修士学位論文

題目

抽象構文木の構造を考慮した編集スクリプトのパターン抽出

指導教員

楠本 真二 教授

報告者

松本 淳之介

令和 2 年 2 月 5 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

令和元年度 修士学位論文

抽象構文木の構造を考慮した編集スクリプトのパターン抽出

松本 淳之介

内容梗概

ソフトウェアの開発において，ソースコードの変更は繰り返される．同じ変更が繰り返されるとき，その変更は変更パターンとして扱える．変更パターンは統合開発環境の補完候補の計算や，自動プログラム修正の修正候補の生成に利用できる．そのため，開発履歴から変更パターンをマイニングする手法が数多く提案されている．変更パターンをマイニングする手法として，Negara らの研究がある．Negara らの研究では IDE の入力を監視し，一定時間間隔で編集スクリプトを求め，変更パターンをマイニングする．しかし，Negara の手法には，「IDE のプラグインが必要」「不正確」「変更パターンの理解の難しさ」という課題がある．このような課題が生じるのは，変更パターンをマイニングする際に抽象構文木の構造を考慮していないことが原因である．

そこで本研究では，抽象構文木の構造を考慮した変更パターンのマイニング手法を提案する．提案手法は Git リポジトリを入力として受け取り，コミット間での編集スクリプトを計算し，抽象構文木に対して頻出部分木マイニングを行なうことで，抽象構文木の構造を保持した変更パターンを出力する．

また，提案手法を TC2P として実装し，TC2P と既存手法である Negara らの研究と比較実験を行なった．その結果， 既存手法よりも正確に変更パターンを TC2P はマイニングできただけでなく，既存手法ではマイニングできいなかった変更パターンもマイニングできた．

主な用語

パターン抽出，リポジトリマイニング，編集スクリプト，変更パターン

目次

1 はじめに 1

2 準備 3

2.1 抽象構文木 3

2.2 GumTree 3

2.3 ラベル付き順序木 4

2.4 頻出部分木マ 0.95 イニング 5

3 研究動機 9

4 提案手法 10

4.1 Step 1. 統合木の作成 10

4.2 Step 2. 頻出部分木マイニング 13

4.3 実装 14

5 評価実験 16

5.1 データセット 16

5.2 Base 16

5.3 実験設定 17

6 実験結果 19

6.1 実験 1 19

6.2 実験 2 24

6.3 実験 3 24

7 妥当性の脅威 26

8 関連研究 27

8.1 パターンマイニング 27

8.2 AST の差分 27

8.3 変更パターン 27

9 あとがき 29

謝辞 30

参考文献 31

図目次

1 編集スクリプトの例と，その編集スクリプトに基づいて AST を操作する例 5
2 最右拡張の例．青色のノードは最右技を，黄色のノードは最右拡張によって追加されたノードを表す． 6
3 順序木の例 7
4 図 3 の順序木に対して FREQT を用いて頻出部分木マイニングを行なう過程．青色の矢印は最右拡張を，出現回数が 2 回以上の部分木のみ $\mathcal{F}_k$ に追加されている様子を緑色の矢印で表している． 8
5 変更例 10
6 提案手法の概要 11
7 統合木が作成される流れ 12
8 不必要な変更パターンの例 14
9 図 8 から不必要な部分木を取り除いた変更パターン 14
10 TC2P と Base の結果のベン図 20
11 デバッグ時にのみログを出力する変更例 22
12 図 11 のような変更からマイニングされた変更パターン (色のついている箇所はその編集操作が適用されていることを示す) 22
13 try-with-reource を追加する変更パターン 23
14 try-with-resource を使った実際の変更 23
15 null チェックをメソッド化した変更パターン 24
16 テストに使用するライブラリを変更した際に伴う変更パターン 25

表目次

1 Base に与えるパラメータ 18
2 実験結果 19
3 各最小サポートでの実行時間 24
4 各ドメインでの実行時間 25

1 はじめに

ソフトウェアの開発において，ソースコードの変更は繰り返される [1, 2, 3, 4]．同様の変更が繰り返されるとき，その変更は変更パターンとして扱われる．変更パターンを知ることで開発者や研究者は多くの知見を得ることができる．例えば，統合開発環境の開発者は，変更パターンを知ることで，補完候補の精度を向上させたり，変更を手動で行なうのではなく自動で行なったりできる [5, 6, 7]．ライブラリの開発者は，ライブラリの誤った使用を修正する変更パターンを知ることで，頻繁に発生する誤った使い方として注意喚起できる．また，自動プログラム修正の研究者は，プログラムのバグを修正する変更パターンを知ることで，そのような変更パターンを自動プログラム修正の修正候補として利用できる [8, 9]．

変更パターンを求める研究は数多くなされている [3, 5, 8, 10, 11, 12]．変更パターンを求める手法の多くは，開発履歴に含まれる変更を編集操作の列である編集スクリプトで表現し，その編集スクリプトに対して頻出する編集操作をマイニングする．

しかしこのような研究の多くは特定の目的に特化した手法になっており，マイニングできない変更パターンが存在するなど，制限のある手法が多い．マイニングする変更パターンに制限のない汎用的な手法として，Negara らの研究 [10] がある．Negara らの研究では IDE の入力を監視し，一定時間間隔で編集スクリプトを求め，変更パターンをマイニングする．しかし，Negara の手法には，「IDE のプラグインが必要」「不正確」「変更パターンの理解の難しさ」という課題がある．この中でも「不正確」は特に重大な課題である．詳しくは 3 章で説明するが，Negara らの手法が出力する変更パターンは同一の変更を表しているとは限らず，変更パターンとして適切でない．このような課題が生じるのは，変更パターンをマイニングする際に抽象構文木の構造を考慮していないことが原因であると著者は考える．

そこで本研究では，抽象構文木の構造を考慮した変更パターンのマイニング手法を提案する．提案手法は Git リポジトリを入力として受け取り，コミット間での編集スクリプトを計算し，抽象構文木に対して頻出部分木マイニングを行なうことで，抽象構文木の構造を保持した変更パターンを出力する．抽象構文木の構造を保持したまま変更パターンを出力することで正確に変更パターンを出力するだけでなく，他の研究や開発で容易にその変更パターンを利用できる．

また，提案手法を TC2P[*1] として実装し，TC2P と既存手法である Negara らの研究とを比較する実験を行なった．その結果， 既存手法よりも正確に変更パターンを TC2P はマイニングできただけでなく，既存手法ではマイニングできていなかった変更パターンもマイニングできた．

本研究では 2 章で準備として抽象構文木の差分を求める方法や頻出部分木マイニングについて述べる．3 章で研究動機を示す．4 章で提案手法について示す．5 章で提案手法の評価実験について述べ，6

[*1] https://github.com/kusumotolab/TC2P

章でその結果を示す．7 章で妥当性の脅威について述べる．また 8 章では関連研究について述べ，最後に 9 章で本研究と今後の課題について述べる．

2 準備

2.1 抽象構文木

抽象構文木 (Abstract Syntax Tree，以降 AST と呼ぶ) とはソースコードを表現した木構造である．1 つのソースファイルに対して 1 つの AST が構築され，AST の各ノードは以下の 5 つで構成されている．

ID 各 AST 内で固有の識別子

親ノードへの参照 各ノードは木構造上の親ノードへの参照を持つ．ただし根ノードには親が存在しないので，何も保持しない．

子ノードへの参照 各ノードは木構造上の子ノードへの参照を持つ．ただし葉ノードには子が存在しないので，何も保持しない．

ラベル if 文や変数宣言など文法上の型を表す．

値 メソッド名や変数名など各ノードが持つラベル以外の情報である．持つ情報がない場合は null になる．

2.2 GumTree

AST の差分を検知する手法として，GumTree[13] がある．GumTree は入力として編集前のソースコードと編集後のソースコードを受けとる．それぞれのソースコードから AST を構築し，それらの木構造の違いを編集スクリプトとして出力する．

GumTree の出力する編集スクリプトは以下の 4 種類の編集操作から構成される．

挿入 $insert(t, t_p, i, l, v)$

ノードの追加を表す．挿入するノードの ID として t，ラベルとして l，値として v，挿入後に親ノードとするノードの ID として t_p，何番目の子にするかを表す数値として i を引数に持つ．

削除 $delete(t)$

ノードの削除を表す．削除するノードの ID として t を引数に持つ．

更新 $update(t, v)$

ノードの値の更新を表す．更新の対象となるノードの ID として t，新しい値として v を引数に持つ．

移動 $move(t, t_p, i)$

部分木の移動を表す．移動する部分木の根ノードの ID として t，移動先の親ノードの ID として

t_p，何番目の子にするかを表す数値として i を引数に持つ．

1 つの編集スクリプトに含まれる編集操作の数を本研究では編集スクリプトの長さと呼ぶ．図 1 に編集スクリプトの例と AST を操作する例を示す．この例の場合，編集スクリプトの長さは 6 である．

