修士学位論文

題目

相関ルールを用いたソースコード修正候補の推薦

指導教員
楠本 真二 教授

報告者
切貫 弘之

平成 27 年 2 月 6 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 26 年度 修士学位論文

相関ルールを用いたソースコード修正候補の推薦

切貫 弘之

内容梗概

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，ソフトウェアリポジトリ中の修正履歴を分析し，同時に修正されやすいファイルやメソッドを明らかにする研究が行われている．それらの研究は，開発者がファイルやメソッドに修正を加えた際，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ことを目的の 1 つとしている．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，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．提案手法を評価するために，複数のオープンソース・ソフトウェアに対して調査を行ったところ，過去に見落とされた多数の修正漏れを発見することができた．

主な用語

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

目次

1 まえがき **1**

2 準備 **3**

2.1 バージョン管理システム 3

2.2 修正漏れ 4

2.3 相関ルール 4

3 研究動機 **6**

3.1 ロジカルカップリング 6

3.2 類似するコード片に対する修正漏れ防止 7

4 提案手法 **8**

4.1 修正履歴情報のデータベース化 8

4.2 相関ルールの作成 12

4.3 修正候補の提示 13

5 評価実験 **16**

5.1 調査方法 16

5.1.1 調査項目 1 : 相関ルールの適合率・再現率の調査 17

5.1.2 調査項目 2: 修正候補の推薦精度の調査 18

5.1.3 調査項目 3: オープンソース・ソフトウェア中の修正漏れの調査 19

5.2 実験結果 20

5.2.1 調査項目 1 : 相関ルールの適合率・再現率の調査 20

5.2.2 調査項目 2: 修正候補の推薦精度の調査 21

5.2.3 調査項目 3: オープンソース・ソフトウェア中の修正漏れの調査 23

6 議論 **25**

6.1 パラメータの閾値と適合率・再現率の差について 25

6.2 提案手法が開発のどの時点でも有効であるか 26

6.3 正解が提示された順位について 26

6.4 実際に見つかった修正漏れ 27

6.5 より精度を上げる方法について 27

6.6 正規化の是非 27

6.7 提案手法の問題点 28

6.8 妥当性への脅威 28

7　おわりに **30**

謝辞 **31**

参考文献 **32**

図目次

1 バージョン管理システムのイメージ 3
2 提案手法の概要 9
3 修正の抽出 10
4 最長共通部分列を用いた微小修正の特定 12
5 相関ルールの適合率と再現率 20
6 正解を提示できた割合 21
7 正解が何番目に提示されたか 22
8 適合率・再現率の推移 25

表 目 次

1 トランザクションとアイテムの例 5
2 表 1 から作成される相関ルール 6
3 修正候補の提示と並べ替えの説明に用いる例 15
4 実験対象ソフトウェア 16
5 ソースファイルに修正が行われたリビジョンの数 18
6 修正漏れ候補の数 23
7 修正漏れ候補の分類 24
8 2 回以上共起した微小修正の組の割合 25
9 正解が推薦された順位 (jEdit) 26
10 正解が推薦された順位 (ArgoUML) 27
11 出現回数の少なかった微小修正 28

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, 3, 4, 5, 6, 7]．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防止や修正の推薦以外にも，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 [8]．修正漏れ防止や修正の推薦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 [9]．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，文の追加・削除を含んだ修正や類似していないコード片に対する修正に対応することができない．ソフトウェア開発においては内容の異なる修正が同時に行われることも多く，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，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．

提案手法において，相関ルールを作成するためには，プロジェクトにおいて過去にどのような修正がなされたかという情報を解析する必要がある．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ディレクトリをリポジトリと呼び，リポジトリを解析することでソフトウェア開発に有益な情報を得ることができる [10, 11, 12, 13, 14]．このような解析作業をリポジトリマイニングと呼ぶ．リポジトリ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．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，ソースコードの修正に関するものも多い [15, 16, 17].

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．

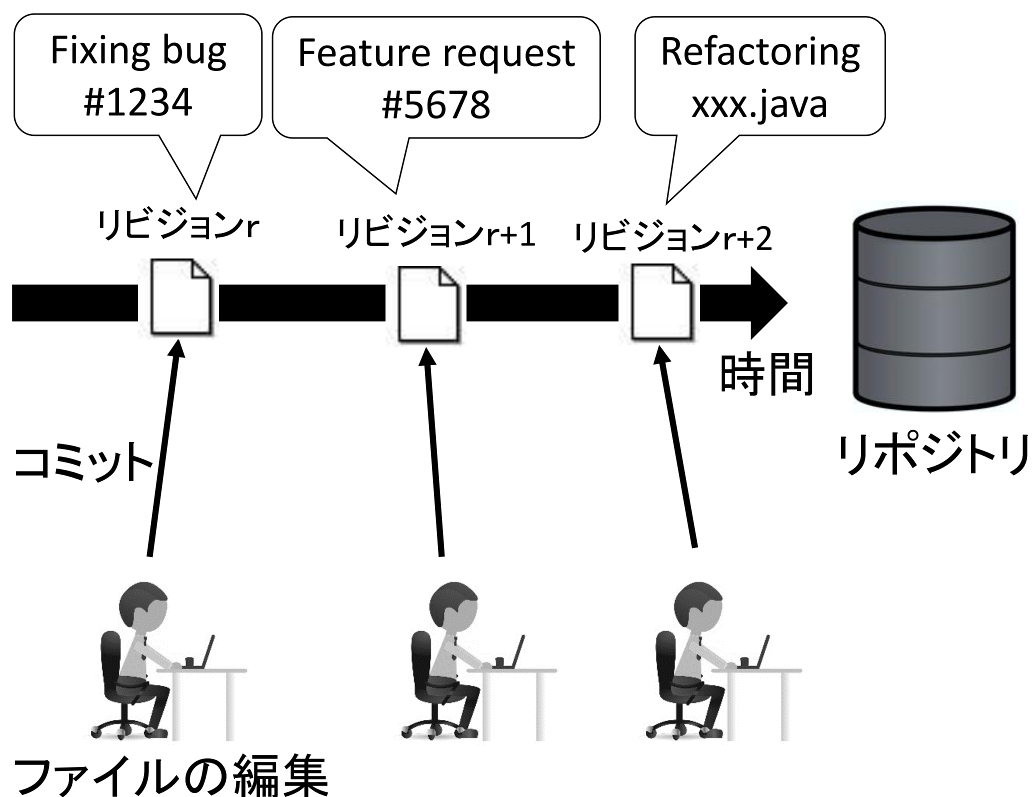


図 1: バージョン管理システムのイメージ

2　準備

本章では，本研究における前提知識となる用語の説明等を行う．

2.1　バージョン管理システム

ソフトウェアを開発する際は，バージョン管理システムが用いられることが多い．バージョン管理システムとは，CVS，Subversion，git に代表されるコンピュータ上で作成，編集されたファイルの修正履歴を管理するためのシステムである [18, 19, 20]．特にソフトウェア開発では，ソースコードの管理に用いられることが多い．バージョン管理システムにおいて，修正履歴を保存しておくディレクトリをリポジトリと呼び，リポジトリに対して編集したファイルの変更情報を保管することをコミットという．開発者は，ファイルを編集してそれをコミットするという作業を繰り返すことでソフトウェア開発を行う．ソフトウェアの細かい修正を表す改訂番号をリビジョンと呼び，コミットを行うことでリビジョンの番号が 1 つ増える．バージョン管理システムのイメージを図 1 に表す．

