候補を何番目に推薦できるのかを調査する．

調査項目 2: 修正漏れの調査

提案手法をコミットに対して適用し，修正漏れ候補を検出する．検出された修正漏れ候補が本当に修正漏れであるかどうかを手動で調べることで，提案手法が実際に有効であるかどうかを確認する．

4.1 調 査 方 法

本節では，各調査をどのように行うかについて述べる．

調査項目 1: 修正候補の推薦精度の調査

前処理として，対象としているコミットにおいて，以前のコミットで一度も出現していない微小修正が出現した場合はそれを除外する．コミット中の微小修正を 1 つ削除し，推薦すべき正解の微小修正とする．そのコミットに対して提案手法を適用し，推薦候補中に正解の微小修正が含まれるかどうかを調べる．この操作を全てのコミットの全ての微小修正に対して行い，正解の微小修正が推薦候補の中に含まれる割合を計算する．また，正解の微小修正が推薦された場合に，修正候補の並べ替えを行った際，正解の微小修正を何番目に提示できたかを調べる．なお，学習データは対象コミット以前の全てのコミットとする．

表 1　実験対象ソフトウェア

ソフトウェア名	開発言語	ソースファイルに修正が行われたリビジョン数
gcc	C	18,914
FreeBSD	C	13,154
ArgoUML	Java	11,738
jEdit	Java	9,930

調査項目 2: 修正漏れの調査

コミットに対して提案手法を用いたとき，相関ルール $x \Rightarrow y$ に基づいて微小修正 y が推薦されたとする．ここで，微小修正 y がそのコミットに含まれていない場合，y を**修正漏れ候補**と定義する．修正漏れ候補は以下の 2 種類に分類される．

閉じた修正漏れ候補:　後に y が行われているが x が行われていないというコミットが存在する

開いた修正漏れ候補:　後に y が行われているが x が行われていないというコミットが存在しない

閉じた修正漏れとは，開発者が修正漏れに気づき，後のコミットで修正されたものと定義する．また，開いた修正漏れとは，開発者が修正漏れに気づかず，そのままになっているものと定義する．

jEdit と ArgoUML から閉じた修正漏れ候補を無作為に 30 個ずつ選び，それらを手動で調べ，以下の 3 種類に分類した．

- 重要な修正漏れ（修正漏れによりプログラムの動作が変わる）
- 重要でない修正漏れ（修正漏れによりプログラムの動作が変わらない）
- 修正漏れでない

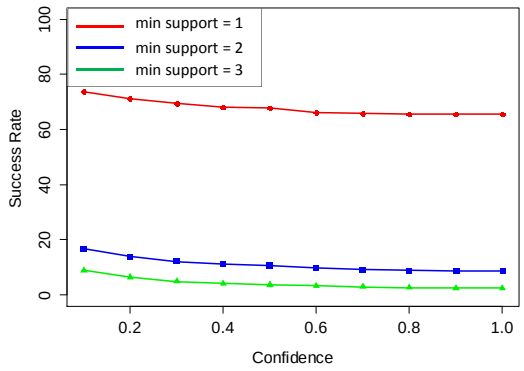
これらの分類は，ソースコードとコミットメッセージから判断する．また，相関ルールの最低支持度は 2，最低確信度は 1 とした．

4.2 実 験 結 果

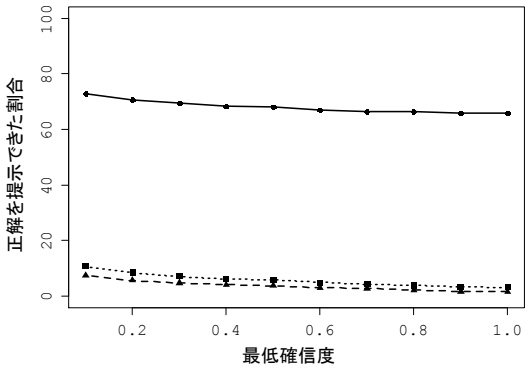
調査項目 1: 修正候補の推薦精度の調査

まず，コミットで行われた微小修正のうちの 1 つを削除し，行われるべき微小修正が 1 つだけ欠けたコミットを作る．そして，削除された微小修正を推薦できるかどうかを調べることで提案手法を評価する．削除された微小修正を正解とし，正解を提示できた割合をプロットしたものが図 2 である．縦軸が正解を提示できた割合の百分率を，横軸が確信度を示しており，最低支持度が 1 の場合，2 の場合，3 の場合でそれぞれプロットしている．図 2 より，最低支持度が 1 の場合は FreeBSD と jEdit において，65%以上の割合で正解を提示できていることが分かる．一方，他の 2 つのソフトウェアについては FreeBSD と jEdit に比べると低い値になっている．最低支持度が 2 以上の場合は，正解を提示できた割合はソフトウェアによって大きな差はなかった．