GumTree の処理は以下の 2 つで構成されている．

1. ノードのマッチング処理
2. マッチング処理の結果を基にした編集スクリプトの計算処理

GumTree が行なうノードのマッチング処理はトップダウンフェーズとボトムアップフェーズの 2 段階で構成されている．まずトップダウンフェーズでは，入力された 2 つの AST の根から辿っていき，2 つの AST 間で完全一致する部分木を対応づける．ボトムアップフェーズでは，トップダウンフェーズで対応づけられた部分技の根ノードから親を辿り，対応づける候補のノードを探す．ボトムアップフェーズでは候補のノードを根とした部分木の類似度を計算し，それが閾値を超えていれば対応づける．

その後，マッチング処理の結果を基に GumTree は編集スクリプトを計算する．マッチング処理の結果から，各ノードは以下の 3 つに分類される．

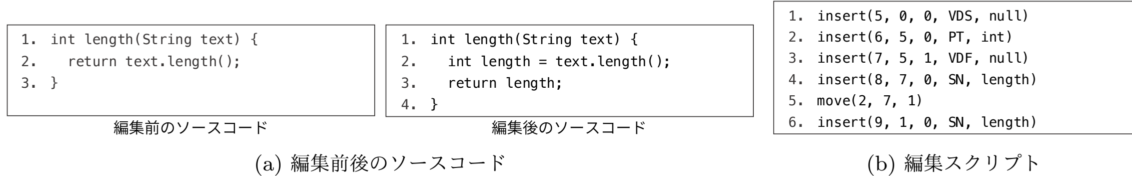
- 編集前の AST にのみ存在するノード
- 編集後の AST に存在するノード
- 編集前後の AST に存在するノード

編集前にのみ存在するノードは，言い換えれば編集後には消えているため，編集によって削除されたことがわかる．同様に編集後にのみ存在するノードは編集によって挿入されたことがわかる．編集前後に存在するノードの親ノードが変わっていれば，そのノードの位置が移動したことがわかり，ノードの値が変わっていれば更新されたことわかり，同じ親ノードかつ同じ値であればそのノードは編集されていないことがわかる．このように，マッチング処理の結果から GumTree は編集スクリプトを計算できる．2 つの木とマッチング処理の結果からその木構造の違いを計算する際の計算は十分に最適化されており [14]，GumTree はこの手法を採用している．

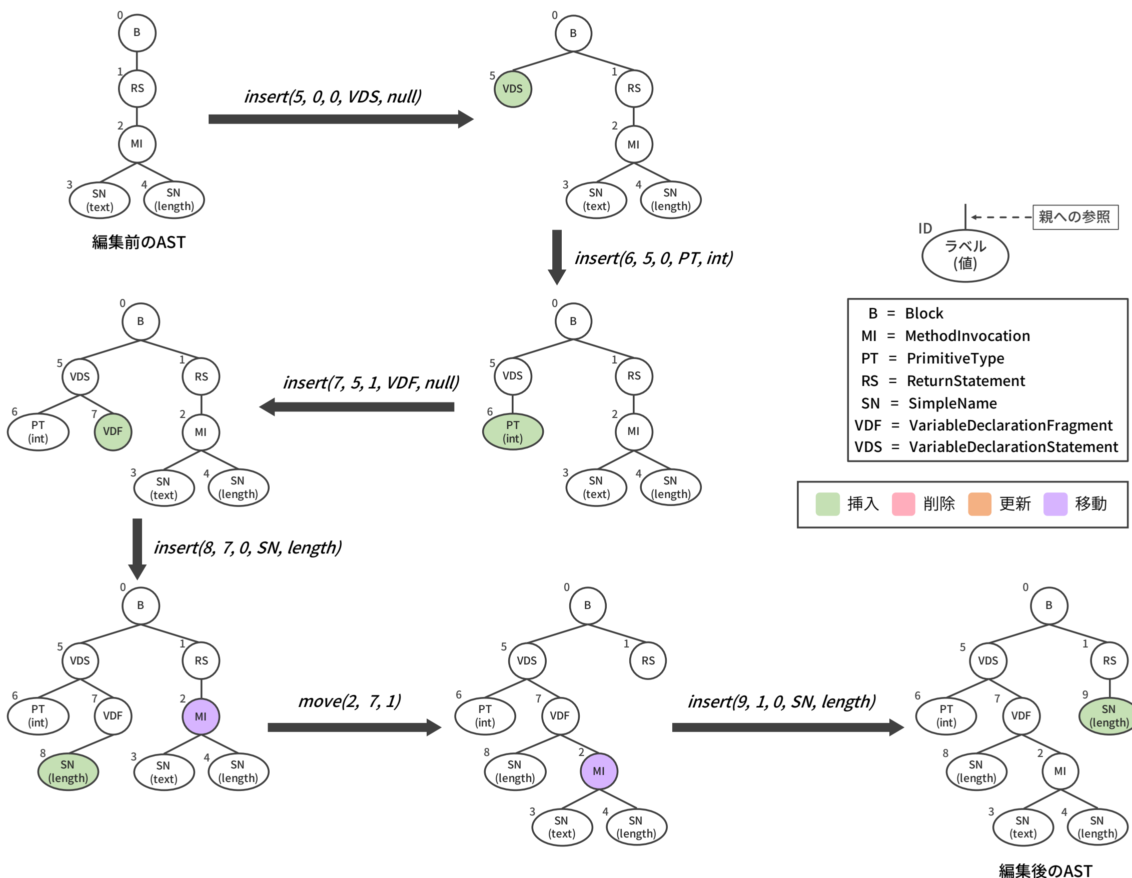
GumTree は様々な研究で応用されている．例えば，Maven のビルドファイルの解析 [15] や自動プログラム修正 [16, 17]，JavaScript のバグのパターンの発見 [18] などに用いられている．

2.3 ラベル付き順序木

順序木とは，木構造に含まれるノードの子ノードに順序関係がある木構造である．また，ラベル付き順序木とは $\mathcal{L} = \{l_0, l_1, l_2, ...\}$ をラベルの有限集合としたとき，各ノードがラベル $l \in \mathcal{L}$ を持つ順序木である．以降ラベル付き順序木のこと単に順序木と呼ぶ．



(a) 編集前後のソースコード　　(b) 編集スクリプト



(c) 編集スクリプトによって AST が変換される例

図 1　編集スクリプトの例と，その編集スクリプトに基づいて AST を操作する例

2.4　頻出部分木マ 0.95 イニング

順序木から頻出する部分木を求めるアルゴリズムとして FREQT[19] がある．FREQT は，頻出するアイテム集合を求める Apriori アルゴリズム [20] を順序木から頻出部分木を求めるために改良したアルゴリズムである．FREQT は入力として，順序木 D と最小サポート σ(minimum support) $(0 \leq \sigma \leq 1)$ を受け取る．順序木 D に含まれる部分木 T の頻出度 $F_D(T)$ が，$\sigma \leq F_D(T)$ のとき，部分木 T を頻出する部分木として出力する．頻出度 $F_D(T)$ とは順序木 D 内に部分木 T が出現する回数を順序木 D に含まれるノードの数で割った値である．

