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

関数の依存関係に基づく探索範囲を拡張したSZZ手法の初期調査

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 手法はバグ混入コミットの特定に広く利用されているが，変更行のみを追跡対象とするため真のバグ混入コミットを見逃すという課題がある．一方で，見逃しを防ぐために単純に探索範囲を拡張することは無関係なコミットの誤検出を招く．そこで本研究では，関数の依存関係を用いた探索範囲の拡張と，大規模言語モデルによる修正内容の意味的評価を組み合わせた新たな手法 DEP-SZZ を提案する．68 の Java OSS プロジェクトを対象とした初期調査の結果，依存関係に基づく探索範囲の拡張によりバグ混入コミットの検出数は 15 件増加したものの，同時に追跡行数も約 1,215 倍と大幅に増大した．しかし，LLM による絞り込みを適用することで，既存手法に対する検出数の優位性（+6 件）を確保しつつ，解析行数の増加幅を約 4.2 倍へと大幅に抑制できることが示された．

キーワード　SZZ 手法，依存関係，LLM

1. はじめに

ソフトウェア開発は人間が行う活動である以上，ソースコードへのバグ混入を完全に回避することは困難である．この問題に対処するために開発者はデバッグを行うが，これには多大なコストを要する．これを受け，バグ分析，バグ予測，自動プログラム修正など，デバッグコストを削減するための様々なアプローチが研究されてきた [1].

SZZ 手法は，バグ関連の研究で使用されるデータセットを構築するために広く利用されている手法である．SZZ 手法では，Git のようなバージョン管理システム上の開発履歴を分析することで，バグの原因となるコードを導入したバグ混入コミットを特定する．具体的には，まずバグを修正したコミットを特定し，そのコミットで変更された行に対して `blame` コマンドを適用する．このコマンドは，特定の行を最後に変更したコミットのハッシュ値を返すため，SZZ 手法はこれを利用して修正対象行の導入元となったコミットをバグ混入コミットとして特定する．

従来の SZZ 手法は，バグ混入コミットで直接変更された行のみを追跡してバグ混入コミットを推測する．しかし，この方法では真のバグ混入コミットを完全には捉えきれず，見逃しが発生するという課題がある [2]〜[4]．例えば，修正された行そのものではなく，同一関数内や同一ファイル内，さらには別ファイルに存在する行を追跡することで，真のバグ混入コミットを特定できるケースが報告されている [4]．しかし，これらの見逃されたバグ混入コミットを特定するために単に探索範囲を広げるだけでは，バグとは無関係なコミットが多く含まれてしまうため実用的ではない．

そこで本研究では，関数の依存関係（コールグラフ）を利用した探索範囲の拡張と，大規模言語モデル（LLM）によるバグ修正に関する本質的な変更箇所の特定を組み合わせた新たな SZZ 手法を提案する．本手法は，以下の 3 つのフェーズで構成される．(1) コールグラフを用いた探索範囲の拡張，(2) LLM を用いた，バグ修正の本質的な変更を含む「コア修正関数」の特定，(3) LLM を用いた，コア修正関数を起点とした依存先関数の選別および詳細な追跡行の特定である．

本稿は，提案手法の実現に向けた初期調査と位置付けられる．具体的には，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，フェーズ 1 およびフェーズ 2 の有効性を検証するため，以下の研究設問（RQ）を設定する．

RQ1: 関数の依存関係に基づく探索範囲の拡張は，SZZ 手法のバグ混入コミット検出能力にどのような影響を与えるのか？

RQ2: LLM による修正内容の重要度に基づく絞り込みは，SZZ 手法のノイズをどの程度抑制できるのか？

なお，フェーズ 3 に関しては，RQ2 の結果に基づいた予備調査を実施し，7. 章の議論にてその結果について報告する．

2. 背　　景

2.1 SZZ 手法

SZZ 手法 [5] は，ソフトウェアの開発履歴からバグ混入コミットを自動で推定するための手法である．SZZ 手法は以下の 2 つのステップで構成される．

1. **バグ修正コミットの特定：**バグ修正コミットは，バグトラッキングシステムの情報やコミットメッセージに基づいて特定される．
2. **バグ混入コミットの特定：**バグ修正コミットで変更された行に対して `blame` コマンドを適用し，その行を最後に変更したコミットをバグ混入コミットとして特定する．

SZZ 手法は大規模なデータセット構築に有用である一方で，その推定精度には課題がある．無関係なコミットをバグ混入コミットとして誤って特定してしまう偽陽性の問題に加え，真のバグ混入コミットを特定できず見逃してしまう偽陰性の問題が指摘されている [2]〜[4].