2.2　修正漏れ

修正漏れは，開発者が行うべき修正の一部を行わなかった際に発生する．修正漏れが発生する主な原因の 1 つとしてコードクローンの存在がある．コードクローンとは，ソースコード中に存在する同一，または類似するコード片のことである [21]．互いにコードクローンであるコード片の集合をクローンセットと呼ぶ．クローンセットの要素であるコード片の 1 つに対して修正が行われるとき，クローンセットに含まれる他のコード片にも同様の修正が必要とされる場合が多い．このとき，開発者がコードクローンの存在を認識していない場合，コードクローンの一部にしか修正が行われずに修正漏れとなってしまう．また，その他の修正漏れとして，新たに open したファイルの close 忘れ，新たに確保したメモリの開放忘れ，新たに宣言した変数の初期化忘れなどが考えられる．

2.3　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．一般的に以下のように表される [22].

$$X \Rightarrow Y$$

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．

相関ルールを用いた分析は，商店における商品の購買の傾向を知るためによく用いられる [23]．この場合は，商店で売られている各商品がアイテム，一人の顧客が一度に購買したアイテムの集合がトランザクションとなる．複数のトランザクションを解析することで,「パンを買った顧客は 80 %の確率で牛乳を買う」というような知見を得ることができる．相関ルールを用いて購買の傾向を知ることで，同時に売れやすい商品を近くに配置したり，ネットショップにおいてある商品を購入した人に別の商品を推薦するといったビジネス上有益な戦略を練ることができる．

各ルールの重要度を評価するための指標として，支持度 (support)・確信度 (confidence) 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 を同時に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 を同時に含む頻度，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同時に存在する確率が高く，ルールの重要度が高い．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同時に存在したために作成された可能性が高いといえる．一般的には (1) のように定義されることが多いが，本論文では簡単にす

るために (2) のように定義する．(2) のように定義したとき，支持度は 1 以上の整数値となる．

$$supp(X \Rightarrow Y) = \frac{freq(X, Y)}{N} \tag{1}$$

$$supp(X \Rightarrow Y) = freq(X, Y) \tag{2}$$

本研究では複数のトランザクション集合を同時に用いないため，(2) のように定義しても問題がない．

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．

$$conf(X \Rightarrow Y) = \frac{freq(X, Y)}{freq(Y)}$$

相関ルールの作成について，例を用いて説明する．例として，表 1 のようなアイテムが各トランザクションに含まれているとする．相関ルールは，各トランザクション中の全てのアイテムの組み合わせについて作成される．よって，表 1 の 3 つのコミットから作成される全相関ルールは表 2 のようになる．表 2 の右側のルールは左側のルールの条件部と帰結部を入れ替えたものになっている．まず，アイテム a と b を共に含むトランザクションが存在するため，$a \Rightarrow b$ というルールが作成できる．a と b を共に含むトランザクションは 1 番と 3 番の 2 つであるため，$a \Rightarrow b$ の支持度は 2 となる．また，a を含むトランザクションでは必ず b も含まれるため，$a \Rightarrow b$ の確信度は 1 となる．同じく $b \Rightarrow a$ というルールも作成できる．$b \Rightarrow a$ の支持度は $a \Rightarrow b$ と同じく 2 になるが，b を含む 3 つのトランザクションのうち，a も含んでいるのは 2 つだけであるため，確信度は 0.67 となる．他のアイテムの組み合わせについても同様に相関ルールを作成する．

表 1: トランザクションとアイテムの例

トランザクション	アイテム
1	$\{a, b, c\}$
2	$\{b, c\}$
3	$\{a, b, d\}$

3 研究動機

修正漏れを予防するための研究は今までにいくつか行われている．本章ではそれらを紹介した後，提案手法がどのように既存手法における問題点を解決できるかについて説明する．

3.1 ロジカルカップリング

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修正されやすいという論理的な結び付きのことである．ここでいうモジュールとは，ファイルやメソッド等のまとまりのある機能を持った部品のことである．ロジカルカップリングを利用して修正漏れを防止する研究について紹介する．Ying らはファイル単位でのロジカルカップリングを検出する手法を提案している [2]．Ying らの手法を用いることで，プロジェクト中で同時に修正されやすいファイルを把握することができる．

また，Zimmermann らは，より細かな粒度でのロジカルカップリングを検出する手法を提案している [3]．Zimmermann らの手法では，リポジトリ中のソースコードについてはメソッド・フィールド単位で，ソースコード以外のファイルについてはファイル単位でロジカルカップリングを検出している．さらに，ロジカルカップリングを利用した修正候補推薦システムを Eclipse のプラグインとして実装している．これにより，開発中のソフトウェアにおいても，修正が行われそうなメソッドやフィールドを開発者に提示することができるようになり，修正漏れが起こるのを防ぐ助けとなっている．

しかし，これらの研究は，修正を行うべきファイルやメソッドは特定できても具体的な修正の内容までは特定することができないという問題点がある．提示されたメソッドに対して行うべき修正を把握している開発者ならば，メソッドレベルの推薦でも適切な修正を行える可能性があるが，そうでない開発者にとってはメソッドレベルの推薦では充分でない恐れがある．また，メソッドの行数が多い場合も，修正を行うべき箇所を特定することが難しく，適切な修正を行うのは困難であると思われる．

表 2: 表 1 から作成される相関ルール

相関ルール	支持度	確信度	相関ルール	支持度	確信度
$a \Rightarrow b$	2	1	$b \Rightarrow a$	2	0.67
$b \Rightarrow c$	1	0.5	$c \Rightarrow b$	1	1.0
$b \Rightarrow c$	2	0.67	$c \Rightarrow b$	2	1.0
$a \Rightarrow d$	1	0.5	$d \Rightarrow a$	1	1.0
$b \Rightarrow d$	1	0.33	$d \Rightarrow b$	1	1.0

3.2　類似するコード片に対する修正漏れ防止

類似するコード片に対しての修正漏れを防ぐことを目的とした研究を紹介する．代表的なものに，コードクローン検出の研究が挙げられる．コードクローンについては，2.2 節で簡単に説明している．プログラム中のコードクローンの存在はソースコードの保守性を低下させる要因となる．なぜなら，クローンセット中のコード片に対して修正を行う際，しばしばクローンセット中の他のコード片に対しても同様の修正を行う必要があり，それが修正漏れにつながる恐れがあるためである．泉田らは，オープンソース・ソフトウェアのバッファオーバーフローの修正を題材に，修正が必要なコード片の見逃し防止策として，grep とコードクローン検出ツールが有効であることを示した [9]．コードクローンの位置を把握することで，クローンセットに対し同様の修正を行う際の修正漏れを防止することができる．

また，コードクローン検出では，コピーアンドペースト後の変数名の置換漏れなどの軽微な修正漏れを検出できるが，文の追加・削除などの修正漏れを検出するのは難しいという問題がある．そこで，Higo らはリポジトリをマイニングすることでコード片レベルでの修正漏れを検出する手法を提案した [24]．Higo らの手法は，修正履歴からどのようなコード片がどのように変更されているのかを学習することで，文の追加・削除などのパターンも学習することができ，コードクローン検出を用いた手法での問題点を解決している．しかし，Higo らの手法もコードクローン検出を用いた手法と同様に，類似していないコード片に対してそれぞれ異なる修正を行う際に起こる修正漏れを検出することができないという問題がある．

Meng らは，内容や文脈が完全に同一ではないが広い範囲にわたる修正が存在することに着目し，ソースコード自動修正ツール LASE を開発した [25]．LASE は 2 つ以上の類似する修正の例を与えることで，同様の修正が必要と思われる箇所を特定し，自動で修正を行うことができる．LASE の特徴は，修正が完全に同一でなくても，複数の修正例において共通する文脈を考慮し，自動的に修正を行うことができる点である．LASE の問題点として，あらかじめ開発者が修正の例を入力として与える必要がある上，文脈や修正内容に何らかの共通部分がなければ使うことができない点が挙げられる．