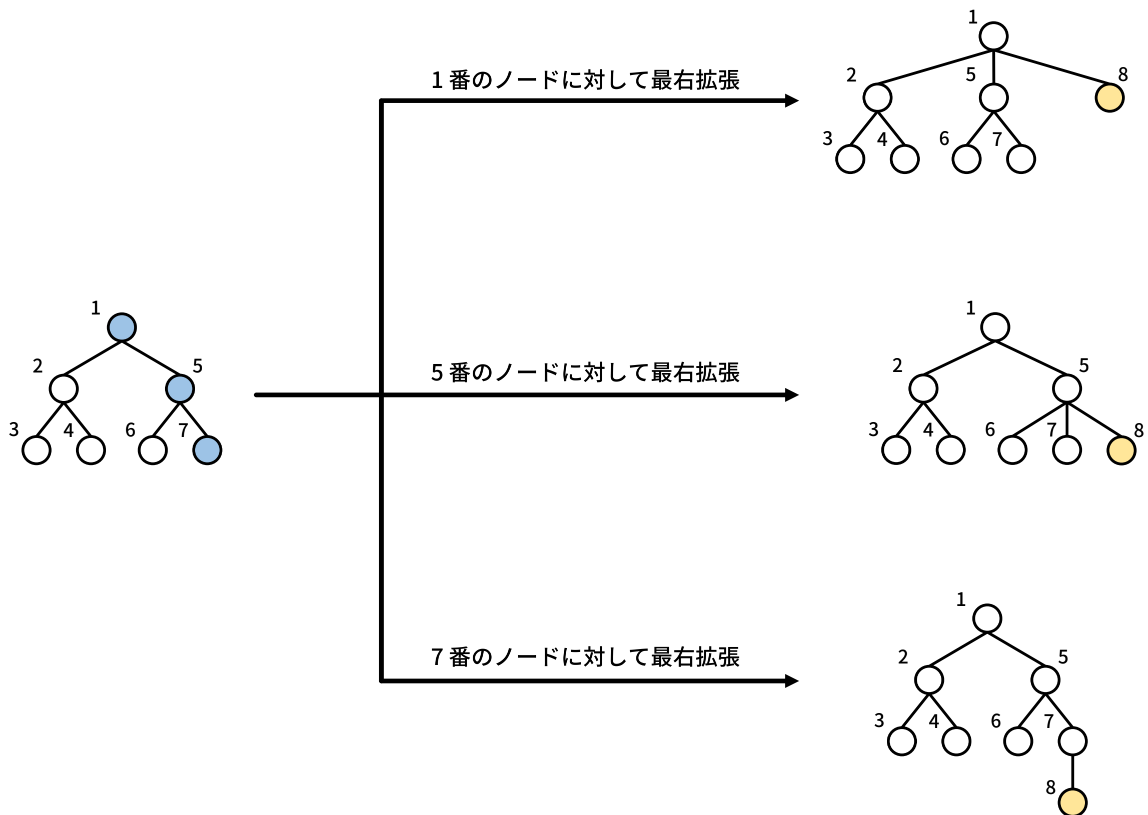


図2　最右拡張の例．青色のノードは最右技を，黄色のノードは最右拡張によって追加されたノードを表す．

FREQT はノードの数が k である頻出部分木の集合 $\mathcal{F}_k$ から，ノードを 1 つ加えたノードの数が $k+1$ である頻出部分木の集合 $\mathcal{F}_{k+1}$ を求める．FREQT は以下の処理で頻出部分木を求める．

Step 1.　入力された順序木 D のノードを走査し，$\mathcal{F}_1$ を求める．

Step 2.　$1 \leq k$ に対して，$\mathcal{F}_k$ からサイズ $k+1$ の候補木 C を列挙し，候補木集合 C_{k+1} に加える．

Step 3.　順序木 D における各候補木 $C \in C_{k+1}$ の頻出度 $\mathcal{F}_D(C)$ を求め，$\sigma \leq \mathcal{F}_D(C)$ であれば C を $\mathcal{F}_{k+1}$ に加える．

Step 4.　$\mathcal{F}_{k+1}$ が空集合であれば終了する．そうでなければ $k = k+1$ として **Step 2.** に戻る．

以上の処理で候補木を全て列挙し，頻出部分木を求める．

Step 2. で $\mathcal{F}_k$ からサイズ $k+1$ の候補木 C を列挙する際，$\mathcal{F}_k$ に含まれる全ての頻出部分木の全てのノードに対して，全てのラベル $l \in \mathcal{L}$ を追加するのは非効率である．そこで FREQT は候補木 C を生成する際，最右拡張を行なう．最右拡張について説明する前にまず最右技について説明する．順序木 T において根ノードから深さ優先で木を走査し，一番最後に到達するノードを最右葉とする．最右技とは根ノードから最右葉への経路のことであり，最右拡張とは最右枝のノードに子ノードを追加すること

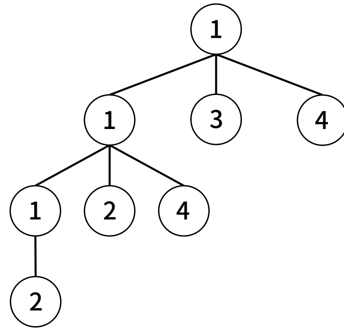


図 3　順序木の例

である．最右拡張を行なうことで，重複も漏れもなく候補木集合 C_{k+1} を求められる．

最右拡張の例を図 2 に示す．この例の場合，1, 5，7 番のノードが最右技に含まれており，それぞれのノードに対して，8 番のノードを追加している．

また，最右拡張をする際に全てのラベル $l \in L$ に対して子ノードを追加するのも非効率である．そこで FREQT は以下の工夫を施している．

ノードスキップ　サイズ 1 の頻出部分木集合 $\mathcal{F}_1$ に含まれていないラベルは，サイズ k の頻出部分木集合 $\mathcal{F}_k$ にも含まれないため，FREQT では $\mathcal{F}_1$ に含まれているラベルのみ最右拡張で追加する

枝スキップ　最右拡張の起点となるノードのラベルを l とすると，サイズ 2 の頻出部分木集合 $\mathcal{F}_2$ のうち根ノードのラベルとして l を持つ部分木の子のラベルのみ最右拡張で追加する

図 3 の順序木に対して，FREQT を用いて頻出部分木を求める過程の頻出部分木集合 $\mathcal{F}_k$ と候補木集合 C_k を図 4 に示す．C_2 を求める際，ノードスキップにより $\mathcal{F}_1$ に含まれるノードのみを用いて最右拡張しており，3 番のノードを追加していないことがわかる．また $3 \leq k$ に対して C_k を求める際，枝スキップにより 1 番のノードに対してのみ，1, 2, 4 番のノードを追加していることがわかる．

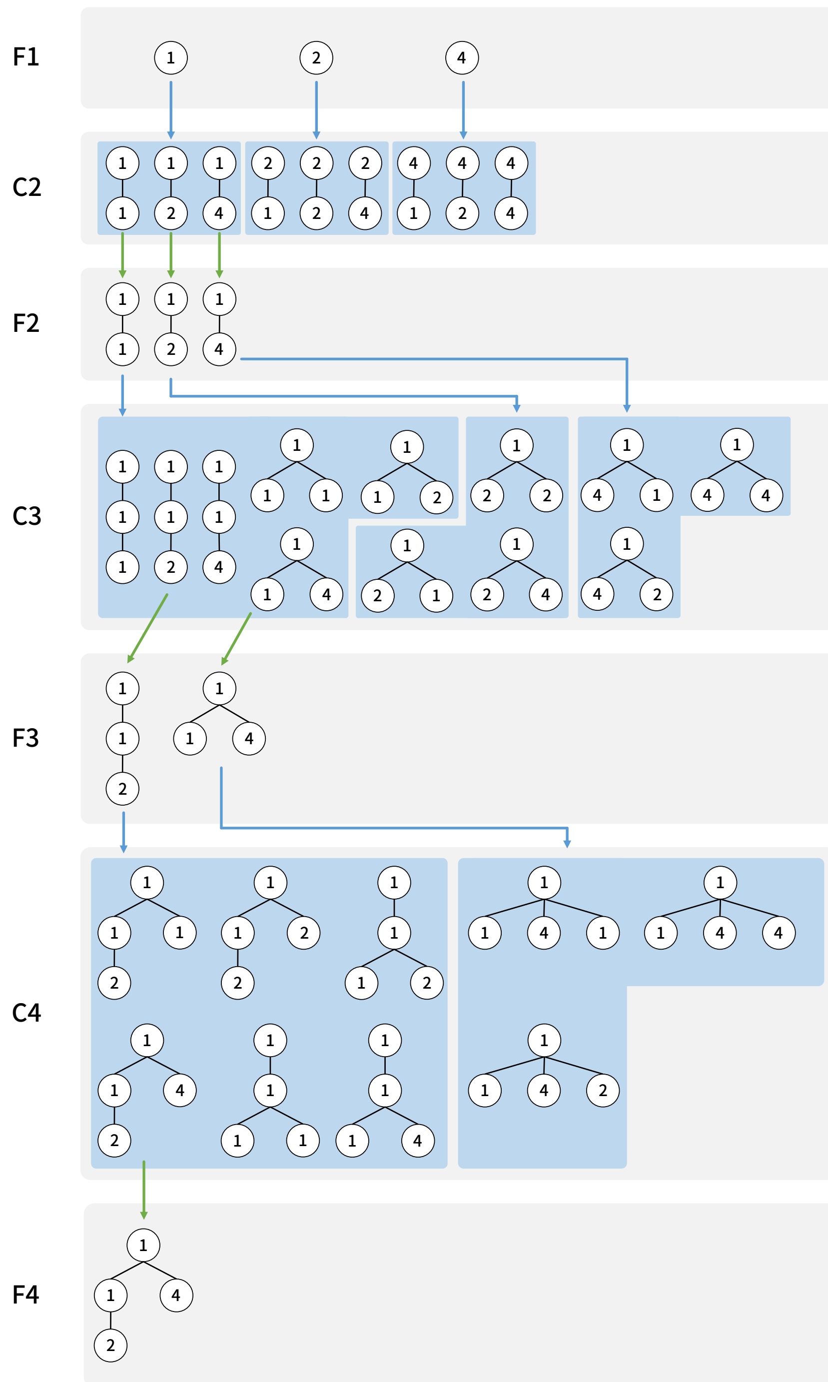


図 4　図 3 の順序木に対して FREQT を用いて頻出部分木マイニングを行なう過程．青色の矢印は最右拡張を，出現回数が 2 回以上の部分木のみ $\mathcal{F}_k$ に追加されている様子を緑色の矢印で表している．

3　研究動機

変更パターンをマイニングする研究として Negara らの研究 [10] がある．Negara らの研究は統合開発環境 (以降，IDE) の入力を監視し，一定の時間間隔で編集スクリプトを計算し，その編集スクリプトに含まれる編集操作を対象にマイニングを行なう．