2.2 SZZ 手法の精度改善とノイズ抑制

SZZ 手法のバグ混入コミット推定精度を向上させるために，様々なフィルタリング手法が提案されてきた．Kim ら [6] が提案した **AG-SZZ** は，アノテーショングラフを用いることで，空白やコメントといったプログラムの動作に影響しない変更を除外する手法である．Da Costa ら [7] が提案した **MA-SZZ** は，AG-SZZ では誤検知してしまうブランチ操作やマージコミットなどのメタ変更を適切に処理するために，グラフ構造を改良した手法である．

さらに近年では，大規模言語モデル（LLM）を用いた手法も提案されている． **LLM4SZZ** は，バグ修正コミットのメッセージやコード変更の内容からバグの根本原因を分析し，真にバグの原因となっている行を特定して `blame` を行う手法である．加えて，`blame` によって得られたバグ混入コミット候補に対しても，LLM でバグ修正コミットとの意味的な関連性を評価することでノイズを抑制する．これにより，従来のヒューリスティックなルールでは判別が困難であったノイズを効果的に排除し，高い精度でバグ混入コミットを特定できることが示されている．

2.3 バグ混入コミットの見逃しと探索範囲の拡張

SZZ 手法は，アルゴリズムや探索範囲の制約により，バグ混入コミットを見逃してしまう課題がある．

Ghost commit [3] は，SZZ 手法の仕組み上，特定が不可能なケースを指す．これには 2 つのパターンがある．1 つ目は，バグ修正コミットが行の追加のみで構成されており，追跡の起点となる削除行が存在しない場合である．2 つ目は，バグ混入コミットが行の削除のみをしており，追加行が存在しない場合である．これら 2 つのうち，1 つ目のケースの方が頻度が高いことが報告されている．

一方で，Ghost commit には該当しない場合であっても，従来の SZZ 手法では真のバグ混入コミットを特定できないケースが存在する． **TC-SZZ** [4] は，このような見逃しを防ぐために，バグ修正コミットで変更された行だけでなく，同一関数内や同一ファイル内に存在する行に対しても `blame` を適用する手法である．この研究により，真のバグ混入コミットをより多く特定するためには変更行のみを追跡するだけでは不十分であり，変更行以外にも探索範囲を広げる必要があることが示された．しかし，TC-SZZ のように探索範囲を広げることは，同時にバグとは無関係なコミットを大量に候補に含めることにつながるため，実用的な精度を維持することが困難であるという課題が残る．

2.4 動　　機

SZZ 手法の改善において，適合率の向上と再現率の向上はトレードオフの関係にある．既存の LLM を用いた手法は高いノイズ除去性能を持つ一方で，その探索範囲は修正箇所周辺の文脈に限られるため，より広範な依存関係に起因する見逃しを捉えることは難しい．対照的に，TC-SZZ のように探索範囲を単純に拡張するアプローチは，見逃しを低減できる反面，ノイズの増大を招き実用的な精度を損なうという課題がある．

したがって，真のバグ混入コミットを高精度に特定するためには，探索範囲を論理的に拡張して見逃しを防ぐと同時に，拡張によって生じたノイズを強力に抑制する新たなアプローチが不可欠である． そこで本研究では，関数の依存関係を表すコールグラフに基づいて探索範囲を拡張し，かつ LLM の文脈理解能力を用いてフィルタリングを行う手法を提案する．このアプローチにより，従来手法が見逃していたコミットを捉えつつ，高い精度を維持することを目指す．