本節で挙げた既存研究に共通する問題点は，完全に同一，または類似する修正でなければ推薦できないという点にある．しかし，修正作業を行う上で，同一の内容ではないが同時に行われるべき修正も存在する．例えば，変数宣言の追加とその初期化処理や，ファイルの open と close 等がそれにあたる．そのような同一の内容ではないが同時に行うべき修正の修正漏れを防ぐことができれば開発者にとって有益であると考える．そこで，本論文では，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する．提案手法を用いることで，従来手法では検出することができないような修正漏れを検出することが可能である．

4　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．図 2 は提案手法全体の流れを示している．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

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 k から $k+1$ 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．

4.1　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 $r+1(1 \leq r \leq n)$ 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 3 は連続する 2 つのリビジョン間で行われた修正を抽出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．

STEPA-1: ソースコード中の文の検出

STEPA-2: リビジョン間で行われた修正の抽出

STEPA-3: 識別子の正規化

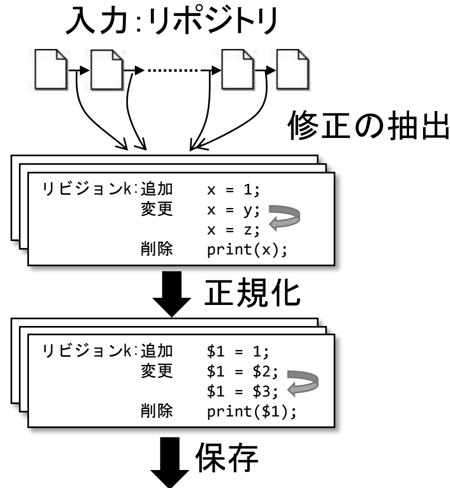
STEPA-4: 正規化された修正情報の保存

以降では，それぞれのステップについて詳細に説明する．

STEPA-1: ソースコード中の文の検出

リビジョン r から $r+1(1 \leq r \leq n)$ 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．

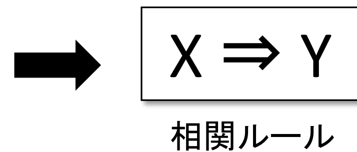
A. 過去の修正情報の抽出



コミット	行われた修正
1	{修正A，修正B，修正C，修正D}
2	{修正A，修正C，修正E，修正F}
…	…
k	{修正A，修正B，修正F}
…	…

データベース

B. 相関ルールの作成



相関ルール

適用

C. 修正候補の提示

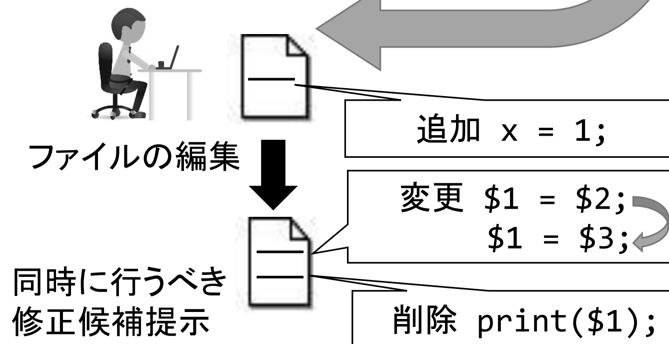


図 2: 提案手法の概要

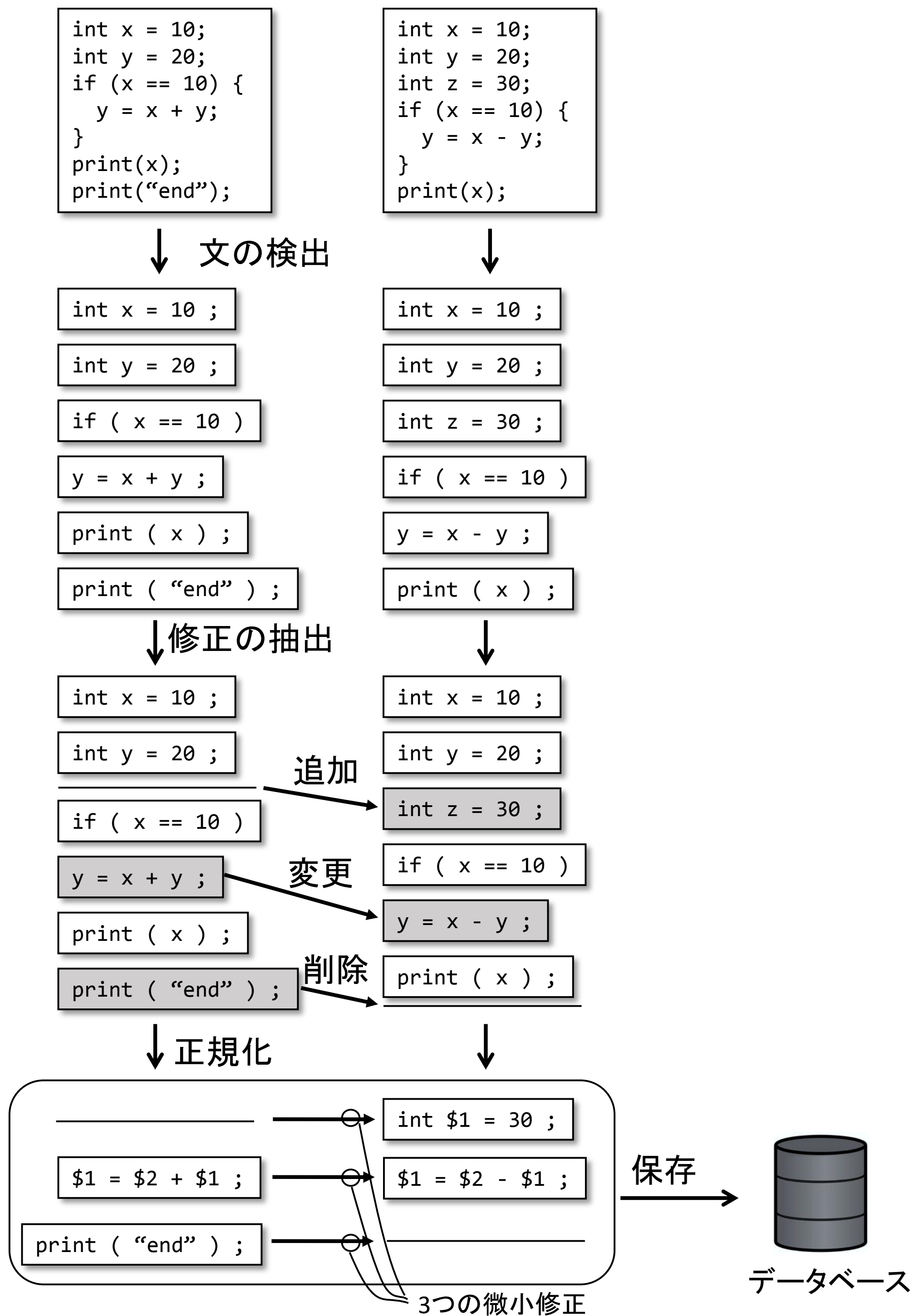


図 3: 修正の抽出

STEPA-2: 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 $common_k (1 \leq k \leq m)$ 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 $before_k (0 \leq k \leq m)$ を以下のように定義する．

$before_0$**:** $common_1$ の前に来る文の列

$before_k (1 \leq k \leq m-1)$**:** $common_k$ 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 $after_k (0 \leq k \leq m)$ 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:** $before_k \neq \emptyset \wedge after_k = \emptyset$

例として，以下のような文の並び α，β があるとする．

$$\alpha = (A, B, C, D, E, F, G)$$

$$\beta = (A, P, Q, C, R, S, D, E, G)$$

もし，α が修正前，β 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．図 4 はその様子を表したものである．