Negara の研究には 3 つの課題がある．1 つ目の課題は IDE の入力を監視する必要があることである．Negara らの研究は IDE の入力を監視するためのプラグインを開発者が導入し，その記録から変更パターンをマイニングする．そのため，Git などのバージョン管理システムから変更パターンをマイニングできない．つまり変更パターンをマイニングするためには，開発を行なう前に開発者の同意を得て，IDE にプラグインを導入する必要があり，例えば「Java1.8 を導入する際の変更パターン」といった過去の変更パターンはマイニングできない．

2 つ目の課題は編集スクリプトに含まれる編集操作のみを対象にマイニングを行なっていることである．図 5 の 2 つの変更を例に考えてみる．これらの変更はどちらも `if` 文を追加し，`return` 文を追加しているが，その 2 つの文の相対的な位置が異なり，この 2 つの変更を同一の変更パターンとして扱うことは不適切である．しかし，Negara らの研究では編集操作の種類と操作対象のノードのラベルが一致すれば，同一の操作として扱う．よって，これらを同一の変更パターンとして扱い，(1)`if` 文を表す `IfStatement` ノードを追加，(2)`return` 文を表す `ReturnStatement` ノードを追加，という変更パターンを出力する．このように Negara らの研究は同一とするべきでない変更パターンを同一として扱ってマイニングしている．

3 つ目の課題は，出力された変更パターンの情報だけでは，その変更パターンが実際にどのような変更を表しているのか理解できないことである．図 5 に対して出力された変更パターンについて考えてみる．(1)`IfStatement` ノードを追加，(2)`ReturnStatement` ノードを追加，といった変更パターンの情報だけでは，図 5 のどちらの変更を表しているか特定できない．よって出力された変更パターンの情報を別の開発や研究で利用する場合，利用者はその変更パターンが実際はどのような変更であったのか記録されている IDE への入力を再生し，確認する必要がある．

このような課題が発生する原因はマイニングする際に，各編集操作が適用するノード間の相対的な位置関係を考慮していないことである．つまり，図 5 の場合，`ReturnStatement` ノードが `IfStatement` ノードに対してどの位置に追加されたのか考慮する必要がある．

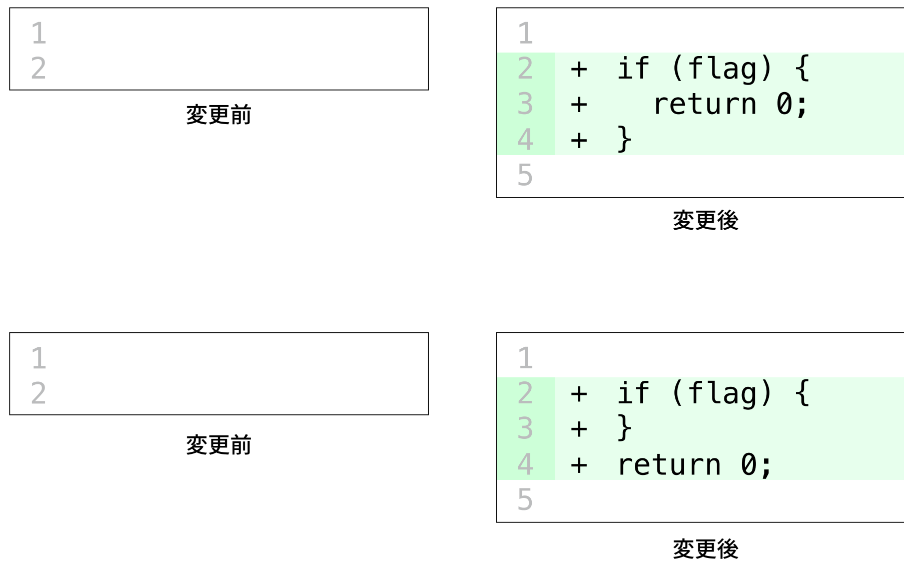


図 5　変更例

4　提案手法

提案手法の概要を図 6 に示す．提案手法は入力として Git リポジトリを受け取り，そのリポジトリに含まれる変更パターンを出力する．提案手法は以下の 2 つの処理で構成されている．

Step 1.　統合木の作成

Step 2.　頻出部分木マイニング

まず入力された Git リポジトリからコミットログを抽出する．各コミットに対して，GumTree[13] を用いて編集スクリプトを求める．次に編集スクリプトから統合木を作成する．統合木とは編集操作と編集前後の AST を 1 つにまとめた木である．統合木については以降の節で詳しく説明する．作成した統合木に対して，頻出部分木マイニングを実行し，変更パターンを求める．なお，編集操作を 1 つも含まない頻出部分木もマイニングされてしまうので，そのような頻出部分木を出力しないよう，出力する前に編集操作を含んでいるか確かめ，含んでいない頻出部分木は出力しない．

4.1　Step 1. 統合木の作成

統合木を作成する目的は編集操作の対象であるノード間の相対的な位置関係を 1 つの木にまとめることである．GumTree[13] が出力する編集スクリプトから，編集前後の AST のどのノードが操作されたかわかる．編集前後の AST に対して以下の処理を行なう．

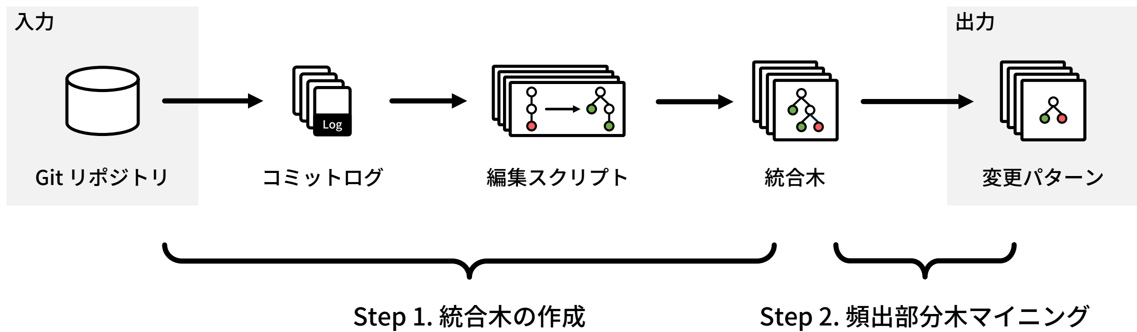


図 6　提案手法の概要

Step A.　編集前後の AST の統合

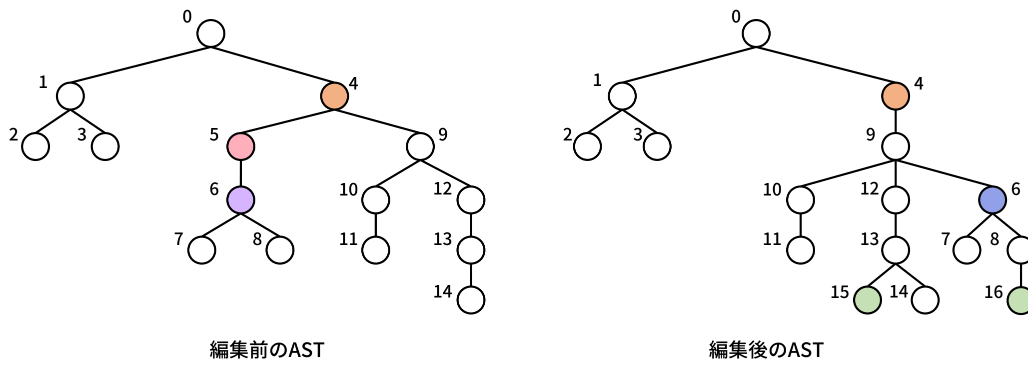
Step B.　不要なノードの削除

まず **Step A.** では編集前後の AST を 1 つにまとめる．編集後の AST には挿入・移動・更新といった編集操作の対象になっているノードが含まれており，そのノードを編集前の AST に加えていく．挿入・移動されたノードの親ノードもまた挿入・移動されている場合があるため，編集後の AST の深さが浅い編集操作から順に編集前の AST に加えていく．挿入されたノードは操作対象のノードのみを編集前の AST に加え，移動されたノードは操作対象のノードを根ノードとする部分木を編集前の AST に加える．移動元のノードには「移動前」，移動後のノードには「移動後」と移動の前後で異なる編集操作を割り当てる．これは移動の前後では同一の部分木が存在し，それらを変更パターンとして扱わないため工夫のである．なお，更新されたノードは編集前後で同じ位置に存在するため，編集前の AST に加えない．

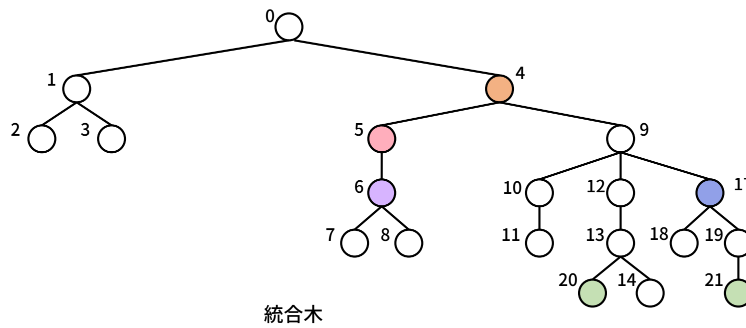
以上の処理で生成した統合木では，編集していないノードを多数含んでしまうので，無駄な変更パターンをマイニングしてしまい，効率的ではない．そこで **Step B.** では 1 つにまとめた AST から変更パターンのマイニングに不要なノードを取り除く．以下のどの条件にも当てはまらないノードを取り除いていく．

