一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

SZZ手法によるバグ混入コミット推定精度向上のための コンテキストを考慮した開発履歴の提案

近藤　偉成†　近藤　将成††　亀井　靖高††　肥後　芳樹†

† 大阪大学大学院情報科学研究科 〒565-0871 大阪府吹田市山田丘 1-5
†† 九州大学大学院システム情報科学研究院 〒819-0395 福岡県福岡市西区元岡 744
E-mail: †{i-kondou,higo}@ist.osaka-u.ac.jp, ††{kondo,kamei}@ait.kyushu-u.ac.jp

あらまし　SZZ 手法は，対象ソフトウェアの着目したバグについて，そのソフトウェアの開発履歴からそのバグを混入したコミットを自動で推測する方法である．SZZ 手法はソフトウェア工学の研究において広く利用されてはいるが，特定の状況において推測が誤ってしまうことが課題である．この課題を解決するため，先行研究ではソースコードを各行が単一トークンで構成されるように整形した特殊な開発履歴を用いて SZZ 手法を適用する試みを行った．しかし，この特殊な開発履歴を使うことにより，また別の状況においてバグ混入コミットの推測が誤ってしまうことがわかった．そこで本稿では，通常の開発履歴を用いた場合や各行単一トークンの開発履歴を用いた場合にうまくバグ混入コミットを推測できない状況においても，正しくバグ混入コミットを推測できる開発履歴のフォーマットを提案する．また本稿では，68 の OSS プロジェクトを対象に行った，通常の開発履歴，各行単一トークンの開発履歴，本研究で提案するフォーマットの開発履歴を用いて SZZ 手法を適用した比較結果についても報告する．
キーワード　SZZ 手法，バグ混入コミット，バグ修正コミット，コンテキスト

1. はじめに

ソフトウェア開発は人間が手作業によって行うため，開発工程においてソフトウェアに誤り（ソフトウェアのバグ）が混入することは避けられない．ソフトウェア開発者はバグの混入を防ぐためにデバッグ作業を行うが，デバッグ作業は人的・金銭的に大きなコストがかかるため負担が大きい [1]．デバッグ作業のコスト削減のためにバグの分析，予測，修正などのバグに関する研究 [2]〜[6] が行われている．

バグに関する研究を行うために広く利用される手法のひとつに SZZ 手法 [7] がある．SZZ 手法はソフトウェアの開発履歴において，バグを混入したコミットを自動で推測する方法であり，バグに関する研究における分析対象のデータセット構築に利用される [4], [6], [8]．SZZ 手法はソースコードの版管理システムである Git の開発履歴と `blame` という機能 [9] を利用する．Git の開発履歴はコミットという単位で管理され，各コミットはソースコードの変更を行単位の削除および追加で管理する．`blame` は与えられたソースコードの行が追加されたコミットのハッシュ値を表示することができる．SZZ 手法では，バグを修正したコミットにおける変更行にバグが存在していたと仮定し，`blame` を利用して，その行が追加されたコミットをバグ混入コミットとして特定する．

SZZ 手法はバグを混入したコミットを推測するために広く利用されるが，その推測精度は高くないことが報告されている [10]．理由のひとつとして `blame` を利用したバグ混入コミット特定の限界がある [11]．図 1 に Git の開発履歴として 3 つのコミットを示す．コミット A において 10 行目が修正されている．SZZ 手法は 10 行目にバグが混入していたと仮定し，`blame` を利用して 10 行目が追加されたコミット B を特定する．しかし，コミット B では 10 行目に対してコメントの変更しかしておらず，実際のバグ混入はコミット C で発生している．通常の Git の開発履歴は行単位で変更を管理するため，コメント変更などバグとは関係しない変更で，バグ混入コミットの特定に失敗し，SZZ 手法の推測精度の低下につながる．