- (B) からの (P,Q) への変更
- (Q,S) の追加
- (F) の削除

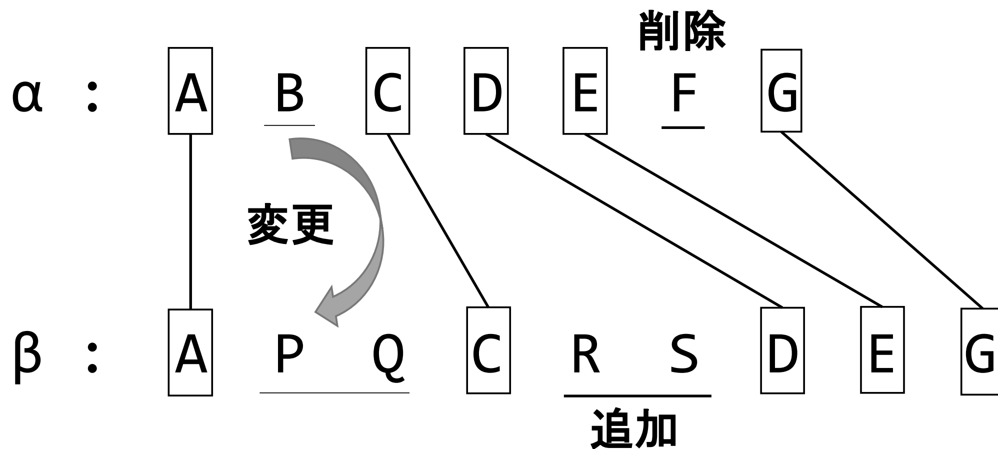


図 4: 最長共通部分列を用いた微小修正の特定

STEPA-3: 識別子の正規化

このステップでは STEPA-2 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．例えば，p = p.getX() + 1; という文は $0 = $0.getX() + 1 という文に変更される．なお，微小修正の比較は，修正された文が正規化された後の文字列を比較することで行う．

STEPA-4: 正規化された修正情報の保存

最後に，STEPA-3 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 $r + 1(1 \leq r \leq n)$ の間で行われた 1 つ以上の微小修正がデータベースに保存される．

STEPA-1〜STEPA-4 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．

4.2 相関ルールの作成

本節では，データベース中の修正履歴情報から相関ルールを作成する処理について説明する．

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．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．コミットが n 回行われたとき，k 番目のコミットで行われた微小修正の数を a_k とすると，データベース全体から相関ルールを作成した場合のルールの数は以下のように表される．

$$\sum_{k=1}^{n} {}_{a_k}P_2$$

各相関ルールは評価指標として，支持度と確信度を持つ．

作成されたルールの例を以下に示す．行頭の−は削除された行を，+は追加された行を示している．

条件部

```
-  lastIndent = getScreenLineEndOffset(screenLine) - 1;
+  lastIndent = getLineStartOffset(line) + lineInfos[subregion].offset
                              + lineInfos[subregion].length - 1;
```

帰結部

```
-  offset = caret;
+  offset = caret - buffer.getLineStartOffset(line);
```

このルールは，変数 lastIndent の計算方法が変更されたとき，同時に変数 offset の計算方法も変更されるというルールを示している．この例は，実際にオープンソース・ソフトウェア jEdit において作成されたルールで，支持度は 2，確信度は 1 である．

4.3 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．

STEPB-1: 最新リビジョンに対して行われた修正の抽出

STEPB-2: 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．

STEPB-1: 最新リビジョンに対して行われた修正の抽出

4.1 節の STEPA-1〜STEPA-3 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，4.1 節の STEPA-1 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，STEPA-2 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，STEPA-3 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．

STEPB-2: 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，信頼度の高い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，信頼度の高い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．

修正候補の提示と並べ替えについて，例を用いて説明する．最新リビジョンに対して $x_1, ..., x_4$ という 4 つの微小修正が行われているとする．表 3 の (a) は有効なルールと支持度・確信度の一覧を示しており，(b) は提示される修正候補と並べ替えに用いられる各種指標の一覧を示している．例では，$x_1, ..., x_4$ のうちのいずれかを条件部に持つルールが 4 つあり，それらの帰結部は，y_1, y_2, y_3 のいずれかである．よって，開発者には y_1, y_2, y_3 の 3 つが修正候補として提示される．ここで，y_1 は推薦の元となるルールが 2 つ存在しているので，y_1 のルール数は 2 となる．また，支持度と確信度はそれぞれのルールで大きい方を取るので，支持度は 4，確信度は 0.67 となる．y_2, y_3 については，それらを推薦する元となるルールが 1 つだけなので，ルール数は 1 で，支持度・確信度は元となるルールの値をそのまま用いる．並べ替えに用いる指標によって提示される順番が変化する場合がある．例

えば，y_1, y_2, y_3 を支持度を用いて並べ替えを行った場合，$y_3 \to y_1 \to y_2$ の順に提示され，確信度を用いて並べ替えを行った場合，$y_2 \to y_3 \to y_1$ の順に提示される．

また，並べ替えに複数の指標を用いることもできる．ある指標について同じ値を持つ候補があった場合に別の指標を用いて順位をつけることが可能である．例えば，表 3 の例では，ルール数を用いて並べ替えを行うと y_2 と y_3 に順位をつけることができないが，次に支持度もしくは確信度を用いて並べ替えを行うことで順位をつけることができる．支持度・確信度・ルール数が全て同じものに関しては，順位をつけることができないので，どの候補が上位に提示されるかはプログラムの実行に依存する．

表 3: 修正候補の提示と並べ替えの説明に用いる例

(a) 相関ルールと支持度・確信度

相関ルール	支持度	確信度
$x_1 \Rightarrow y_1$	4	0.60
$x_2 \Rightarrow y_1$	3	0.67
$x_3 \Rightarrow y_2$	2	1.0
$x_4 \Rightarrow y_3$	5	0.8

(b) 提示される修正候補と各種指標

修正候補	支持度	確信度	ルール数
y_1	4	0.67	2
y_2	2	1.0	1
y_3	5	0.8	1

5　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を表 4 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: 相関ルールの適合率・再現率の調査
提案手法では，リポジトリ中の修正履歴情報を学習データとし，微小修正間の相関ルールを作成している．学習データとしているリポジトリに対して新たなコミットが行われたと仮定し，そのコミットで行われた修正はどの程度相関ルールに従うのかを調査する．ルールの適合率・再現率が高ければ高いほど，そのルールは信頼できるといえる．様々な閾値を用いて適合率・再現率を調べることで，閾値をどの値に設定した場合，信頼できる相関ルールが作成されるのかを調べる．

調査項目 2: 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 3: オープンソース・ソフトウェア中の修正漏れの調査
提案手法を実際のオープンソース・ソフトウェアに対して適用し，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．

5.1　調査方法

本節では，5 章に記述した各調査をどのように行うかについて述べる．

表 4: 実験対象ソフトウェア

ソフトウェア名	開発言語	総リビジョン数	開始リビジョン	終了リビジョン
gcc	C	130,804	1988/11/23	2007/12/03
FreeBSD	C	230,286	1993/06/12	2012/01/18
ArgoUML	Java	19,895	1998/01/27	2012/07/19
jEdit	Java	22,016	2006/07/01	2012/08/17

Algorithm 1 調査項目 1 の調査方法

入力: リポジトリ
出力: TP, FP
  $TP \leftarrow 0, FP \leftarrow 0$
  for $i = 0$ to n **do**
    M_i 中の初出の微小修正を除外
    for all $x \in M_i$ **do**
      for all $y \in M_k, y \neq x$ **do**
        if $\exists x \Rightarrow y \in R$ **then**
          $TP \leftarrow TP + 1$
        else
          $FP \leftarrow FP + 1$
        end if
      end for
    end for
    M_i を学習データに加え，相関ルールを更新
  end for