3. 提 案 手 法

本研究では，関数の依存関係と LLM による意味解析を統合したバグ混入コミット推定手法 **DEP-SZZ**（Dependency-aware SZZ with LLM）を提案する．本手法は，従来の SZZ 手法が抱える，探索範囲の狭さに起因する見逃しと単純な探索範囲拡大によるノイズ増加というトレードオフの解消を目的とする．DEP-SZZ は，図 1 に示す 3 つのフェーズで構成される．

3.1 依存関係に基づく探索範囲の拡張

DEP-SZZ は，まずバグ修正コミットにおける変更行を含む関数（修正関数）を特定し，さらにその関数と依存関係にある関数（依存関係関数）を収集する．従来の SZZ 手法では削除行のみを追跡していたが，本手法では Ghost commit に対処するため，削除行だけでなく追加行を含む関数も収集の対象とする．具体的には，バグ修正コミットの親コミットに対して抽象構文木（AST）を用いた解析を行い，修正関数を呼び出し・被呼び出し関係にある関数を特定し，コールグラフを作成する．これにより，修正箇所と論理的に繋がりのある関数群を探索候補として初期プールに追加する．

3.2 重要度に基づく修正箇所の特定

バグ修正コミットには，バグの本質的な修正以外にも，リファクタリング，フォーマット修正，テストコードの追加など，バグ混入の特定には寄与しない変更が含まれることが多い．そこで DEP-SZZ では，LLM を用いて変更内容の意味的に分類し，バグの根本原因を含む関数を特定・順位付けする．

まず，バグ修正コミットの変更差分を解析し，連続する変更行の塊であるチャンクのリストを作成する．各チャンクに対して LLM を用い，以下の 3 つのカテゴリのいずれかに分類する．

Core（本質的修正）: バグを修正するために不可欠な変更．ロジックの欠陥修正や条件分岐の変更などが該当する．入力されたチャンクリストのうち，少なくとも 1 つのチャンクが Core としてラベル付けされる制約を設ける．

Surrounding（周辺修正）: Core の修正に伴って必要となった補助的な変更．変数のリネーム，ログ出力の変更，あるいは Core 修正に付随する軽微な調整などが含まれる．

Unrelated（無関係）: バグ修正とは直接関係のない変更．空白

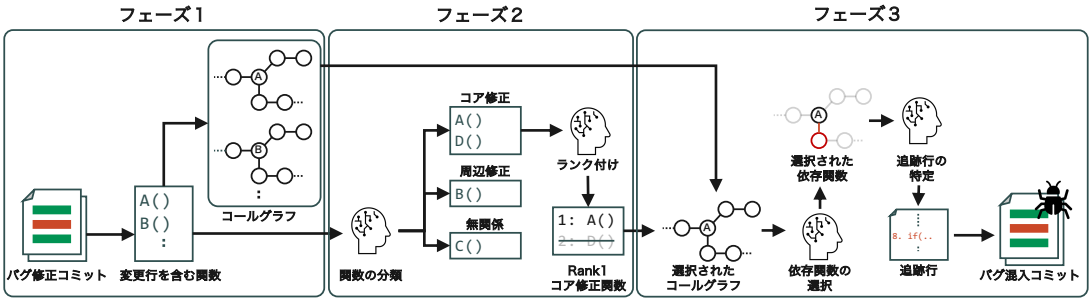

図 1: 提案手法

の調整，コメントの修正，独立したリファクタリングなどが該当する．

次に，分類されたチャンクをそれを包含する関数へとマッピングし，「Core」に分類されたチャンクを 1 つ以上含む関数を**コア修正関数**として定義する．1 つのバグ修正コミットにおいて複数のコア修正関数が特定される場合，それらの中にはバグの根本原因を修正しているものもあれば，その影響波及箇所を修正しているだけのものも含まれる．

本手法がチャンク単位ではなく関数単位でランク付けを行う動機は，コンテキストの拡張にある．断片的なチャンク情報のみでは，その変更がバグの根本原因であるか否かを判断するための情報が不足する場合がある．これに対し，関数単位で評価を行うことで，LLM は変更箇所を含む関数全体のロジックや制御構造といったより広範な文脈を考慮することが可能となる．