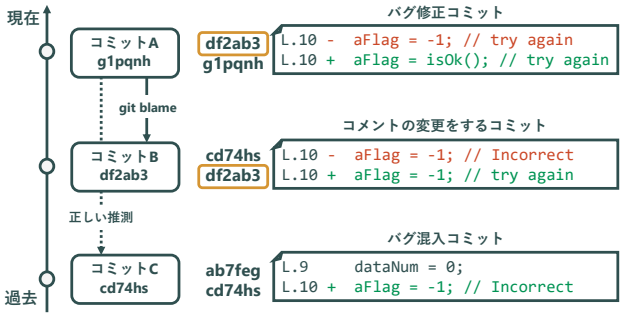

図 1: blame を利用して誤った推測をする例

SZZ 手法の推測精度向上のために，Git の開発履歴を各行が単一トークンで構成される開発履歴に整形する手法が提案されている [12]．単一トークンごとに `blame` を適用できるようになり，コメント変更のようなバグの混入とは関係がない変更の影響を無視できるようになる．しかし，各行単一トークンの開発履歴を利用することで新たな問題が起こることも報告されている [12]．例えば，図 2 のようにバグ修正コミット

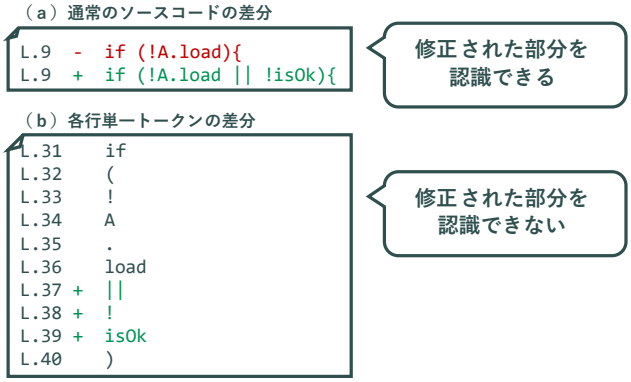

図 2: バグ修正コミットで各フォーマットで認識される差分

の変更が，各行単一トークンでは追加のみで表現される場合である．blame は与えられた行が追加されたコミットのハッシュ値を表示するため，追加行に対して適用してもその行を追加してバグ修正を行ったコミットを指してしまう．

そこで，本研究では，通常の開発履歴および各行単一トークンの開発履歴のどちらにおいても SZZ 手法の推測がうまくいかない状況において，正しくバグ混入コミットを推測できる新たな開発履歴フォーマットを提案する．具体的には，各行単一トークンの開発履歴に対して，各トークンに前後のコンテキストを加える．トークンの前後のコンテキストを加えることで，バグ修正コミットにおいて新たに追加されたトークンが，ソースコード上のどの部分を修正したのかを明確に特定できるようになる。

本研究では，68 のオープンソースソフトウェア (OSS) プロジェクトを対象に各行 3 トークン開発履歴が SZZ 手法に与える影響を調査する．具体的には，通常の開発履歴，各行単一トークンの開発履歴，本研究で提案するフォーマットの開発履歴を用いて SZZ 手法を適用し，結果を比較することで以下の 3 つ研究設問（RQ）に対する回答を得ることを目指す．

RQ1: 各行 3 トークンの開発履歴は SZZ 手法の精度を向上させるのか？

RQ2: 各行 3 トークンの開発履歴を利用することで正しく推測されるバグ混入コミットは，他の開発履歴を利用した場合とどのように異なるのか？

RQ3: 各行 3 トークンの開発履歴を利用することで誤って推測されるバグ混入コミットは，他の開発履歴を利用した場合とどのように異なるのか？

実験の結果，各行 3 トークン開発履歴を利用すると，正しく推測するバグ混入コミットの数を増加させることがわかった．そして，この開発履歴を利用することの利点と欠点を他の開発履歴との比較によって明らかにした．

以降，第 2. 章では本研究の背景を関連研究の観点から述べる．第 3. 章では本研究の提案手法を述べる．第 4. 章では対象のデータセットと実験方法を述べる．第 5. 章では RQ に対する実験結果および分析結果について述べる．第 6. 章では SZZ 手法の精度を向上させるための手法を実験結果より考察する．第 7. 章では結論と今後の課題について述べる．

2. 背　　景

2.1 SZZ 手 法

SZZ 手法 [7] は，開発履歴を持つプロジェクトからバグを混入するコミットを特定するために広く利用されている手法である．具体的には以下の 2 つの手順を持つ．

手順（1）：　バグ修正コミットの識別

手順（2）：　バグ混入コミットの推定

手順（1）でバグ修正コミットを識別し，そこで変更された行を明らかにする．手順（2）では手順（1）で明らかになった変更行を追加した過去のコミットを blame を使って取り出す．取り出されたコミットをバグ混入コミットであるとする．

SZZ 手法は広くソフトウェア工学研究において使われている [5], [6], [8] が，以下の 2 つの課題がありバグ混入コミット推定精度に悪影響を及ぼすことが知られている [10]〜[17].