5.1.1　調査項目 1：相関ルールの適合率・再現率の調査

学習データに含まれないコミットを対象とし，相関ルールに従う微小修正の組がどの程度存在するかを調べることで相関ルールの適合率・再現率を求める．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．

調査の手順を説明する．まず，対象としているコミットで行われた微小修正の集合から要素を 2 つ選んで，一方をルールの条件部，もう一方を帰結部とみなす．コミットで行われた微小修正の数が k のとき，そのような組み合わせの数は全部で ${}_kP_2$ である．このような条件部と帰結部の組み合わせについて，それと等しい相関ルールが存在するかどうかを調べる．そのようなルールが存在すれば，その組み合わせはルールに従っているとして True Positive（以下，TP）, 行われていなければ False Positive（以下，FP）と判定する．この操作をリポジトリ中の全コミットに対して行い，全 TP 数，全 FP 数，各コミットで作りうる全組み合わせ数の合計を求める．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．

詳細な手順を Algorithm1 に示す．ここではリポジトリ中に n 個のコミットが存在しているとし，i 番目のコミットで行われた微小修正の集合を M_i，M_i 中の初出でない微小修正の数を k_i とする．また，作成された相関ルールの集合を R とする．このとき，コミットにおける条件部と帰結部の全組み合わせ数は ${}_{k_i}P_2$ になる．

ここまでの操作で求めた TP の数，FP の数を用いて適合率と再現率を計算する．適合率と再現率

の計算式を以下に示す．

$$適合率 = \frac{TP の数}{TP の数 + FP の数}$$

$$再現率 = \frac{TP の数}{\sum_{i=1}^{n} k_i P_2}$$

適合率は推薦された候補のうちの正解の候補の割合を示しており，再現率は推薦されるべき候補のうちの推薦できた候補の割合を示している．これらの値が高ければ高いほど，相関ルールの信頼性が高いといえる．

調査は全てのソフトウェアに対して行うが，gcc と FreeBSD に関しては実行時間の都合上 1〜60,000 リビジョンまでを対象とした．対象としたコミットのうち，ソースファイルに修正が行われたリビジョンの数を表 5 に示す．また，相関ルールのフィルタリングに関しては，最低支持度 1 から 3 まで，最低確信度を 1 から 0.1 まで 0.1 刻みで変化させてそれぞれ測定した．

5.1.2　調査項目 2: 修正候補の推薦精度の調査

コミットで行われた微小修正のうちの 1 つを削除し，相関ルールに基づいて，残りの微小修正から削除された微小修正を推薦でき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，削除された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，削除された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，削除された微小修正を何番目に提示できるかを調べる．

手順を説明する．調査項目 1 と同じく，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 つ選択して削除し，推薦すべき正解の微小修正とする．作成された相関ルールの中に，正解の微小修正を帰結部とするようなものが存在し，かつそのルールの条件部の微小修正がコミットに含まれていれば正解の微小修正が推薦可能である．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．

詳細な手順を Algorithm2 に示す．ここではリポジトリ中に n 個のコミットが存在しているとし，i 番目のコミットで行われた微小修正の集合を M_i，M_i 中の初出でない微小修正の数を k_i とする．ま

表 5: ソースファイルに修正が行われたリビジョンの数

ソフトウェア名	リビジョン数
gcc	18,914
FreeBSD	13,154
ArgoUML	11,738
jEdit	9,930

Algorithm 2 調査項目 2 の調査方法

入力: リポジトリ

出力: C

$C \leftarrow 0$

for $i = 0$ to n **do**

M_i 中の初出の微小修正を除外

for all $y \in M_i$ **do**

if $\exists x \in M_i, x \Rightarrow y \in R \land x \neq y$ **then**

$C \leftarrow C + 1$

end if

end for

M_i を学習データに加え，相関ルールを更新

end for

た，作成された相関ルールの集合を R，リポジトリ中の微小修正の合計を K，そのうち手法を用いて提示できた微小修正の合計を C とする．

5.1.3 調査項目 3: オープンソース・ソフトウェア中の修正漏れの調査

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

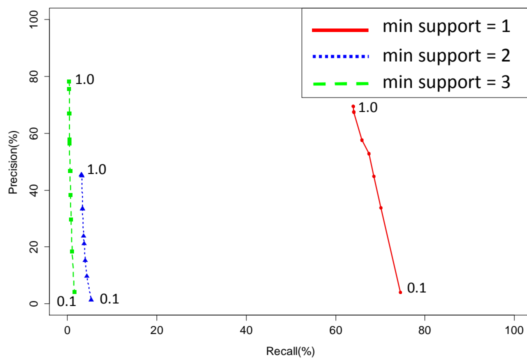
開いた修正漏れ候補: 後に y が行われているが x が行われていないというコミットが存在しない

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．

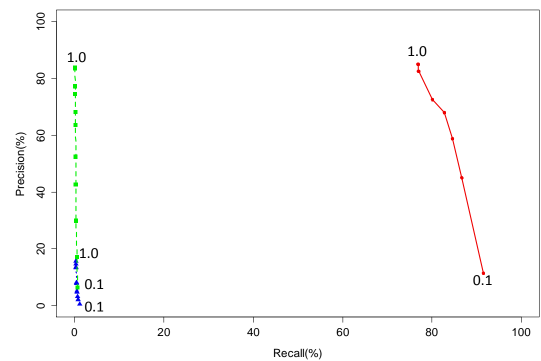
`jEdit` と `ArgoUML` から閉じた修正漏れ候補を無作為に 30 個ずつ選び，それらを手動で調べ，以下の 3 種類に分類した．

- 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

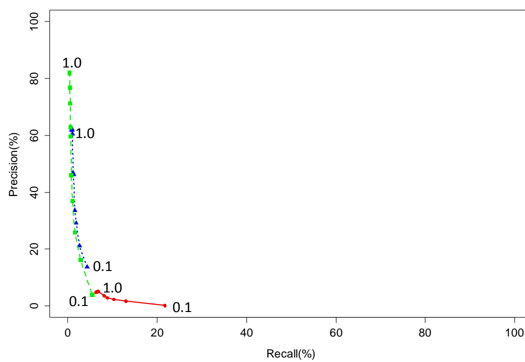
これらの分類は，ソースコードとコミットメッセージから判断する．また，相関ルールの最低支持度は 2，最低確信度は 1 とした．