- 編集操作の対象
- 移動の部分木に含まれている
- 子ノードを複数もつ
- 親ノードが存在する

統合木が作成される流れを図 7 に示す．**Setp A.** で挿入された 15, 16 番のノード，移動された部分木 (根ノードが 6 番) が編集前の AST に追加されている．なお，1 つの木の中でノードの ID が重複するのを避けるため，追加されたノードの ID は重複しないよう変更している．**Step B.** では変更パター



Step A.　編集前後の AST の統合



Step B.　不要なノードを削除

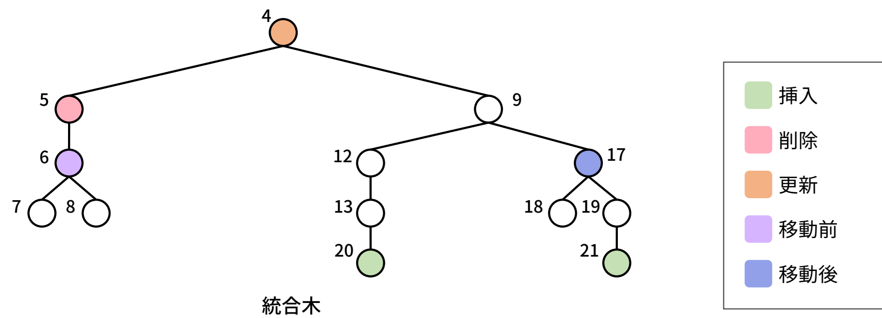


図 7　統合木が作成される流れ

ンに不要なノードとして 0 ～ 3, 10, 11, 14 番のノードが取り除かれていることがわかる．

4.2 Step 2. 頻出部分木マイニング

作成した統合木に対して頻出部分木をマイニングする．提案手法は FREQT[19] を基にマイニングする．

4.2.1 FREQT の拡張

FREQT は 1 つの木に含まれる頻出部分木を求めるが，提案手法では複数の木に含まれる頻出部分木を求める．そのため提案手法では， 2.4 節で述べた FREQT の **Step 1.** で $\mathcal{F}_1$ を求めるために走査するデータ木を 1 つから複数に拡張する．

また FREQT では，最小サポート σ を $0 \leq \sigma \leq 1$ とし，σ 以上の割合で出現する部分木を頻出部分木としている．しかし，変更パターンをマイニングする上では，変更パターンの出現する割合ではなくて，出現回数を基準とした方が自然であり，実際に既存研究は出現回数を基準としている [10, 11]．そこで提案手法でも変更パターンの最小サポート σ を $1 \leq \sigma$ とし，出現回数が σ 回以上である部分木を頻出部分木とした．

4.2.2 ノードの等価性

頻出部分木をマイニングするためには，統合木に含まれるノードの等価性を定義する必要がある．提案手法では以下の全てが一致するノードを等価なノードとして扱った．

- 編集操作の種類
- 編集対象のノードのラベル
- 編集対象のノードの値
- 編集操作の種類が更新の場合，編集対象のノードの新しい値

4.2.3 計算量を抑える工夫

提案手法では計算量を抑えるために，不必要な変更パターンを生むことが確定している頻出部分木を候補木集合 C に加えない．図 8 を例に説明する．提案手法は頻出部分木を求めるために最右拡張を行なう．最右拡張は最右技に対してのみノードを追加していくため，新たに追加したノードの左側，つまり追加したノードの兄にあたるノードの部分木 T_{left} に対して，今後新たにノードが追加されることはない．T_{left} が変更パターンに必要でない部分木であれば，図 8 の変更パターンを候補木集合 C_{k+1} に加えない．

以下の条件を同時に満たす部分木 T_{left} を変更パターンに不必要な部分木とした．

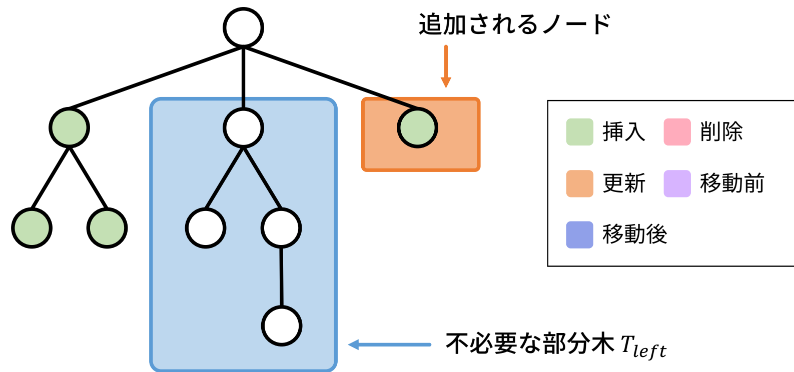


図 8　不必要な変更パターンの例

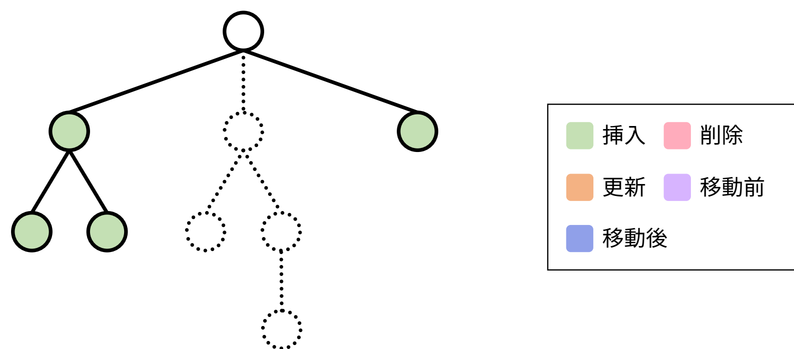


図 9　図 8 から不必要な部分木を取り除いた変更パターン

- 移動された部分木の中に含まれていない
- 部分木 T_{left} 内に編集操作を 1 つも含まない

以上の条件を満たす部分木 T_{left} を含む部分木 T を候補木集合 C_{k+1} に含まなくても，FREQT の性質上，図 9 のような部分木から不必要な部分木 T_{left} を取り除いた部分木 T' は頻出部分木として出力される．よってこのような頻出部分木を候補木集合 C_{k+1} に加えなくても問題ない．

4.3　実装

提案手法の実装として TC2P[*2] を実装した． TC2P は Java を用いて実装されている．

Git リポジトリに含まれる全てのコミットに対して編集スクリプトを計算する処理は長い時間を必要とする．何度も同様の変更に対して編集スクリプトを計算することを避けるため，TC2P では作成した統合木を SQLite に保存する．頻出部分木マイニングを行なう際は，SQLite から統合木を取り出して頻出部分木マイニングを行なう．

[*2] https://github.com/kusumotolab/TC2P

また，TC2P は FREQT 内のサイズ k の頻出部分木集合 $\mathcal{F}_k$ を求める処理を並列で実行する．

5 評価実験

提案手法を評価するために，Java で記述されているオープンソースソフトウェア (以降 OSS とする) に対して提案手法を実行した．この評価実験では以下のリサーチクエスチョン (以降 RQ とする) に答えていく．

RQ1 提案手法は既存手法より正確な変更パターンを出力しているか

RQ2 提案手法の実行速度はどの程度であるか

RQ3 プロジェクト間での変更パターンは存在するか

5.1 データセット

評価実験をするにあたり，Borges ら [21] による調査データ*3のリポジトリのうち Java で記述されている 202 個のリポジトリを用いた．Borges らによって，各リポジトリはそのリポジトリが扱うドメインとして，以下の 6 つのドメインのいずれかかが割り当てられている．

- Application-Software
- Documantation
- Non-Web-Library-and-Frameworks
- Software-Tool
- System-Software
- Web-Library-and-Frameworks

実験ではこれらのリポジトリの最新 1,000 コミットを用いた．

5.2 Base

TC2P を評価するために，既存手法と比較する必要がある．しかし，既存研究である Negara らの研究 [10] のツール及びデータセットは公開されていない．そこで論文に述べられている情報 [10, 22] を基に Base を実装した．Base は編集スクリプトの集合をマイニングし，編集操作の集合を変更パターンとして出力する．