課題（A）　バグが混入された行が変更されるコミットが複数回あると blame が誤った結果を返す．具体的には，コード整形やリファクタリング，コメントの変更などのバグに関係ない部分の変更である．

課題（B）　行の追加によってバグの修正がされた場合，blame を適用する箇所が特定できず，バグ混入コミットを特定することができない．

これらの課題を解決することが求められる．

2.2 関連研究と課題

SZZ 手法の課題を解決することを目指して研究が行われている [10]〜[17]. Watanabe ら [12] は，課題（A）に対処するため，各行単一トークンの開発履歴を活用した SZZ 手法を提案した．通常の SZZ 手法は，行単位のコミットによって構成される Git の開発履歴を利用して blame によるバグ混入コミットの推定を行う．この手法では，Git の開発履歴を各行単一トークンのコミットによって構成される開発履歴に変換した上で blame を適用する．各行単一トークンの開発履歴は，トークン単位で変更を追跡できるため， 図 1 のような例に対処できる．

しかし，各行単一トークンの開発履歴を使う場合に，通常の開発履歴では発生していなかった課題（B）が発生する場合がある． 図 2（a）のコミット変更を各行単一トークンのコミットに変換すると 図 2（b）になる．通常のコミットでは行の削除と追加による修正として Git が認識するが，各行単一トークンのコミットでは追加のみの修正として Git が認識する．そのため，課題（B）が発生しバグ混入コミットを推定できない．

2.3 解　決　策

図 1 及び 図 2 の課題は各行単一トークンの開発履歴に対して各トークンに前後のコンテキストを加えることで解決できる．具体的には，各行単一トークンの開発履歴を各行 3 トークンの開発履歴に変換する． 図 3 に，課題（A）の状況における通常の開発履歴と各行 3 トークンの開発履歴のコミットを示す．通常の開発履歴では，バグ混入コミットとしてバグの原因である変数 x とは関係ない部分を変更しているコミット B を特定してしまう．各行 3 トークンの開発履歴は行単位よりも細かく変更を追跡することができる．そのため，変数

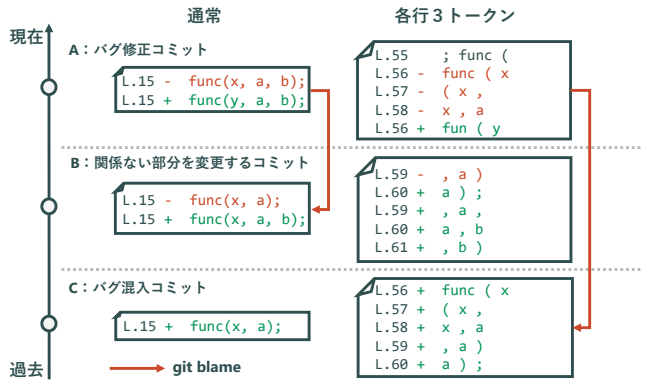

図 3: 課題（A）の状況における各行 3 トークンの blame

x を正しく追跡し変数 x を混入したコミット C をバグ混入コミットとして正しく特定できる．

図 4 は開発履歴を各行 3 トークンに変換した際のコミットの変更である．各行単一トークンでは追加のみの変更になってしまうが，各行 3 トークンとすることで削除と追加を含んだ変更となる．これにより，課題（B）に対応できる．

このように，各トークンの前後のコンテキストを考慮した開発履歴を活用することで SZZ 手法の課題を解決できる．本研究では，各行 3 トークンの開発履歴を活用した SZZ 手法を既存の SZZ 手法と比較することで評価する．

3. 提案手法

本研究では，対象プロジェクトの開発履歴から，各行 3 トークンに変換した開発履歴を作成し，各行 3 トークンの開発履歴に SZZ 手法を実行する手法を提案する．各行 3 トークンの開発履歴は 図 3 や 図 4 のようになる．各行には 3 トークンが含まれており，各トークンは空白によって区切られる．なお，ソースコードファイルの最初のトークンおよび最後のトークンは，前もしくは後のトークンが存在しない．そのため，最初のトークンの前には FileStart，最後のトークンの後には FileEnd という文字列を追加することで 3 トークンとしている．