そこで本手法では，特定されたコア修正関数のリストと各関数の変更内容を LLM に入力し，「どの関数がバグの根本原因を修正しているか」という観点で評価させる．これにより，最も優先的に調査すべき関数を順位付けし，探索の優先順位を決定する．

3.3 追跡対象の最適化

最後に，ランク付けされた上位のコア修正関数と 3.1 節で収集したその依存関係関数の中から，実際に `blame` を適用して追跡すべき行を決定する．

単にすべての依存関係関数を辿ると探索範囲が広がりすぎるため，ここでも LLM を活用したフィルタリングを行う．具体的には，以下の手順で絞り込みを行う．

（1） **依存関係の選別:** コア修正関数の修正内容に基づき，その依存関係にある関数のうちバグの原因が潜んでいる可能性が高い関数を LLM に選択させる．

（2） **追跡行の特定:** 選別された依存関係関数およびコア修正関数の中で，具体的にどの行を追跡すべきかを LLM に決定させる．ここでは，単純な変更行だけでなくバグのトリガーとなっている未変更の行も選択の対象とする．

以上のステップにより，DEP-SZZ は広範な依存関係を考慮しつつも，LLM の推論能力によって探索対象を論理的に絞り込むことで，高精度かつ高再現率なバグ混入コミットの特定を実現する．

4. 実験設計

4.1 研究設問

3. 章で述べた通り，提案手法である DEP-SZZ は 3 つのフェーズで構成される．本実験では，手法の基盤となるフェーズ 1 およびフェーズ 2 の有効性検証に焦点を当てる．なお，追跡対象の最適化を行うフェーズ 3 については，7. 章の議論において予備調査として扱い，詳細な評価は今後の課題とする．

具体的には，以下の 2 つの研究設問を設定する．

RQ1: 関数の依存関係に基づく探索範囲の拡張は，SZZ 手法のバグ混入コミット検出能力にどのような影響を与えるのか？
本設問は，探索範囲の拡張を行うフェーズ 1 を評価対象とする．ここでは，探索範囲の拡張による検出能力（TP）の向上と，それに伴う解析行数の増加を定量的に評価し，両者のトレードオフを明らかにする．

RQ2: LLM による修正内容の重要度に基づく絞り込みは，SZZ 手法のノイズをどの程度抑制できるのか？
本設問は，重要度に基づく特定を行うフェーズ 2 を評価対象とする．ここでは，LLM による「コア修正関数の特定」および「ランク付け」が，検出能力（TP）と，探索コストとしての解析行数および関数の数に与える影響を検証する．

4.2 データセット

本研究では，開発者情報付きオラクルデータセット [8] を使用した．このデータセットは，主要な 8 つのプログラミング言語で記述された 1,854 の OSS プロジェクトを含んでおり，開発者の手動検証によってバグ修正コミットとバグ混入コミットの対応関係が高い信頼性で紐付けられている．本実験では，これらのうち Java で記述された 68 の OSS プロジェクトを抽出して解析対象とした．

4.3 LLM の構成

提案手法のフェーズ 2 における変更内容の重要度評価および分類タスクには，OpenAI 社の大規模言語モデル **GPT-5.2**（モデル ID： `gpt-5.2-2025-12-11`）を採用した．GPT をはじめとする大規模言語モデル（LLM）は，ソースコードの理解や文脈を考慮した推論において高い性能を示すことが知られており，この特性は本研究の目的であるノイズ除去と本質的修正の特定に適している．そこで本実験では，現時点で最新のモデルを利用することで，その能力を最大限に活用する．本実