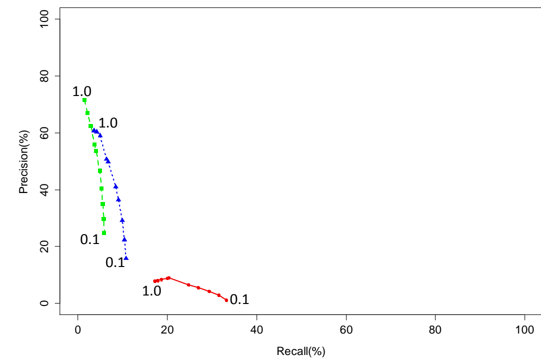
(a) FreeBSD



(b) jEdit



(c) gcc



(d) ArgoUML

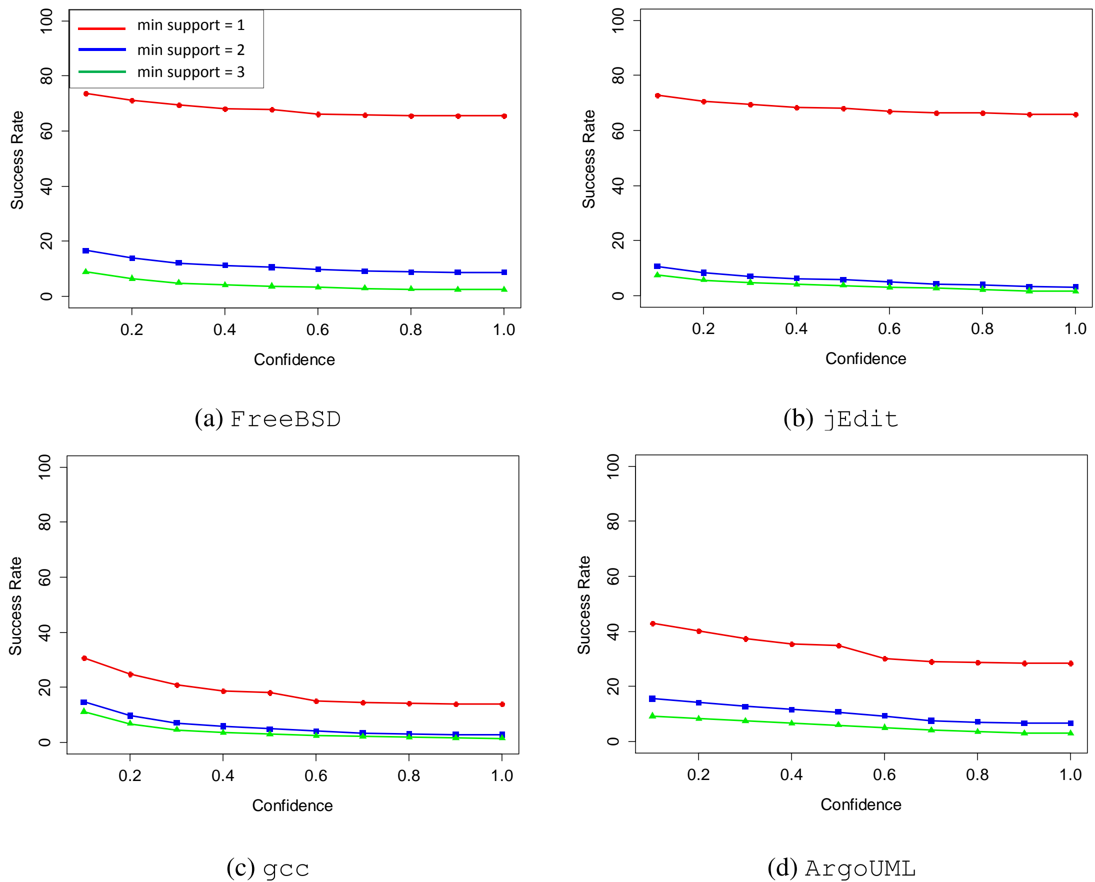
図 5: 相関ルールの適合率と再現率

5.2 実験結果

5.2.1 調査項目 1：相関ルールの適合率・再現率の調査

調査項目 1 の結果を図 5 に示す．グラフは縦軸が適合率で，横軸が再現率を表しており，最低支持度が 1 の場合，2 の場合，3 の場合のそれぞれをプロットしている．また，グラフ上の点は最低確信度を表しており，最低確信度が 0.1 から 1.0 まで 0.1 刻みでプロットされている．

グラフより，最低支持度が 1 の場合は，FreeBSD と jEdit においては適合率と再現率が共に高くなっている．特に jEdit では，最低確信度が 1.0 のとき，適合率が 85%，再現率が 77%と高い値が出ている．一方，gcc と ArgoUML では適合率と再現率が共に低い．最低支持度が 3 の場合は，全てのソフトウェアにおいて再現率が低くなっているものの，全てのソフトウェアで 80 %程度の適合率が出ていることが分かる．



(a) FreeBSD (b) jEdit

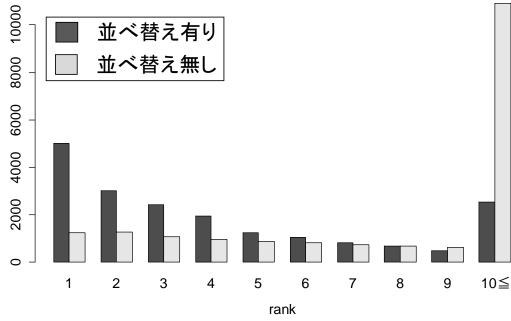
(c) gcc (d) ArgoUML

図 6: 正解を提示できた割合

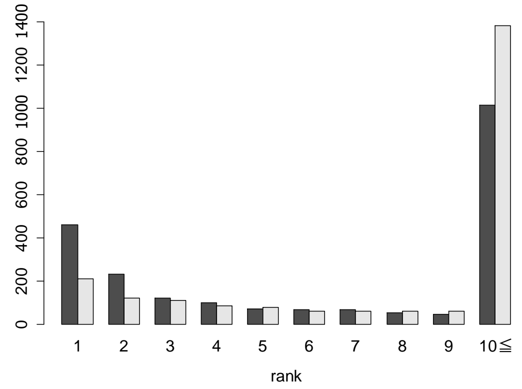
5.2.2 調査項目 2: 修正候補の推薦精度の調査

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 6 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 6 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．

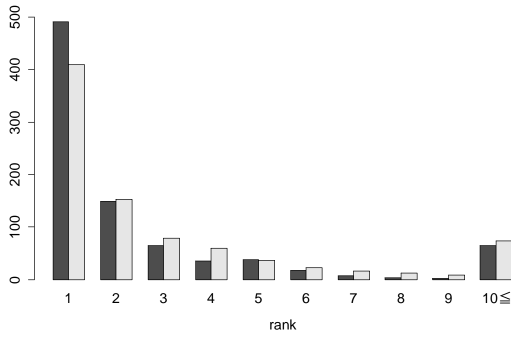
次に，正解の微小修正が提示できた場合において，正解が何番目に提示されるかについての結果を図 7 に示す．ここでは代表として jEdit と ArgoUML での結果を示す．相関ルールの最低確信度は 1 とする．それぞれのソフトウェアで支持度が 1〜3 の場合の計 6 つのグラフを載せている．各グラフは，横軸が提示された順位，縦軸がその順位に提示された微小修正の数であり，候補の並べ替え



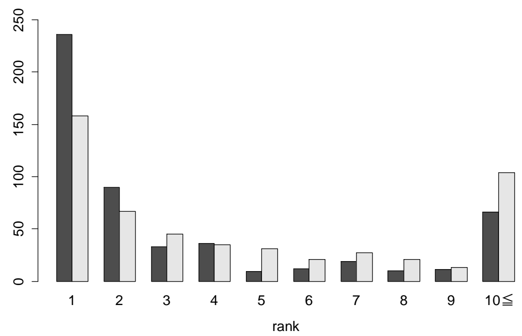
(a) jEdit (最低支持度 1)



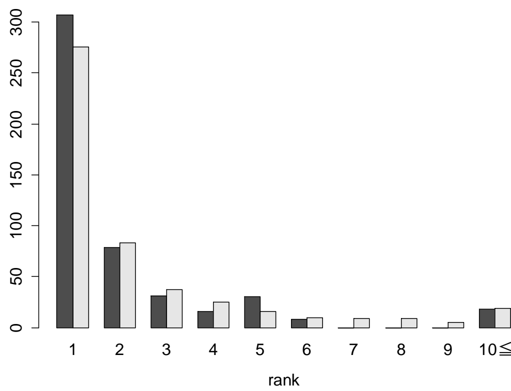
(b) ArgoUML (最低支持度 1)