この各行 3 トークンの開発履歴の作成には，先行研究 [12] で各行単一トークンの開発履歴を作成するために使用されていた cregit [18] というツールに変更を加えたものを使用した．cregit はソースコードファイルを解析し，ソースコードに含まれるすべてのトークンを含むファイルを生成する．生成されるファイルの各行は，各トークンとその種類であり，各行の順序は元のソースコードファイルにあらわれるトークンの順序と一致している． 図 5 の通常のファイルを cregit に与えると，各行単一トークンのファイルが生成される．なお，本書はわかりやすさを考慮し，各行単一トークンのソースコードにはトークンタイプを記載していない．本研究では，cregit の変換先のファイルへ書き込む処理の前に各トークンに対して前後のトークンを追加する処理を行うよう変更し，各行 3 トークンの開発履歴を作成できるようにした． 図 5 の通常のファイルを cregit を変更したツールに与えると，各行 3 トークンのファイルが生成される．

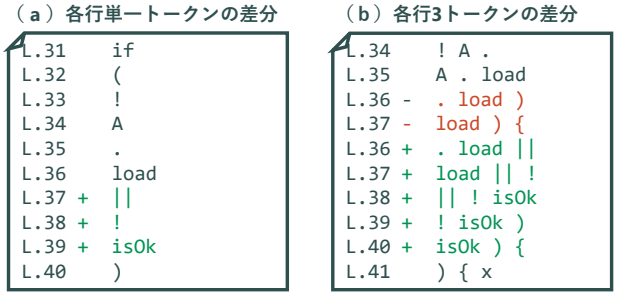

図 4: 課題（B）の状況における各行 3 トークンの差分

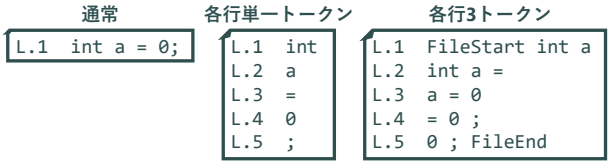

図 5: 各フォーマットのソースコードファイル

4. 実験設定

4.1 データセット

バグ混入コミットの正解データとして，開発者情報付きオラクルデータセット [19] を使用した．開発者によってバグ混入コミットがラベル付されており，8 つの主要なプログラミング言語で書かれた合計 1,854 のリポジトリが含まれている．この実験では，先行研究 [12] で利用されていた，主に Java で開発された 68 の OSS プロジェクトを対象とした．

4.2 実験の流れ

図 6 に実験の流れを示す．まず，SZZ 手法に対する影響の比較を行う 3 つのフォーマットの開発履歴を作成した．3 つのフォーマットの開発履歴とは行単位の開発履歴，トークン単位の開発履歴，そして提案手法である 3-gram の開発履歴である．各開発履歴の作成方法を示す．

行単位の開発履歴は，対象とする 68 の OSS プロジェクトをクローンした通常の開発履歴である．

トークン単位の開発履歴は，対象とする 68 の OSS プロジェクトの開発履歴に対して cregit を使用して作成する．cregit を使用して，各行が単一トークンで構成されたフォーマットの開発履歴に変換する．

3-gram の開発履歴は，対象とする 68 の OSS プロジェクトの開発履歴に対して cregit に変更を加えたツールを使用して作成する．このツールを使用して，各行単一トークンの開発履歴に対して，各トークンに前後のトークンを追加した各行が 3 トークンで構成されたフォーマットの開発履歴に変換する．

次に，PySZZ v2 [19] というツールを 3 つの開発履歴に対して適用する．PySZZ v2 は，入力として Git の開発履歴とバグ修正コミットのハッシュ値を受け取り，バグ混入コミットとして推測したコミットのハッシュ値を返す．