Base における編集操作の等価性は TC2P とは異なり，以下の全てが一致する編集操作を等価とした．

- 編集操作の種類

*3 https://goo.gl/73Sbvz

- 編集対象のノードのラベル

なお，既存研究 [10] は IDE の入力を監視し，一定時間間隔で編集スクリプトを計算し，変更パターンをマイニングする．これは 1 つの編集スクリプトの中には 1 つの関心事の編集操作しか含めないための工夫である．しかし提案手法と既存手法を評価するには，提案手法と入力を統一し，各コミットの各ファイルの編集スクリプトを計算し，変更パターンをマイニングする必要がある．1 つのファイルの編集スクリプトをそのままマイニングの対象にすると，「マイニング対象の編集スクリプトには 1 つの関心事の編集操作しか含めていない」が前提の既存手法が不利となる．そこで可能な限り公平に既存手法と提案手法を評価するため，マイニング対象のリポジトリに対して FinerGit[23] を適用した．FinerGit とは Historage[24] のアイデアを基にしたツールである．FinerGit は Java ファイルを含んでいるリポジトリを入力として受け取り，メソッドがファイル化されたリポジトリを出力する．マイニング対象のリポジトリに対して FinerGit を適用し，出力されたリポジトリを TC2P や Base に入力として与えることで，各ファイルごとの編集スクリプトではなく，各メソッドごとの編集スクリプトに対して TC2P や Base は変更パターンをマイニングする．このように 1 つ編集スクリプトの中に 1 つのメソッドの変更しか含めないことで，可能な限り既存手法が想定している状況に近づけて実験を行なう．

5.3 実験設定

RQ に答えるために，以下の 3 つの実験を行なった．

実験 1 データセットの各リポジトリに対して，TC2P がマイニングした変更パターンと Base がマイニングした変更パターンを比較する実験

実験 2 データセットの各リポジトリに対して，TC2P の最小サポートを変更して，その実行速度を評価する実験

実験 3 データセットの各ドメインに対して，プロジェクトをまたいだ変更パターンを TC2P でマイニングする実験

全ての実験でタイムアウトを 2 時間に設定した．2 時間を超えた場合，強制的に実行を終了する．

また，全ての実験で 64GB のメモリと 12 コアの CPU をもつマシンで実験を行なった．TC2P，及び Base は並列に処理を実行するように実装されている．

5.3.1 実験 1

実験 1 ではデータセットに含まれる各プロジェクトに対して変更パターンをマイニングする．

TC2P を実行するために，最小サポートを設定する必要がある．実験 1 では最小サポートを 10 とした．

Base には Absolute Frequency Threshold(以降 AFT) と Dynamic Threshold(以降 DT) を設定する必要がある．出現回数が AFT を超えない変更パターン候補は変更パターンとして扱わない．また，変更パターン候補のサイズに出現回数をかけた値が DT を超えていない変更パターン候補は変更パターンとして扱わない．Base は Apriori アルゴリズム [20] のようにアイテムを 1 つずつ追加して変更パターンの候補を列挙するが，追加するアイテムの出現回数に応じて AFT と DT を動的に閾値を切り替える．実験 1 では，AFT と DT に表 1 のようにした．

5.3.2　実験 2

実験 2 ではデータセットに含まれる各リポジトリに対して，TC2P に与える最小サポートの値を変えて，変更パターンをマイニングし，その実行速度を評価する．TC2P に与える最小サポートを 10, 25, 50, 100 の 4 つの値で実験した．

5.3.3　実験 3

実験 3 ではデータセットに含まれるドメインごとに変更パターンをマイニングする．ただしそのまま TC2P を実験に用いるのでは，実験 1 と同様の変更パターンがマイニングされてしまう．そこで実験 3 ではプロジェクトをまたいでいる変更パターンのみを対象にマイニングする．

マイニング対象のデータセットが実験 1 と比べて大きくなるので，実験 3 では TC2P に最小サポートとして 25 を与えた．

表 1　Base に与えるパラメータ

出現回数 $\mathcal{F}$	AFT	DT
$250 \leq \mathcal{F}$	10	250
$50 \leq \mathcal{F} < 250$	10	50
$10 \leq \mathcal{F} < 50$	10	10

6 実験結果

6.1 実験 1

実験 1 の結果の概要を表 2 に示す．TC2P と比べて Base の変更パターンの数が極端に少ないのは，TC2P の出力が統合木の部分木であり，Base の出力が編集操作の集合だからである．つまり，TC2P は変更パターンに含まれる各編集操作の数が異なれば別の変更パターンとするが，Base は変更パターンに含まれる各編集操作の数が異なっていても編集操作の種類が同一であれば同じ変更パターンとするからである．

Base と比べて TC2P が正確に変更パターンを検出できるのか確認するために，各手法でマイニングされた変更パターンを比較する．しかし，TC2P は木構造で変更パターンを表現し，Base は編集操作の集合で変更パターンを表現するため，純粋な比較ができない．そこで，TC2P で出力された木構造の変更パターンに含まれる編集操作を取り出し，編集操作の集合に変換して比較する．Base がマイニングした変更パターンの編集操作の集合と TC2P がマイニングした変更パターンを変換した編集操作の集合が完全一致した場合，それらの変更パターンは同一とみなす．

図 10 は 2 つの手法の結果をベン図で表現している．TC2P でのみマイニングされた変更パターンの数は 62,809 個，両方の手法でマイニングされた変更パターンの数は 10,974 個，Base のみでマイニングされた変更パターンの数は 488,137 個という結果になった．表 2 の TC2P でマイニングされた変更パターンの数と図 10 の TC2P でマイニングされた変更パターンの数が異なっているのは，木構造の変更パターンを編集操作の集合の変更パターンに変換しているからである．

まず，TC2P でのみマイニングされた変更パターンについて注目する．これらの変更パターンの中に含まれる移動という編集操作について注目すると，62% の変更パターンに移動が含まれていた．既存手法では編集スクリプトの編集操作に移動がなく，Base も既存手法に則って実装されているため，その差だと考えられる．変更パターンを表現する上で移動という編集操作は意味のある情報であるため，その点で TC2P は Base より優れているといえる．

また，残りの 38% については，Base が AFT と DT を採用していることが原因であると考えられる．TC2P は出現回数が 10 回以上の変更パターンを全てマイニングしているが，Base は出現回数が

表 2 実験結果

手法	変更パターンの数	実行時間の平均 (秒)	タイムアウトの数
TC2P	93,019,161	30.3	17
Base	499,111	55.8	3

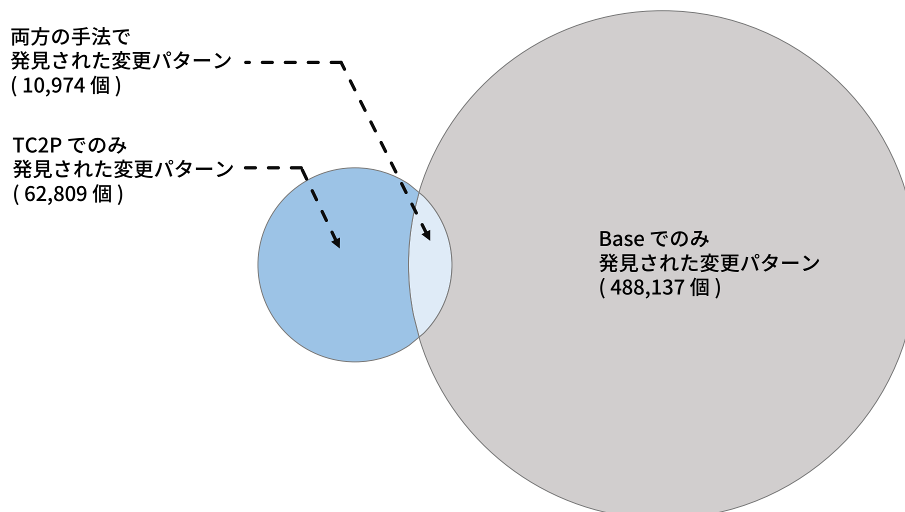


図 10　TC2P と Base の結果のベン図

10 回以上の変更パターンを全てマイニングしているわけではない．動的な閾値を用いずに，出現回数が 10 回以上の変更パターンを全てマイニングできているという点で TC2P は Base より優れているといえる．

次に Base でのみ発見された変更パターンについて考える．Base のみで発見された変更パターンは Base で発見した変更パターンの 97% を占めており，これらの変更パターンについて以下の 2 種類が考えられる．

1. TC2P では移動を用いて表現されていた変更パターン
2. 木構造を考慮すると変更パターンではない変更パターン

TC2P では編集操作として移動を用いて変更パターンを表現しているが，それを用いずに表現している変更パターンが 1. に当てはまると考えられる． また，構造を無視することで異なる変更を誤って同一の変更と扱い，出現回数を多く数えてしまい， その結果変更パターンではないにも関わらず変更パターンとして出力された変更パターンが 2. に当てはまると考えられる． これらのことから，Base のみで発見された変更パターンは，TC2P ではより良い形で出力された，もしくはそもそも変更パターンとして出力されなかったと考えられる．

そのことを確かめるため，Base でのみ発見された変更パターンに対して目視で確認を行なった．Base でのみ発見された変更パターンに対して，編集操作の数が多い上位 1,000 個の変更パターンからランダムに 30 個の変更パターンを選び，その変更パターン前後のソースコードを目視で確認した．その結果，29 個の変更パターンは異なる変更を同一の変更として扱い，出現回数が増えたため，不適切に変更パターンとして出力していた．また，残りの 1 個の変更パターンに含まれる変更は全て同一の変更