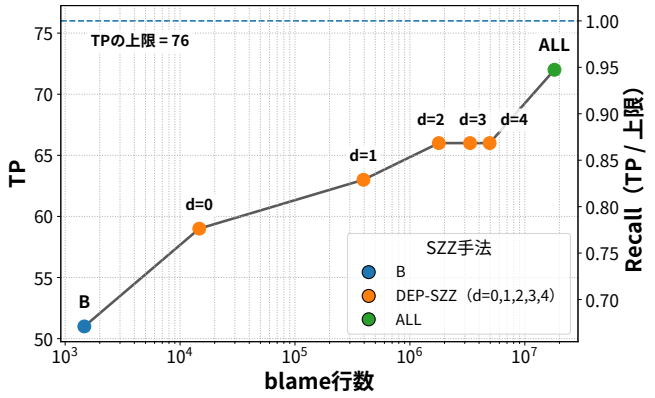

図 2: TP 数と解析行数の関係

験では結果の再現性を最大化するため，LLM の生成パラメータである Temperature は 0 に設定し，ランダム性を排除した決定論的な出力を得られるように構成した．

4.4 評 価 指 標

本研究では，提案手法の有効性を測るため，以下の 2 つの指標を用いる．

検出能力（True Positive: TP）： バグ修正コミット単位の検出成功数．手法が特定した候補コミットの中に，真のバグ混入コミットが 1 つでも含まれる場合に TP として計上する．

探索コスト（解析行数および関数数）： `blame` を実行した行数および関数の総数．これらは調査範囲の広さを表す指標として用いる．

5. RQ1: 関数の依存関係に基づく探索範囲の拡張は，SZZ 手法のバグ混入コミット検出能力にどのような影響を与えるのか？

本節では，関数の依存関係に基づく探索範囲の段階的な拡張がバグ検出能力と探索効率に与える影響を定量的に評価する．

5.1 実 験 手 順

本実験では，検出能力（TP）と探索コスト（解析行数）の観点から，以下の 3 つのアプローチを比較する．

B-SZZ: 変更された行のみを追跡対象とする従来の SZZ 手法．本実験では比較の基準となるベースラインとして使用する．

DEP-SZZ: 提案手法．変更行を含む関数を起点とし，コールグラフ上の依存関係を辿って探索範囲を段階的に拡張する．本手法は関数単位で解析を行うため，対象関数に含まれる全ての行に対して `blame` を実行する点が B-SZZ と異なる．本実験では依存関係の深さを 0 から 4 まで変化させて評価を行う．ここで深さ 0 は修正関数自身であり，深さ 1 は修正関数と直接の依存関係にある関数，深さ 2 はさらにその先の一段階深い依存関係にある関数を指す．

All-SZZ: プロジェクト内の全ソースファイルを対象に `blame` を実行するアプローチ．探索範囲を最大化した場合の理論的な検出上限を確認するために用いる．

5.2 結　　果

DEP-SZZ は B-SZZ と比較してより多くのバグ混入コミットを検出した． 図 2 に，各手法における TP 数と解析行数を示す．B-SZZ の TP 数が 51 件であったのに対し，DEP-SZZ は依存関係の深さ 0 で 59 件を検出し，ベースラインを上回った．さらに，深さ 1 では 63 件，深さ 2 では 66 件と探索範囲を広げるにつれて TP 数は増加した．しかし，深さ 2 以降は TP 数が 66 件で変化せず，検出能力は飽和する傾向が見られた．一方，探索範囲を最大化した All-SZZ は 72 件の TP を検出しており，B-SZZ と比較して 21 件の増加となった．この結果から，B-SZZ が見逃しているバグ混入コミットの多くは，変更行の周辺や依存関係先に存在することが確認できる．

探索範囲の拡大に伴い，解析行数は急激に増加する． 図 2 に示すように，B-SZZ の解析行数が 1,468 行であったのに対し，深さ 0 の DEP-SZZ では 14,686 行と約 10 倍の行数を解析した．DEP-SZZ において最大の TP 数を記録した深さ 2 では 1,783,667 行となり，これは B-SZZ の約 1,215 倍に相当する．さらに All-SZZ では 18,117,530 行（B-SZZ の約 12,340 倍）への解析が必要となる．

DEP-SZZ は All-SZZ に近い検出能力を持ちながら，探索効率において優れている． All-SZZ は 72 件の TP を検出するために膨大な行数を探索する必要がある．これに対し，DEP-SZZ（深さ 2）は All-SZZ の約 10 分の 1 の探索行数で，All-SZZ が検出した TP の約 92%（66/72 件）をカバーした．これは，無差別に探索範囲を広げるのではなく，関数の依存関係に基づいて探索を行うことが，見逃しの抑制と探索コストの抑制のバランスにおいて有効であることを示している．