4.3 評　価

本実験では 2 つの観点から各行 3 トークンの開発履歴を活用した SZZ 手法を評価する．

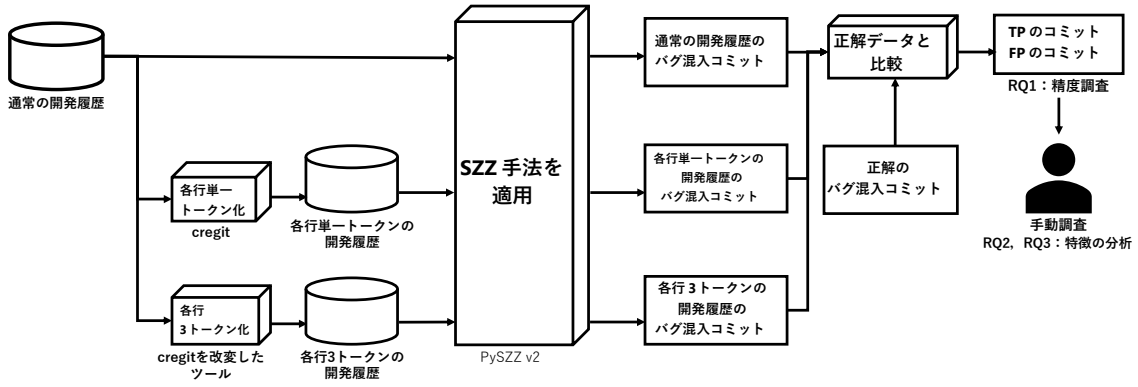

図 6: 実験の流れ

バグ混入コミットの推定精度を評価：SZZ 手法によって推測されたバグ混入コミットと開発者がラベル付したバグ混入コミットを比較し，正しく推測したバグ混入コミットの数（TP），バグ混入コミットと判定したが実際はバグ混入コミットではなかったコミット数（FP），Precision，Recall，F1 score を評価指標として算出する．これを開発履歴ごとに計算する．

各行 3 トークンの開発履歴を活用することで正しく推測できたバグ混入コミットおよび誤ってバグ混入コミットとして特定されたコミットの特徴の目視分析：目視でこれらのコミットのソースコードに対する変更を分析することで，開発履歴の違いが及ぼす影響を明らかにする．

5. 実験結果

5.1 RQ1: 各行 3 トークンの開発履歴は SZZ 手法の精度を向上させるのか？

発見 1：各行 3 トークンの開発履歴フォーマットは正しく特定できたバグ混入コミットの数を増加させる． 表 1 は各開発履歴フォーマットに対して SZZ 手法を適用した時のバグ混入コミットの推測結果の評価値である．各行 3 トークンを活用すると，TP の件数は行単位の開発履歴より 2 件多く，トークン単位の開発履歴より 12 件多い 53 件であった．3 つの開発履歴の中で最も多くのバグ混入コミットを特定できた．

発見 2：各行 3 トークンの開発履歴フォーマットはバグ混入コミットの推定精度を低下させる．FP の数は行単位の開発履歴より 39 件多く，トークン単位の開発履歴より 25 件多い 137 件であった．バグ混入コミットではないコミットに対してバグ混入コミットと推測してしまう件数が，3 つの開発履歴フォーマットの中で最も多い．加えて，行単位の開発履歴と比較して Precision は 0.063 低下し，F1 score は 0.061 低下する．以上から，各行 3 トークンの開発履歴は SZZ 手法の精度を向上させることはできなかったといえる．

表 1: 異なる開発履歴フォーマットごとのバグ混入コミット推測結果の評価指標値

フォーマット	TP	FP	Precision	Recall	F1 Score
行	51	98	0.342	0.671	0.459
トークン	43	112	0.277	0.566	0.372
3-gram	**53**	137	0.279	0.694	0.398

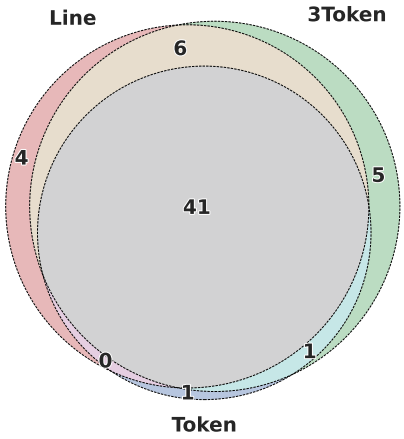

図 7: 各変更履歴の TP の包含関係

5.2 RQ2: 各行 3 トークンの開発履歴を利用することで正しく推測されるバグ混入コミットは，他の開発履歴を利用した場合とどのように異なるのか？

発見 3：各行 3 トークンのフォーマットを使うことで，課題（B）によって見逃していたバグ混入コミットを発見可能になる． 図 7 は通常の開発履歴（Line，赤色），各行単一トークン（Token，青色），および，各行 3 トークン（3Token，緑色）それぞれの TP の包含関係を図示している．各行 3 トークンのみで発見可能であったバグ混入コミットは 5 件であった．これらは全て課題（B）によって発見されていなかった．つまり，通常の開発履歴では追加のみで修正されているバグ修正コミットに対応するバグ混入コミットであり，`blame` が失敗するコミットである． 図 8 は対象バグ混入コミットのバグを修正したコミットの差分の一部である．通常のコミットでは追加のみの修正であるが，各行 3 トークンのコミットでは削除と追加による修正となる．そのため，各行 3 トークンでは削除行に `blame` を適用してバグ混入コミットを特定できる．