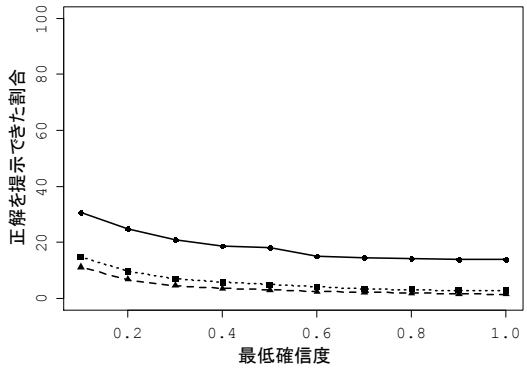
次に，正解の微小修正が提示できた場合において，支持度が 1 または 2 のときに正解が上位 1 位，3 位，5 位に提示された割合を表 2 と表 3 に示す．ここでは代表として jEdit と ArgoUML での結果を示す．相関ルールの最低確信度は 1 とする．候補の並べ替えについては，まずルール数が高いものを上位にし，ルール数が等しい候補に関しては支持度が高いものを上位にし



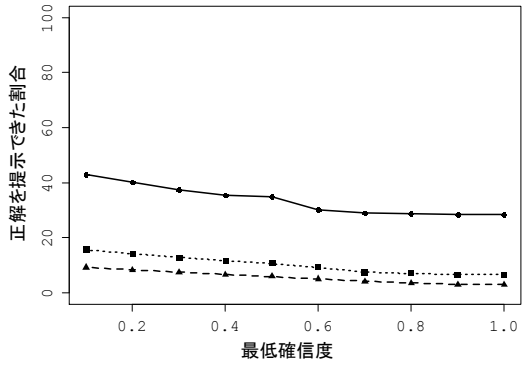
(a) FreeBSD



(b) jEdit



(c) gcc



(d) ArgoUML

図 2　正解を提示できた割合

ている．なお，最低確信度を 1 にしているため．候補の確信度は 1 で固定となり並べ替えに用いることはできない．並べ替えを行わない場合の候補の順位はプログラムの実行に依存する．

表 2 と表 3 より，全ての場合において支持度が高ければ高いほど上位に提示される候補が多くなっていることが分かる．また，候補の並べ替えを行うことで上位に提示される割合が増えている．

調査項目 2: 修正漏れの調査

調査項目 2 の結果について述べる．まず，各ソフトウェアから見つかった修正漏れ候補の数を表 4 に示す．ArgoUML では多数の修正漏れ候補が見つかったが，閉じている修正漏れ候補は全体の 10%程度だった．

次に，閉じた修正漏れ候補を手動で分類した結果を表 5 に示す．修正漏れ候補の中で実際に修正漏れであるものは約半分程度であった．

jEdit において実際に見つかった修正漏れの例を図 3 に示す．リビジョン 7,061 に対するコミットで，MirrorList.java において文 (a) の追加が行われており，それに対して提案手法が文 (b) の追加を推薦したが，そのコミットでは文 (b) の追加は行われなかった．

しかし，その 4 日後のリビジョン 7,109 に対するコミットで，ProjectPlugin.java において文 (b) の追加が行われた．このコミットでのコミットログには“work around broken JAXP config in Java 1.4”という文が含まれていた．JAXP とは Java API for XML Processing の略である．文 (a) の追加は，XML のパースに XMLReader クラスを使うようにするという変更に伴う修正である．XMLReaderFactory は SAX という XML の

表 2　正解が推薦された順位 (jEdit)

最低支持度	1			2		
上位 n 位	1	3	5	1	3	5
並べ替え無し	6.4%	18.6%	28.1%	47.0%	73.6%	64.6%
並べ替え有り	26.2%	54.6%	71.1%	56.3%	80.8%	89.2%

表 3　正解が推薦された順位 (ArgoUML)

最低支持度	1			2		
上位 n 位	1	3	5	1	3	5
並べ替え無し	9.4 %	19.8%	27.0%	30.3%	51.7%	64.4%
並べ替え有り	20.6%	36.5%	44.2%	45.2%	68.8%	77.4%