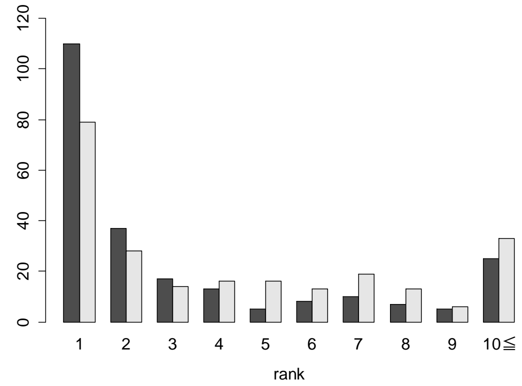
(c) jEdit (最低支持度 2)



(d) ArgoUML (最低支持度 2)



(e) jEdit (最低支持度 3)



(g) ArgoUML (最低支持度 3)

図 7: 正解が何番目に提示されたか

を行ったものとそうでないものの結果を示している．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．

図 7 より，全てのグラフ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．最低支持度が 1 の時で並べ替え無しの場合，どちらのソフトウェアでも 10 位以降に提示された候補が多いことが分かる．並べ替えを行った場合，特に jEdit で顕著な改善が見られる．最低支持度が 2 以上の場合は，上位に正解の候補が提示される割合が高くなっている．

5.2.3 調査項目 3: オープンソース・ソフトウェア中の修正漏れの調査

調査項目 3 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 6 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 7 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示す．リビジョン 7061 に対するコミットで，MirrorList.java において“ LReader parser = XMLReaderFactory . createXMLReader ( )”という文の追加が行われており，それに対して提案手法が以下の様な文の追加を推薦したが，そのコミットでは推薦された修正は行われなかった．

```
if ( OperatingSystem . hasJava14 ( ) && ! OperatingSystem . hasJava15 ( )
              && System . getProperty ( "org.xml.sax.driver" ) == null )
                       System . setProperty ( "org.xml.sax.driver",
                               "org.apache.crimson.parser.XMLReaderImpl" );
```

しかし，その 4 日後のリビジョン 7109 に対するコミットで，ProjectPlugin.java において推薦された修正が行われた．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．“ LReader parser = XMLReaderFactory . createXMLReader ( )”という文の追加は，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文の追加である．XMLReaderFactory は SAX という XML のパー

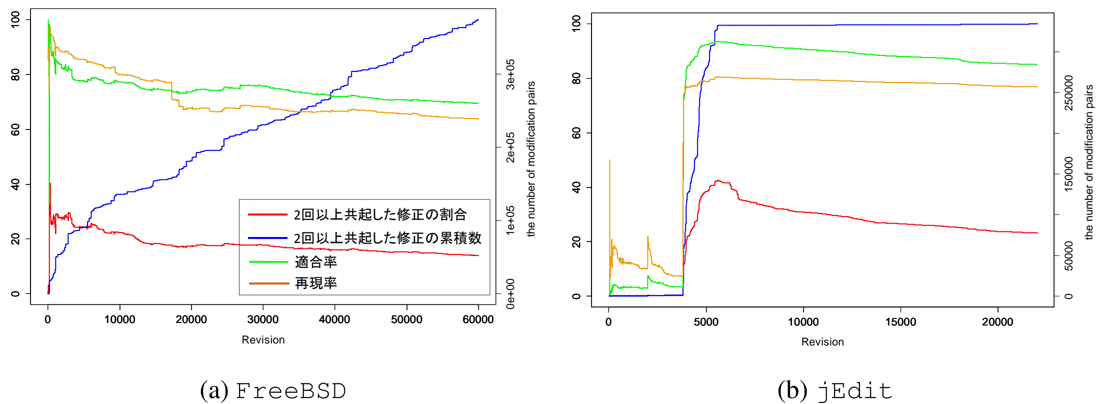
表 6: 修正漏れ候補の数

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

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．

表 7: 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



(a) `FreeBSD` (b) `jEdit`

図 8: 適合率・再現率の推移

6 議論

6.1 パラメータの閾値と適合率・再現率の差について

一般的に支持度が低いルールは，偶然生まれたものである確率が高く，信頼性が低い．しかし，図 5 より，`FreeBSD` と `jEdit` においては最低支持度が 1 の場合でも高い適合率・再現率が出ており，作成されたルールが信頼できるものであることを示している．一方，`gcc` と `ArgoUML` に関しては，最低支持度が 1 の場合では適合率・再現率共に低いため，最低支持度を 1 に設定して手法を用いるのは難しいと思われる．しかし，最低支持度を 2 以上に上げることで適合率を高くすることができるため，一度に推薦される候補の数は少なくなるが誤った候補が推薦される確率は低くなる．よって，`gcc` と `ArgoUML` に関しては，最低支持度を 2 以上にすることで，正確性を重視した推薦を行うことができると考えられる．

次に，図 5 に示される適合率・再現率の差について述べる．最低支持度が 1 の場合の適合率と再現率は，2 回以上共起した微小修正の組の数に左右されると考えられる．そこで，各ソフトウェアについて，全ての共起した微小修正の組に対する 2 回以上共起した微小修正の組の割合を調べた．その結果を表 8 に示す．評価項目 1 で結果が良かった `FreeBSD` と `jEdit` については，2 回以上共起した微小修正の組の割合が高いことが分かる．このことから，2 回以上共起した微小修正の組の割合が高いソフトウェアが提案手法に向いていると考えられる．

表 8: 2 回以上共起した微小修正の組の割合

`FreeBSD`	`jEdit`	`gcc`	`ArgoUML`
16%	20%	0.2%	0.4%

6.2　提案手法が開発のどの時点でも有効であるか

図 5 より，FreeBSD と jEdit に関しては，最低支持度が 1 の場合でも適合率・再現率共に高い値が出ている．この結果は，対象リポジトリ全体を解析した上での結果である．よって，2 回目以降の微小修正の共起がリポジトリの後半に集中している場合，リポジトリの前半では提案手法の効果が低くなってしまう恐れがある．そこで，FreeBSD と jEdit について，リポジトリを解析する過程での適合率・再現率の推移を調べた．その結果が図 8 である．図 8 の各グラフは左の縦軸が 2 回以上共起した微小修正の組の割合・適合率・再現率を百分率で表したものであり，右の縦軸が 2 回以上共起した微小修正の累積数を表したものである．また，横軸はそのリビジョン数である．これより，相関ルールの適合率・再現率は 2 回以上共起した微小修正の組の割合に依存して上下することが分かる．また，ArgoUML に関しては，2 回以上共起した修正の組が一定のペースで増えており，全体を通して適合率・再現率が高いことが分かる．したがって，開発の全体を通して手法が有効であると考えられる．jEdit に関しては，リビジョン 3,800 前後から適合率・再現率が高くなっているため，リビジョン 3,800 以降で手法が有効に働くと考えられる．何故このようなことが起こったか調べた結果，リビジョン 3,800 前後から 5000 リビジョン前後まで，新バージョンへの移行に伴って複数のソースファイルに対して大規模な修正が行われていたことがわかった．それ以降のリビジョンでは，ソースファイルに修正が行われないコミットや，行われても小規模であるコミットが増えている．2 回以上共起した微小修正の組の数の変化はそれに伴うものではないかと思われる．

6.3　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が提示されない確率が高くなるが，正解が提示された場合にそれ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．正解が上位 n 位に提示された確率を表 9，10 に示す．表 9，10 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．また，両ソフトウェアにおいて最低支持度が 2 の場合は並べ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．

表 9: 正解が推薦された順位 (jEdit)

最低支持度	1			2			3		
上位 n 位	1	3	5	1	3	5	1	3	5
並べ替え無し	6.4%	18.6%	28.1%	47.0%	73.6%	64.6%	45.4%	81.0%	89.4%
並べ替え有り	26.2%	54.6%	71.1%	56.3%	80.8%	89.2%	62.9%	85.3%	94.7%