であったが，その変更パターンは移動を用いることでより適切に表現できる変更パターンであった．その変更パターンに対して，移動を用いることでより適切に表現した変更パターンは，TC2P のみでマイニングされた変更パターンの中に含まれていた．

6.1.1 変更パターンの例

TC2P によってマイニングされた変更パターンの例を示す．

デバッグ時にのみログを出力する変更を図 11 に，そのような変更からマイニングされた変更パターンを図 12 に示す．実際は木構造として変更パターンが出力されるが，説明のためここではソースコードで表現している．変数名を表す `SimpleName` が変更パターンとして含まれていない箇所は，「$ + 数字」で表現している．編集後には if 文が追加され，編集前の 1 行目が if 文の中に移動していることがわかる．この移動という編集操作はすでに述べたように Base では扱わない編集操作であるが，もしたとえ Base に移動が存在したとしても，Base が出力する編集操作の集合は各ノードの相対的な位置関係を無視してしまうため，ノードがどこからどこに移動したのかわからない．一方 TC2P であれば AST の構造の情報を用いて変更パターンをマイニングしているため，図 12 のように，ノードがどこからどこに移動したかがわかる．これより，木構造での変更パターンのマイニングと移動という編集操作の相性が良いことがわかる．

次に try-with-resource 文を追加する変更パターンの例の図 13 に示す．`$VARIABLE_DECLARARION_STATEMENT` は変数宣言を，`$EXPRESSION_STATEMENT` は式を表す文を表している．変数の宣言を try-with-resource 文に移動させ，具体的な処理をそのブロックの中で行なっている．また，try-with-resource 文を導入することで不要となった `close` メソッドの呼び出しが削除されている．

具体的な式や文ではなく抽象化された AST ノードに対して編集操作の移動が出力されていることに注目する．提案手法は AST に対して頻出部分木マイニングを適用するため，具体的な処理 (どのような変数が宣言されているか/どのような処理がなされているのか) に関わらず，それらを抽象的に変更パターンとしてマニイングできる．この変更パターンがマイニングされた実際のソースコードの変更例を図 14 に示す．図 13 のように全く異なる変数を宣言していたり，全く異なる処理をしていることがわかる．このように TC2P は木構造に対してマイニングを行なうことで，図 12 のような具体的な処理だけでなく，図 13 のような具体的な処理に対しても変更パターンをマイニングできることがわかる．

変数の null チェックを行なう処理をメソッド化した変更パターンの例を図 15 に示す．null かどうかを確認する処理を複数の箇所で行なっていたが，その処理をメソッドにし，複数箇所で変更が行なわれ，変更パターンとしてマイニングされた．このように，対象のプロジェクト特有の変更パターンも TC2P ではマイニングできた．

以上より，RQ1 の回答として，「**提案手法は既存手法ではマイニングできていない変更パターンをマ**

```
1     @Override
2     protected void onCreate(final Bundle pSavedInstanceState) {
3  -    Debug.d(this.getClass().getSimpleName() + ".onCreate" +
            " @(Thread: '" + Thread.currentThread().getName() + "')");
4       super.onCreate(pSavedInstanceState);
5       this.mGamePaused = true;
6       this.mEngine = this.onCreateEngine(this.onCreateEngineOptions());
7       this.applyEngineOptions();
8       this.onSetContentView();
9     }
```

編集前

```
1     @Override
2     protected void onCreate(final Bundle pSavedInstanceState) {
3  +    if(BuildConfig.DEBUG) {
4  +      Debug.d(this.getClass().getSimpleName() + ".onCreate" +
              " @(Thread: '" + Thread.currentThread().getName() + "')");
5  +    }
6       super.onCreate(pSavedInstanceState);
7       this.mGamePaused = true;
8       this.mEngine = this.onCreateEngine(this.onCreateEngineOptions());
9       this.applyEngineOptions();
10      this.onSetContentView();
11    }
```

編集後

図 11　デバッグ時にのみログを出力する変更例

```
1. Debug.d($0.getSimpleName() + " @(Thread: '" + $2.getName() + "')");
```

編集前

```
1. if (BuildConfig.DEBUG) {
2.   Debug.d($0.getSimpleName() + " @(Thread: '" + $2.getName() + "')");
3. }
```

編集後

挿入　削除　移動　更新

図 12　図 11 のような変更からマイニングされた変更パターン (色のついている箇所はその編集操作が適用されていることを示す)

イニングすることができ，さらに提案手法は AST の構造を考慮することで，適切でない変更パターンを誤ってマイニングすることもない」といえる．

```
1. $VARIABLE_DECALRATION_FAGMENT;
2. $EXPRESSION_STATEMENT;
3. $0.close();
```

編集前

```
1. try ($VARIABLE_DECLARATION_FRAGMENT) {
2.   $EXPRESSION_STATEMENT;
3. }
```

編集後

■ 挿入 ■ 削除 ■ 移動 ■ 更新

図 13　try-with-reource を追加する変更パターン

```
1. ResultSet rs = statement.executeQuery();
2. Assert.assertFalse(rs.next());
3. rs.close();
```

編集前

```
1. try (ResultSet rs = statement.executeQuery()) {
2.   Assert.assertFalse(rs.next());
3. }
```

編集後

■ 挿入 ■ 削除 ■ 移動 ■ 更新

(a) 変更例 1

```
1. Connection c = ds.getConnection();
2. Assert.assertNotNull(c);
3. c.close();
```

編集前

```
1. try (Connection c = ds.getConnection()) {
2.   Assert.assertNotNull(c);
3. }
```

編集後

■ 挿入 ■ 削除 ■ 移動 ■ 更新

(b) 変更例 2

図 14　try-with-resource を使った実際の変更

```
1. if ($0 == null) {
2.   throw new NullPointerException($1);
3. }
```

編集前

```
1. ObjectUtil.checkNotNull($2, $3);
```

編集後

挿入 削除 移動 更新

図 15　null チェックをメソッド化した変更パターン

6.2　実験 2

実験 2 の結果を表 3 に示す．なお，実行時間の平均はタイムアウトしなかったプロジェクトに対して計算した．FREQT のアルゴリズムは出現する変更パターンの数が多いほど，実行時間が長くなる．そのため，最小サポートの数が大きいほど，変更パターンと扱われる部分木の数が減るため，実行時間は短くなっている．ただし，最小サポートが 10 の時と 25 の時を比較すると，25 の時の方が長くなっている．これはタイムアウトのプロジェクトが減ったためであると考えられる．

以上より，RQ2 の回答として「**タイムアウトのプロジェクトがいくつか存在するものの，多くのプロジェクトに対して短い実行時間でマイニングできており，TC2P の実行時間は十分実用的な時間である**」といえる．

6.3　実験 3

実験 3 の結果の概要を表 4 に示す．Non-Web-Library-and-Frameworks のみタイムアウトとなったが，残りの 5 つのドメインについては制限時間内にマイニングできた．同じドメインであればプロジェクトをまたがっていても変更パターンが十分存在することがわかる．

表 3　各最小サポートでの実行時間

最小サポート	実行時間の平均 (秒)	タイムアウトの数
10	30.3	17
25	38.8	6
50	11.9	2
100	4.8	0

```
1. assertFalse($0);
2. assertFalse($1);
3. assertFalse($2);
```
編集前

```
1. assertThat($0).isFalse();
2. assertThat($1).isFalse();
3. assertThat($2).isFalse();
```
編集後

挿入 削除 移動 更新

図 16　テストに使用するライブラリを変更した際に伴う変更パターン

実験 3 でマイニングされた変更パターンの例を図 16 に示す．これはテストに使用するライブラリを Hamcrest[*4]から AssertJ[*5]に移行する変更パターンである．この変更パターンは Web-Library-and-Frameworks からマイニングされた．しかし，このような変更パターンは Web-Library-and-Frameworks というドメイン特有の変更パターンではない．マイニングされたパターンから，ドメインに特化した変更パターンを見つからなかった．

以上より，RQ3 の回答として「**TC2P ではリポジトリをまたいだ変更パターンをマイニングできるが，ドメイン特有の変更パターンはマイニングされなかった**」といえる．

表 4　各ドメインでの実行時間

ドメイン	マイニングされた変更パターンの数	実行時間 (秒)
Application-Software	10,441	33
Documentation	1,950	13
Non-Web-Library-and-Frameworks	-	タイムアウト
Software-Tool	64,882	525
System-Software	145,798	1,240
Web-Library-and-Frameworks	600,416	4,858

[*4] http://hamcrest.org/JavaHamcrest/
[*5] https://github.com/joel-costigliola/assertj-core

7 妥当性の脅威

実験では全て Java のプロジェクトを対象に行なったが，GumTree は Java 以外にも Python や JavaScript などの言語にも対応しており，Java 以外の言語で同一の結果が得られるとは限らない．