表 4　修正漏れ候補の数

ソフトウェア名	閉じた修正漏れ候補	開いた修正漏れ候補	合計
jEdit	130	63	193
ArgoUML	368	2,656	3,024

表 5　修正漏れ候補の分類

ソフトウェア名	重要な修正漏れ	重要でない修正漏れ	修正漏れでない	合計
jEdit	7	3	20	30
ArgoUML	1	19	10	30

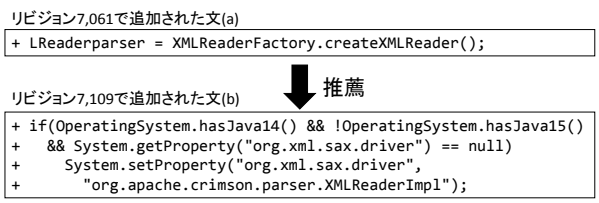


図 3　修正漏れの例

パースのための API に含まれるクラスである．Java1.4 では，XMLReaderFactory の初期設定を行う処理が必要であり，推薦された修正はその初期化処理である．よって，この例は XML に関する処理を追加したが，初期化処理を行い忘れているという修正漏れであると考えられ，提案手法を用いていれば，この修正漏れを未然に防ぐことができたと考えられる．

5. 議　　論

5.1 正解が提示された順位について

相関ルールの最低支持度を上げると，推薦される候補の数が少なくなるため正解の候補が提示されない確率が高くなるが，正解の候補が提示できた場合に，正解の候補が上位に提示される確率も高くなる．そのため，最低支持度が 2 以上の場合は並べ替えを行わなくても上位に正解が提示されやすくなっている．また，表 2 と 3 より，並べ替えは最低支持度が 1 の場合に高い効果があることが分かる．両ソフトウェアにおいて最低支持度が 2 の場合は並び替えを行うことで約 70%以上の確率で上位 3 位までに正解を提示できている．これは十分に高い確率であると考えられる．並べ替えの方法について，最低確信度が高い場合においては，まずルール数，次に支持度，3 番目に確信度という優先順位で並べ替えるのが最も結果が良かったため，今回はこれを採用した．

5.2 提案手法の問題点

提案手法の問題点として，修正の内容は提示できるが，修正の位置までは提示できないということが挙げられる．修正内容が削除または変更である場合は，修正の対象となる文の列がソースコード中に存在するため，それと一致する部分に修正箇所を絞ることが可能である．しかし，修正内容が追加の場合，修正箇所を絞ることができないため，プログラムの構造を把握していない開発者でも修正を行えるという利点が弱まってしまう．この問題を解決するために以下の様な方法が考えられる．

既存手法と組み合わせる　既存手法 [3] を用いることで，修正が行われたファイルやメソッドに対してそれと同時に修正されやすいファイルやメソッドを提示することができる．提案手法と既存手法を組み合わせることで，提示された修正が行われそうなファイルやメソッドに当たりをつけることが可能になる．

修正が行われた文脈を考慮する　提案手法で提示される修正は必ず過去に一度行われたものである．よって，その修正が過去に行われたファイルやメソッド，前後の文の情報を用いることで修正箇所を特定することができる可能性がある．例えば，P という文の追加が推薦され，過去のコミットで，A という文と B という文の間の行に P の追加が行われていたとする．このとき，編集中のソースコード中に AB という文の並びがあった場合，同じく AB の間の行に P が追加される可能性が高いと予測できる．

6. おわりに

本研究では，開発者が修正を行った際，それと同時に行われるべき修正の内容を提示する手法を提案した．この手法では，プログラム文レベルの相関ルールを作成することで，従来手法では提示することができなかった具体的な修正の内容を提示している．よって，提案手法を用いることで，今までは防ぐことができなかった修正漏れを防ぐことが可能である．

本研究では提案手法の有効性を評価するため，4 つのオープンソース・ソフトウェアを対象に実験を行った．その結果，プログラム文レベルの相関ルールを用いることが効果的であり，これを用いることで高い精度で正しい修正候補を推薦できることを示した．また，ソフトウェアの開発中に実際に起こった修正漏れを提案手法を利用して見つけることができるかを調査した．その結果，提案手法によって自動的に検出できた修正漏れ候補の約半数が実際に修正漏れであった．

今後の課題として，修正の内容だけでなく，修正の位置を特定できるようにすることを考えている．特に修正の内容が文の追加であった場合，追加する文の内容が分かってもどこに追加するかを特定することが難しい．これを解決するために，既存手法と組み合わせることや修正が行われた文脈を考慮するという方法が考えられる．

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：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] B. Ray, Miryung Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 367–377, Nov 2013.

[2] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, Vol. 30, No. 9, pp. 574–586, Sep 2004.

[3] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pp. 563–572, 2004.

[4] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pp. 42–46, 2008.

[5] A. Michail. Data mining library reuse patterns using generalized association rules. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pp. 167–176, 2000.

[6] 泉田聡介, 植田泰士, 神谷年洋, 楠本真二, 井上克郎. ソフトウェア保守のための類似コード片検索ツール. 電子情報通信学会論文誌, Vol. 86, No. 12, pp. 906–908, Dec 2003.

[7] 岡田孝, 元田浩. 相関ルールとその周辺. オペレーションズ・リサーチ：経営の科学, Vol. 47, No. 9, pp. 565–571, Sep 2002.