6.4 実際に見つかった修正漏れ

調査項目 3 において，`ArgoUML` で見つかった修正漏れのほとんどがプログラムの動作に影響を与えないものであった．そのうちの多くは，Override アノテーションのつけ忘れや，パッケージ名から指定していたクラスを直接指定するようにする変更 (例: org.argouml.model.ModelFacade → ModelFacade) であった．プログラムの動作に影響を与える修正漏れについては，それぞれ内容が異なっており傾向は見られなかった．

6.5 より精度を上げる方法について

調査項目 3 で見つかった修正漏れ候補のうち半分は実際には修正漏れでなかった．検出された修正漏れ候補が実際には修正漏れでなかった場合の多くでは，出現頻度の高い修正が推薦されていた．例えば，“int $0;”という文の追加は頻度が高い修正であり，頻度が高い修正を含む相関ルールが多数存在する．特に最低支持度が低い場合，このような出現頻度の高い修正を含むルールが偶然生まれてしまう可能性が高い．そこで，提示される微小修正がリポジトリ中に出現する回数を用いて相関ルールのフィルタリングを行うことで出現頻度の高い修正が提示されることを防ぐことが可能になると考えられる．そこで，調査項目 3 についてさらなる調査を行った．表 7 の各ソフトウェアにおいて修正漏れ候補とされた 30 個の微小修正について，そのうちリポジトリ中での出現回数が少ない 15 個だけを抽出した結果を表 11 に示す．表 7 では実際に修正漏れであったものの割合が 50 %であるのに対し，表 11 では，63 %である．これは，微小修正がリポジトリ中に出現する回数を用いてフィルタリングを行うことで，提示される修正候補の数は減るものの，推薦精度は高くなることを示している．

6.6 正規化の是非

変数名・定数名の正規化を行うことで，変数名・定数名が異なる微小修正を同一とみなすことができる．同一とみなせる微小修正が多いほど，手法を適用する機会を増やすことができるため，修正漏れを見つけやすくなると思われる．

メソッド名やクラス名など，より多くの識別子を正規化することで，更に多くの微小修正を同一とみなすことができる．しかし，提案手法で提示される修正候補は正規化後のものであるため，特殊文字に置き換えられた識別子は開発者自身が適切な名前に再度置き換える必要がある．メソッド名や

表 10: 正解が推薦された順位 (`ArgoUML`)

最低支持度	1			2			3		
上位 n 位	1	3	5	1	3	5	1	3	5
並べ替え無し	9.4 %	19.8%	27.0%	30.3%	51.7%	64.4%	33.3%	51.1%	64.6%
並べ替え有り	20.6%	36.5%	44.2%	45.2%	68.8%	77.4%	46.4%	69.2%	76.8%

クラス名まで正規化されると，修正の内容を提示できるという提案手法の利点が損なわれる恐れがある．

6.7 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．

6.8 妥当性への脅威

本研究で行った実験の結果の妥当性について述べる．

実験に用いたソフトウェア

本研究では手法の評価のために 4 種類のソフトウェアを用いたが，他のソフトウェアを用いることで異なる結果が出る恐れがある．そのため．本研究では，Java と C という異なる言語で開発され，リポジトリの規模も異なるソフトウェアを用いることでより一般的な結果が出るように努めた．

表 11: 出現回数の少なかった微小修正

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	4	3	8	15
ArgoUML	1	11	3	15

修正候補推薦の精度

調査項目 2 において，修正候補推薦精度の評価方法が現実に起こる修正漏れに即していない恐れがある．今回は，コミット中の任意の微小修正が 1 つ欠けているという設定で実験を行った．しかし，実際に起こる修正漏れが 1 つだけとは限らない上，修正漏れになりにくいような微小修正が欠けたコミットに対して手法を適用しようとしている場合もあると思われる．今回は，多数のコミットを実験対象として定量的評価を行うという目的のためにこのような設定で実験を行った．

修正漏れ候補の分類

調査項目 3 において，筆者は提案手法により発見された修正漏れ候補を手動で分類した．できるだけ客観的に判断できるようにソースコードの内容やコミットログを参照したが，主観を完全に取り除くことはできていない．これに関しては被験者実験を行うことにより，より客観的な判断を行うことができると思われる．

リポジトリの品質

本研究は，同じコミットで同時に行われやすい修正の組が存在するという仮定に基づいている．そのように仮定できる理由は，一般的に 1 つのコミットではバグ修正や機能追加など，1 つの目的をもった修正を行うことが求められているためである [26]．同じ目的をもったコミットでは，修正の内容も似ていると考えられる．しかし，開発者はしばしば複数のタスクが行われたコミットを行ってしまう [27, 28, 29]．もし，コミットの粒度に関するポリシーが定められておらず，互いに無関係な修正が同時にコミットされることが多ければ，不適切な相関ルールが作成される割合が高くなり，結果的に手法の精度が下がる原因となる．これについては，コミットをタスク単位で分割する研究が行われており，それらの手法を用いることでリポジトリの品質を高められる可能性がある [30, 31]．本研究では，リポジトリマイニングの研究でよく用いられる有名なソフトウェアを用いており，比較的適切にバージョン管理されているため，リポジトリの品質に問題は無いと思われる．

7　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．実験では，定量的評価を行うため，相関ルールの適合率と再現率や正しく修正候補が推薦できる割合を調べた．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

本研究を行うにあたり，親身なご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，親切なご指導をいただき，多くのご助言を頂きました 井垣 宏 特任准教授に深く感謝申し上げます.

本研究の全過程を通し，日々議論の中で的確なご助言を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

また，本研究に関してだけでなく様々な面において親切なご助力，ご協力を頂きました楠本研究室の皆様に心より感謝いたします.

最後に，本研究に至るまでに，講義，演習，実験などで多くの知識や示唆を頂きました，大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[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] J.M. Bieman, A.A. Andrews, and H.J. Yang. Understanding change-proneness in oo software through visualization. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pp. 44–53, May 2003.

[6] D. Cubranic and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 408–418, May 2003.

[7] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pp. 190–198, Nov 1998.

[8] 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.

[9] 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.

[10] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, pp. 200–210, 2012.

[11] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A.E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, Sep 2010.

[12] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pp. 181–190, 2008.

[13] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 274–283, Sep 2009.

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

[15] Mina Askari and Ric Holt. Information theoretic evaluation of change prediction models for large-scale software. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pp. 126–132, 2006.

[16] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pp. 14–14, May 2007.

[17] H. Kagdi, J.I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pp. 145–154, June 2007.

[18] CVS. `http://www.nongnu.org/cvs/`.

[19] Subversion. `https://subversion.apache.org/`.

[20] git. `http://git-scm.com/`.

[21] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 529–536, Sep 2001.

[22] 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.

[23] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, Vol. 22, No. 2, pp. 207–216, June 1993.

[24] Yoshiki Higo and Shinji Kusumoto. How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments? In *28th IEEE International Conference on Software Maintenance*, pp. 222–231, 9 2012.

[25] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 502–511, 2013.

[26] KDE TechBase: Commit Policy. `https://techbase.kde.org/Policies/Commit_Policy`.

[27] Noa Kusunoki, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. How much do code repositories include peripheral modifications? In *5th International Workshop on Empirical Software Engineering in Practice (IWESEP2013)*, pp. 19–24, 12 2013.

[28] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pp. 121–130, 2013.

[29] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pp. 99–108, 2008.

[30] 松田淳平, 林晋平, 佐伯元司. 編集操作履歴の階層的なグループ化を用いたポリシー準拠のコミットの構成支援. ソフトウェアエンジニアリングシンポジウム 2014 論文集, 第 2014 巻, pp. 76–84, aug 2014.

[31] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pp. 262–265, 2014.