発見 4：各行 3 トークンのフォーマットを使うことで，各行単一トークンの開発履歴で新たに発生する課題（B）を回避しバグ混入コミットを発見できる．通常の開発履歴および各行 3

```
 9     if(a == 0){
10       b = 0;
11 +     break;
12     }
13     x = 0;
```

(a) 通常の差分

```
35     b      =      0
36     =      0      ;
37 -   0      ;      }
37 +   0      ;      break
38 +   ;      break  ;
39 +   break  ;      }
40     ;      }      x
```

(b) 各行 3 トークンの差分

図 8: 行が追加されて各行 3 トークンの差分が削除を含む例

```
1  + func(a, c, null);
2    func(a, d, null);
3    func(b, c, null);
4  - func(b, d, null);
```

(a) 通常の差分

```
2      func (      a
3      (      a      ,
4  -   a      ,      d
5  -   ,      d      )
6  -   d      ,      p
4  +   a      ,      c
5  +   ,      c      )
6  +   c      ,      null
7      ,      null      )
```

(b) 各行 3 トークンの差分

図 9: 通常と各行 3 トークンで差分の取り方が異なる例

トークンでのみ発見可能であったバグ混入コミットは 6 件であった．これらは全て各行単一トークンの開発履歴においてトークンの追加のみとなるバグ修正コミットに対応するバグ混入コミットである．つまり，各行単一トークンの開発履歴にすることで新たに課題（B）が発生していたことを示す．

発見 5：各行 3 トークンのフォーマットを使うことで，Git の差分の取り方に影響を与え，通常の開発履歴では見逃すバグ混入コミットを発見可能になる場合と，見逃す場合が存在する．各行単一トークンおよび各行 3 トークンでのみ発見可能であったバグ混入コミットは 1 件であった．図 9 は対象バグ混入コミットのバグを修正したコミットの差分の一部である．通常のコミットでは 4 行目を削除し，新たなソースコードを 1 行目に追加する形で Git が差分が取られる．一方，各行 3 トークンでは関数呼び出しにおける引数を書き換える形で差分が取られる．この差分の取り方の違いにより，各行単一トークンおよび各行 3 トークンでのみでこのコミットに対応するバグ混入コミットが発見できた．一方で，各行単一トークンでのみ発見されたバグ混入コミット 1 件も同様に差分の取り方によって発見可能になった．そのため，差分の取り方の違いによって良くなる場合と悪くなる場合の双方が観察された．

発見 6：各行 3 トークンでは，コード整形に関する変更を捉えられず推定できないバグ混入コミットがある．

通常の開発履歴でのみ発見可能であったバグ混入コミットは 4 件である．このうち 2 件は if 文の追加によってバグ修正が行われている．この時，if 文追加されたことによって，インデントが追加される．このインデントが追加されたコードを

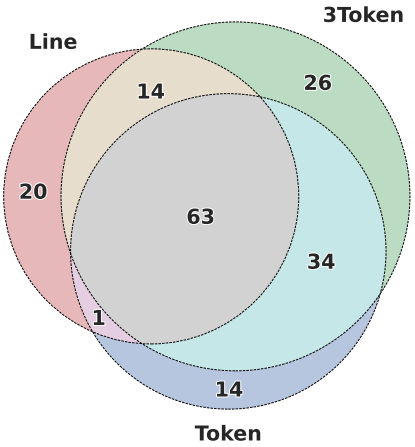

図 10: 各変更履歴の FP の包含関係

blame によって追跡することでバグ混入コミットが見つかる．各行 3 トークンの開発履歴ではインデントの変更は追跡できず，このバグ混入コミットを見逃してしまう．残りの 2 件は，バグ修正コミットで削除された空行に対して blame を適用することで発見されるバグ混入コミットである．このようにインデントや空行などコード整形に関する変更に関連するバグ混入コミットを各行 3 トークンでは見逃してしまう．