既存手法として論文 [10, 22] の情報を基に Base を実装し，TC2P と Base を比較した．しかし，論文に記述されていない工夫などがされている可能性があり，その場合実験の結果が変わることが予想される．

また，実験では各コミット間のメソッド単位での編集スクリプトに対してマイニングを行なうことで，1 つの編集スクリプトに 1 つの関心事に対する編集操作を含めるように実験を行なったが，実際の IDE の入力に対してマイニングした場合，実験の結果が変わることが予想される．

8　関連研究

8.1　パターンマイニング

パターンをマイニングする手法として様々な手法が提案されている．複数のアイテムの集合から，頻出部分集合を求めるアルゴリズムとして Apriori [20] と Back Trackiong [25] が挙げられる．Apriori は幅優先で探索するアルゴリズムであり，Back Tracking は深さ優先で探索するアルゴリズムである．

また，複数のアイテムの列から頻出部分列を求めるアルゴリズムとして Prefix Span[26] が提案されている．この Prefix Span はソースコード中のトークンのパターンをマイニングする研究 [27] で利用されている．

頻出部分木をマイニングするアルゴリズムは FREQT 以外に，TreeMiner[28] が挙げられる．TreeMiner は FREQT と同様に最右拡張を行ない，頻出する部分木を求める．しかし FREQT と異なり，TreeMiner は頻出部分木を求める際に親子かどうかではなく，先祖子孫かどうかで部分木のマイニングを行なう．つまり，出力された部分木の親と子の間に，実際には複数のノードが含まれていることを許したアルゴリズムである．統合木の頻出部分木を厳密に求めるために提案手法では FREQT を採用した．

8.2　AST の差分

AST の差分を求めるアルゴリズムとして GumTree 以外にも Change Distiller[29] が挙げられる．しかし GumTree と Change Distiller の編集スクリプトを比較した時に，GumTree の方が無駄なく正確な移動を検出できているといわれている [13]．また多くの言語に対して GumTree の出力する編集スクリプトの方が正確であるといわれている [13]．そのため，TC2P では GumTree を採用した．

8.3　変更パターン

既存研究として Negara らの研究 [10] については 3 章で述べたが，変更パターンをマイニングする研究は他にもある．Nguyen らの研究 [3] は，同一の高さを持つ部分木しかマイニングの候補に含めない．また，Martinez らの研究 [12] でマイニングすることができる変更パターンは，1 つの部分木に対して，追加・削除・更新されたかしかマイニングできない．どちらの研究も制約が存在するため，TC2P の方がより柔軟に変更パターンをマイニングできる．

メソッドの呼び出しの変更パターンをマイニングする手法 [30, 5] も存在する．しかし，これはメソッドの呼び出しに特化した手法であり，提案手法のようにプログラムの全要素に対して変更パターンをマイニングできない．

Coming[31] は Git のリポジトリから特定の変更パターンのインスタンスをマイニングする手法であ

る．この手法は変更パターンではなく，事前に定義された変更をマイニングするという点で TC2P とは異なる．

CPatMiner[11] はプログラム依存グラフを用いて，プログラムの変更をグラフに変換し，そのグラフに対してマイニングすることで変更パターンを求める．しかし，出力される変更パターンはプログラム依存グラフとして出力されるため，その変更パターンを用いて他の研究に応用する場合，その研究でもプログラム依存グラフを計算し，編集する必要がある．一方で TC2P が出力する変更パターンは AST として出力されるため，他の研究でも容易に応用でき，汎用性の観点から提案手法が優れているといえる．

9　あとがき

Git リポジトリから GumTree を用いて編集スクリプトを求め，統合木に変換し，その統合木に対して頻出部分木マイニングを行なうことで，木構造で変更パターンをマイニングする手法を提案した．さらに提案手法を TC2P として実装した．TC2P を評価するために Java の OSS に対して実行し，既存手法の結果と比較した． その結果，既存手法と比べて TC2P はより正確に変更パターンをマイニングできただけでなく，既存手法ではマイニングできていなかった変更パターンもマイニングできた． また，統合木には AST の構造の情報を含んでおり，その統合木に対してマイニングを行なうことで，具体的な変更パターンも抽象的な変更パターンもマイニングできていることがわかった．

提案手法の応用として以下が挙げられる．

変更漏れの修正　図 15 で示された変更パターンのように，null チェックの処理をメソッドに切り出した場合を考える．メソッドを切り出したにも関わらず，メソッドを用いずに null チェックをする処理が残っていた場合，変更パターンを適用することでそのような変更漏れを修正できると考えられる．

より高レイヤーな編集操作　GumTree が出力する編集スクリプトは AST ノードに対しての挿入・削除・移動・更新といった基本的な編集操作で構成されている．変更パターンに名前をつけることができれば，その変更パターンを新しい編集操作とし，開発者がより容易に編集スクリプトを理解できると考えられる．

また，提案手法の課題としては以下が挙げられる．

スケーラビリティ　実験で使用したリポジトリのいくつかはタイムアウトとなってしまった．より大規模なデータセットに対しても提案手法を実行できるよう改善する必要がある．

フィルタリング　提案手法では一定回数以上繰り返された抽象構文木の構造の変更を変更パターンとして出力する．そのため出力された変更パターンには意味を持たないような変更パターンも数多く存在する．そのような無駄な変更パターンをフィルタリングする必要がある．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本 真二 教授に心より感謝申し上げます.

本研究の全過程において，終始熱心なご指導を頂きました，肥後 芳樹 准教授に深く感謝申し上げます.

本研究に関して，的確で丁寧なご助言を頂きました，柗本 真佑 助教に心より感謝申し上げます.

研究室生活の中で相談に乗って頂き，また励まして頂きました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博士前期課程 2 年の田中紘都氏，同土居真之氏，同中島望氏に深く感謝申し上げます.

研究室生活を大変豊かにして頂きました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博士前期課程 1 年の東英明氏，同九間哲士氏，同富田裕也氏，同中川将氏，同華山魁生氏に深く感謝申し上げます.

研究室の環境維持に多くのご助力を頂きました，大阪大学基礎工学部情報科学科 4 年の出田涼子氏，同市川直人氏，同藤本章良氏，同前島葵氏に深く感謝申し上げます.

研究活動を円滑に進めるために様々な形でご助力を頂きました事務補佐員の神谷智子氏に深く感謝申し上げます.

最後に，本研究に至るまでに講義等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます.

参考文献

[1] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Commun. ACM*, Vol. 59, No. 5, p. 122131, April 2016.

[2] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA ’09, p. 8192, New York, NY, USA, 2009. Association for Computing Machinery.

[3] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE’13, p. 180190. IEEE Press, 2013.

[4] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, p. 306317, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, p. 511522, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE ’09, p. 213222, New York, NY, USA, 2009. Association for Computing Machinery.

[7] R. Robbes and M. Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 317–326, Sep. 2008.

[8] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering*, pp. 802–811, May 2013.

[9] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, p. 613624, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 803–813, 2014.

[11] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering*, pp. 819–830. IEEE, 2019.

[12] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *IEEE International Conference on Software Maintenance*, pp. 388–391, Sep. 2013.

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering*, pp. 313–324, 2014.

[14] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pp. 493–504, 1996.

[15] Christian Macho, Shane Mcintosh, and Martin Pinzger. Extracting build changes with builddiff. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pp. 368–378, 2017.

[16] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 740–751, 2017.

[17] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pp. 25–36, 2019.

[18] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, FSE 2016, pp. 144–156, 2016.

[19] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems*, Vol. 87, No. 12, pp. 2754–2763, 2004.

[20] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB ’94, p. 487499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[21] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution*, pp. 334–344. IEEE, 2016.

[22] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. Mining continuous code changes to detect frequent program transformations. Technical report, 2013.

[23] 肥後芳樹, 楠本真二. Java メソッドの高精度な追跡のための細粒度版管理システムの提案. 電子情報通信学会技術報告, 第 118 巻, pp. 133–138, 1 2019.

[24] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: Fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL ’11, p. 96100, New York, NY, USA, 2011. Association for Computing Machinery.

[25] Roberto J Bayardo Jr. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pp. 85–93, 1998.

[26] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pp. 215–224. Citeseer, 2001.

[27] 石尾隆, 伊達浩典, 三宅達也, 井上克郎ほか. シーケンシャルパターンマイニングを用いたコーディングパターン抽出. 情報処理学会論文誌, Vol. 50, No. 2, pp. 860–871, 2009.

[28] Mohammed J Zaki. Treeminer: An efficient algorithm for mining embedded ordered frequent trees. In *Advanced Methods for Knowledge Discovery from Complex Data*, pp. 123–151. Springer, 2005.

[29] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software*

engineering, Vol. 33, No. 11, pp. 725–743, 2007.

[30] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal api usage patterns. In *IEEE/ACM International Conference on Automated Software Engineering*, pp. 456–459, Nov 2011.

[31] M. Martinez and M. Monperrus. Coming: A tool for mining change pattern instances from git commits. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*, pp. 79–82, May 2019.