RQ1 への回答

関数の依存関係を考慮して探索範囲を拡張した DEP-SZZ は，B-SZZ と比較して最大 15 件多くのバグ混入コミットを特定した．この拡張により解析行数は B-SZZ より大幅に増加するものの，全ファイルを探索する All-SZZ の約 10 分の 1 のコストで同手法が検出したバグ混入コミットの約 92% を網羅している．

6. RQ2: LLM による修正内容の重要度に基づく絞り込みは，SZZ 手法のノイズをどの程度抑制できるのか？

本節では，LLM を用いた重要度評価が，拡張された探索範囲からのノイズ除去と真のバグ混入コミットの特定にどの程

表 1: 各カテゴリの関数における TP とコスト

対象	関数数	解析行数	TP
DEP-SZZ（$d = 0$）	475	14,686	59
変更行を含む関数	475	12,910	56
コア修正関数	**197**	**6,976**	**54**
周辺修正関数	230	5,030	14
無関係関数	48	904	1

表 2: ランク順の探索範囲拡張における TP とコスト

	コア修正関数			コア修正関数 + 呼び出し元			コア修正関数 + 呼び出し先			コア修正関数 + 両方		
対象 (k)	関数数	解析行数	TP	関数数	解析行数	TP	関数数	解析行数	TP	関数数	解析行数	TP
Top 1	70	4,338	50	2,009	103,756	54	13,224	94,033	54	15,086	190,236	57
Top 2	92	5,181	54	2,050	105,223	55	14,458	100,945	57	16,339	197,767	58
Top 3	106	5,418	54	2,343	117,919	56	14,864	104,147	57	17,011	213,130	59
Top 5	124	6,153	54	2,382	119,711	56	15,311	107,887	58	17,479	217,874	60
Top 10	152	6,866	54	2,492	122,600	56	15,553	110,144	59	17,802	222,195	60
All Core	197	7,442	54	2,702	126,188	56	16,156	115,051	59	18,564	230,048	60

※ 拡張される依存関係先は，直接の呼び出し（深さ 1）のみに限定している．

度寄与するかを評価する．

6.1 実 験 手 順

本実験では，バグ修正コミットにおいて変更された行を含む関数，すなわち DEP-SZZ の深さ 0 の探索範囲に含まれる関数を対象とし，以下の 2 段階のプロセスで評価を行う．

重要度に基づく分類: 対象となる全ての変更関数に対し，LLM を用いてコア修正関数，周辺修正関数，無関係関数の 3 つのカテゴリへの分類を行う．本手順では，各カテゴリに分類された関数数および解析行数と，各カテゴリの関数を追跡対象とした場合の TP を計測する．これにより，コア修正関数への分類が，TP を維持しつつ探索コストをどの程度削減できるかを評価する．

ランク付けと探索範囲の拡張: コア修正関数に対し，LLM を用いてバグの根本原因である可能性が高い順にランク付けを行う．その後，各バグ修正コミットについて，ランク上位 k 件（$k = 1, 2, 3, 5, 10$）の関数を選択した場合の検出精度とコストを測定する．コア修正関数のみを探索した場合とコア修正関数を起点として依存関係先へ探索範囲を拡張した場合の双方について，TP，解析行数，関数数を計測する．これにより，ランク付けを利用した優先的な探索が，SZZ の検出能力と効率性にどのような変化をもたらすかを評価する．

6.2 結　　果

コア修正関数への絞り込みは，バグ検出能力を維持したまま探索コストを大幅に削減した．表 1 に，深さ 0 の DEP-SZZ，それを関数内部に限定した変更行を含む関数，および LLM によって分類された各カテゴリの関数数，解析行数，TP 数を示す．コア修正関数は，変更行を含む関数のみを追跡して得られる 56 件の TP のうち 54 件（約 96.4%）を検出し，解析行数も 12,910 行から 6,976 行へと約 46% 削減された．これは，LLM が本質的な修正箇所を適切に特定しており，それ以外の周辺修正や無関係な修正を探索対象から除外しても，バグ混入コミットの検出能力はほとんど損なわれないことを示している．