5.3 RQ3: 各行 3 トークンの開発履歴を利用することで誤って推測されるバグ混入コミットは，他の開発履歴を利用した場合とどのように異なるのか？

発見 7：各行 3 トークンでは blame の追跡対象が増えることにより誤ってバグ混入コミットと推定するコミット数が増加する．図 10 は通常の開発履歴（Line，赤色），各行単一トークン（Token，青色），および，各行 3 トークン（3Token，緑色）それぞれの FP の包含関係図示している．各行 3 トークンでは 26 件の新たな FP を発生させている． 図 11 は左側に通常の開発履歴，右側に各行 3 トークンに変換した開発履歴を載せている．A，B，および，C の 3 つのコミットが載っており，A がバグ修正コミット，C がバグ混入コミットである．通常の開発履歴の場合，A の修正行から C のみをバグ混入コミットとして特定可能である．一方で，各行 3 トークンとした場合，C を誤ってバグ混入コミットとして特定してしまう．なお，発見 3 および発見 4 からわかる通り，この影響により正しく推定可能なバグ混入コミットも存在する．発見 3 および 4 と発見 7 はトレードオフの関係にあると考えられる．

発見 8：通常の開発履歴では空白や改行などのコード整形に関する変更を追跡したことが原因で誤ったバグ混入コミットの推定を行うケースが存在する．発見 6 とトレードオフの関係にある発見である．

発見 9：各行単一トークンでは括弧（例:(),{}）やセミコロンなどの特定のトークン追跡した結果誤ったバグ混入コミットの推定を行うケースが存在する．括弧やセミコロンなどのトークンを変更するコミットが少ないため，発見するべきバグ混入コミットを見逃してしまうことが原因のひとつと考えられる．

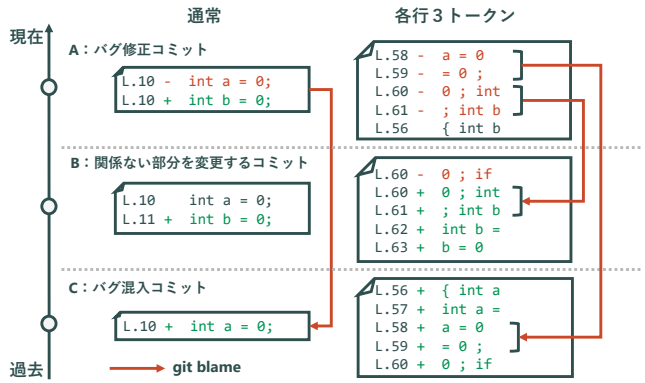

図 11: 各行 3 トークン開発履歴で誤った特定をする例

6. 議　　論

本実験の発見は各行 3 トークンの開発履歴が SZZ 手法の精度向上に与える影響が限定的であることを示している．発見 1，3，4 のように精度向上に寄与するケースと，発見 2，5，6，7 に示すような精度を低下させるケースが観察された．精度を低下させるケースに対応する手法の研究が今後求められる．

本実験の発見から考えられる今後の研究方針のひとつは，推測されたバグ混入コミットをフィルタリングすることである．例えば，発見 9 より，括弧やセミコロンなどの特定のトークンを追跡したことで推測されるバグ混入コミットはしばしば誤りであることがわかっている．この傾向を利用したフィルタリングとして，各行単一トークンの開発履歴を活用した SZZ 手法で推測されたバグ混入コミットから，"}"と";"の追跡によって推測されたコミットの除去が考えられる．実際，除去を行うと各行単一トークンの開発履歴を活用した SZZ 手法の TP の数は 43 件で変化せず，FP の数は 112 件から 4 件減少し 108 件となった．同様のフィルタリングを各行 3 トークンの開発履歴を活用した SZZ 手法に応用することで FP の数を減らせる可能性がある．

7. おわりに

本研究では，SZZ 手法の精度向上を目指し，従来の行単位および各行単一トークンによる開発履歴の課題を克服するため，各行 3 トークンを基本単位とした新たな開発履歴フォーマットを提案した．そして，通常の開発履歴，各行単一トークンの開発履歴，各行 3 トークンの開発履歴が SZZ 手法に与える影響を比較した．その結果，各行 3 トークンの開発履歴は正しく推測したバグ混入コミットの数を増加させるが，バグ混入コミットの推定精度は低下させることがわかった．今後の研究課題として，精度を低下させるケースに対応するため，各行に含めるトークン数を変化させたときの推定精度の影響分析や，誤った推測を減らすためのフィルタリング手法の開発などが考えられる．

謝辞　本研究の一部は，JSPS 科研費 JP21H04877, JP21K18302, JP22K11985, JP22K18630, JP22K17874, 23K24823, 24H00692, JP24K02921, JSPS 二国間交流事業 JPJSBP120239929, 国立研究開発法人科学技術振興機構（JST）先端国際共同研究推進事業（ASPIRE）グラント番号 JPMJAP2415，及び，稲盛財団の稲盛科学研究機構（InaRIS: Inamori Research Institute for Science）フェローシップの助成を受けた．

文　　献

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Increasing software development productivity with reversible debugging”. https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf

[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” Empir. Softw. Eng., vol.19, no.6, pp.1665–1705, 2014.

[3] C.L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” Proc. of the 34th International Conference on Softw. Eng., pp.3–13, 2012.

[4] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, “Automatic program repair,” IEEE Softw., vol.38, no.04, pp.22–27, 2021.

[5] M.L. Bernardi, G. Canfora, G.A. Di Lucca, M. Di Penta, and D. Distante, “Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla,” Proc. of the 16th Eur. Conference on Softw. Maintenance and Reeng., pp.139–148, 2012.

[6] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” IEEE Trans. on Softw. Eng., vol.39, no.6, pp.757–773, 2013.

[7] T.Z. J. Sliwerski and A. Zeller, “When do changes induce fixes,” ACM SIGSOFT Softw. Eng. Notes, vol.30, no.4, pp.1–5, 2005.

[8] R.M. Karampatsis and C. Sutton, “Manysstubs4j dataset (2.0)”. https://doi.org/10.5281/zenodo.3653444

[9] S. Chacon and J. Long, “Git - git-blame documentation,” 2024. https://git-scm.com/docs/git-blame

[10] S. Kim, T. Zimmermann, K. Pan, and E.J. Jr. Whitehead, “Automatic identification of bug-introducing changes,” Proc. of the 21st IEEE/ACM International Conference on Automated Softw. Eng., pp.81–90, 2006.

[11] C. Rezk, Y. Kamei, and S. McIntosh, “The ghost commit problem when identifying fix-inducing changes: An empirical study of apache projects,” IEEE Trans. on Softw. Eng., vol.48, no.9, pp.3297–3309, 2022.

[12] H. Watanabe, M. Kondo, E. Choi, and O. Mizuno, “Benefits and pitfalls of token-level szz: An empirical study on oss projects,” Proc. of the IEEE International Conference on Softw. Analysis, Evol. and Reeng., pp.776–786, 2024.

[13] C. Williams and J. Spacco, “Szz revisited: verifying when changes induce fixes,” Proc. of the 2008 Workshop on Defects in Large Softw. Syst.s, pp.32–36, 2008.

[14] S. Davies, M. Roper, and M. Wood, “Comparing text-based and dependence-based approaches for determining the origins of bugs,” J. of Softw.: Evol. and Process, vol.26, no.9, pp.117–139, 2014.

[15] T. Zimmermann, S. Kim, A. Zeller, and E.J. Whitehead, “Mining version archives for co-changed lines,” Proc. of the 2006 International Workshop on Mining Softw. Repos., pp.72–75, 2006.

[16] E.C. Neto, D.A.d. Costa, and U. Kulesza, “Revisiting and improving szz implementations,” Proc. of the ACM/IEEE International Symp. on Empir. Softw. Eng. and Measurement, pp.1–12, 2019.

[17] L. Tang, L. Bao, X. Xia, and Z. Huang, “Neural szz algorithm,” Proc. of the 38th IEEE/ACM International Conference on Automated Softw. Eng., pp.1024–1035, 2023.

[18] D.M. German, B. Adams, and K. Stewart, “ “cregit: Token-level blame information in git version control repositories,” Empir. Softw. Eng., vol.24, no.4, pp.2725–2763, 2019.

[19] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, “A comprehensive evaluation of szz variants through a developer-informed oracle,” J. of Syst.s and Softw., vol.202, p.111729, 2023.