コア関数のランク付けにより，解析行数を削減しつつ TP を維持可能である．表 2 のコア修正関数列に示すように，Top 2 の時点で TP 数は 54 件に達し，全てのコア修正関数を調査した場合と同値となった．この際，解析対象となる関数数は全 197 関数から 92 関数へと約 53% 削減され，解析行数についても 7,442 行から 5,181 行へと約 30% 削減された．LLM によるランク付けは，探索コストの削減とバグ検出の網羅性を両立する有効な手法である．

ランク上位の関数を起点とした探索範囲の拡張は，効率的に TP を増加させる．表 2 の「コア修正関数 + 両方」列における Top 1 は，全てのコア修正関数（All Core）と比較して，TP 数は 60 件から 57 件へと 3 件の減少に留まり，解析行数は約 17% 削減された．ただし，コア修正関数単体では解析行数を約 30% 以上削減可能であったことと比較すると，依存関係先まで含めた場合の削減効果は限定的であった．なお，呼び出し先への拡張は，呼び出し元に比べて関数数は多いものの，解析行数は少なく，より多くの TP を効率的に検出している．

RQ2 への回答

LLM による重要度評価は，SZZ の探索範囲からノイズを大幅に除去し，効率的なバグ特定を可能にする．コア修正関数への絞り込みにより，96.4% の TP を維持しつつ解析行数を約 46% 削減した．さらに，ランク上位 2 件のコア修正関数を調査するだけで，コア修正関数内の全 TP を網羅しつつ解析行数を約 30% 削減できた．また，ランク上位の関数を起点とした探索範囲の拡張もまた，効率的な TP の増加に寄与する．

7. 議　　論

7.1 成果と課題

RQ1 と RQ2 の結果は，バグ修正コミットにおける修正内容の重要度を評価し，本質的なバグ修正箇所であるコア修正関数を特定することが，SZZ 手法の精度向上と探索範囲の適正化に寄与することを示した．特に，重要度が低い周辺的な修正や無関係な変更を探索対象から除外することでノイズの混入を防ぎつつ高い再現率を維持可能であることが明らかになった．

また，探索範囲の拡張においても，重要な修正として上位にランク付けされたコア修正関数を起点とすることで，効率的に TP を捕捉できる可能性が示された．All-SZZ や深さ 1 の DEP-SZZ といった手法では膨大な解析コストを要していた依存関係先の探索が，起点を絞り込むことで大幅に削減できた．

しかし，本研究の RQ2 までの評価は提案手法の一部に限ら

表 3: Top 2 コア修正関数の呼び出し先選別の影響

手法	関数数	解析行数	TP
コア修正関数 + 呼び出し先	14,458	100,945	57
コア修正関数 + 選択された呼び出し先	**164**	**6,124**	**57**
削減率	-98.9%	-93.9%	±0

れる．表 2 で示した通り，Top 5 において依存関係先を含めた解析行数は依然として約 21.7 万行に達しており，これを全て追跡することはノイズの観点から課題が残る．したがって，実用的な SZZ を実現するためには，拡張された探索範囲の中から真に影響を受けた依存関係関数の選別を行い，さらにその中から，追跡することでバグ混入コミットを特定できる行を特定するフェーズ 3 以降の処理が不可欠である．

7.2 フェーズ 3 の予備調査

上述の課題に対し，本節では依存関係関数の選別に関する予備調査の結果を報告する．RQ2 の結果において，呼び出し元への拡張と比較して，呼び出し先への拡張は解析行数を低く抑えつつ効率的に TP を検出できる傾向が見られた．具体的には，ランク上位 2 件のコア修正関数に対し，双方へ拡張した場合の TP は 58 件であるのに対し，呼び出し先のみへ拡張した場合でも TP は 57 件であり，その差はわずか 1 件であった．この知見に基づき，本予備調査ではランク上位 2 件のコア修正関数とその呼び出し先を対象として，LLM によるフィルタリングを実施し，その有効性を評価した．

7.2.1 実 験 手 順

依存関係関数の選別は，以下の 2 段階のステップで実施した．

（1） **リストに基づく選別:** ランク上位 2 件のコア修正関数のコード，修正コミットにおける変更差分，および呼び出し先関数のリストを LLM に入力し，修正内容によって影響を受ける可能性がある関数を選択させた．これにより，明らかに無関係な関数を初期段階で除外した．

（2） **コードに基づく詳細選別:** ステップ 1 で選択された関数について，その関数自体のソースコードを追加情報として LLM に入力し，実際に修正の影響が波及しているかを再度判定させた．

この手順により，最終的に残った関数数，解析行数，および維持された TP 数を計測した．

7.2.2 結　　果

予備調査の結果，表 3 に示すように，TP 数 57 件を維持したまま，探索対象となる関数数を 14,458 件から 164 件へと，約 98.9% 減少させた．また，解析行数についても 100,945 行から 6,124 行へと，約 93.9% 削減された．この結果は，LLM を用いることで，静的解析だけでは抽出過多となる依存関係先の中から，意味的に影響関係にある関数のみを高い精度で選別できることを示している．

ただし，選別後も依然として 164 件の関数が探索対象として残存している．SZZ の最終的な目的であるバグ混入コミットの特定を正確に行うためには，関数レベルでの絞り込みに加え，関数内のどの行を追跡すべきかを特定する追跡行レベルでの詳細な解析が必要であることが示唆される．

8. おわりに

本研究では，従来の SZZ 手法におけるバグ混入コミットの見逃しと，探索範囲拡大に伴うノイズの増大というトレードオフを解消するため，関数の依存関係に基づく探索範囲の拡張と LLM による重要度評価を統合した手法 DEP-SZZ を提案した．68 の Java OSS プロジェクトを対象とした初期調査の結果，関数の依存関係を利用した探索範囲の拡張は従来手法が見逃していたバグ混入コミットの捕捉に有効であり，LLM によるバグ修正コミットにおける本質的な変更箇所の特定は，検出精度を維持しつつ探索効率の改善に寄与することが示された．

今後の展望として，探索コストのさらなる削減と実用化に向けた検証が挙げられる．今回の実験では，探索範囲の拡張に伴い解析対象となる行数が依然として多く残るという課題が確認された．これに対し，予備調査では LLM を用いて依存関係先をさらに選別することでコストを大幅に削減できる見通しを得ている．したがって今後は，フェーズ 3 にあたる依存関係関数の選別および追跡行の特定を実装し，より詳細な粒度での解析を行うことで，高精度かつ効率的な SZZ 手法の確立を目指す．

謝辞　本研究の一部は, JSPS 科研費 JP23K24823, JP24H00692, JP24K02921, JP25K03100, JP25K03102, JP25K22845 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] 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. Softw. Eng., vol.39, no.6, pp.757–773, 2013.

[2] E. Sahal and A. Tosun, “Identifying Bug-Inducing Changes for Code Additions,” Proc. 12th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas., pp.1–2, 2018.

[3] C. Rezk, Y. Kamei, and S. McIntosh, “The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects,” IEEE Trans. Softw. Eng., vol.48, no.9, pp.3297–3309, 2022.

[4] Y. Lyu, H.J. Kang, R. Widyasari, J. Lawall, and D. Lo, “Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel,” IEEE Trans. Softw. Eng., vol.50, no.9, pp.2219–2239, 2024.

[5] J. Sliwerski, T. Zimmermann, and A. Zeller, “When Do Changes Induce Fixes,” ACM SIGSOFT Softw. Eng. Notes, vol.30, no.4, pp.1–5, 2005.

[6] S. Kim, T. Zimmermann, K. Pan, and E.J. Jr. Whitehead, “Automatic Identification of Bug-Introducing Changes,” Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng., pp.81–90, 2006.

[7] D.A. daCosta, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A.E. Hassan, “A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes,” IEEE Trans. Softw. Eng., vol.43, no.7, pp.641–657, 2017.

[8] 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. Syst. Softw., vol.202, p.111729, 2023.