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
SS2026-21,KBSE2026-21(2026-07)

バグ混入コミットによる過去制約違反の実証研究

近藤　偉成† 　近藤　将成†† 　ヌリオリビエ† 　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,nourry,higo}@ist.osaka-u.ac.jp, ††{kondo,kamei}@ait.kyushu-u.ac.jp

あらまし　バグを効率的に修正・予防するには，バグがどの変更によって，どのような理由で混入されたのかの理解が重要である．バグを混入した変更であるバグ混入コミット（BIC）は，SZZ 手法などによって同定され，実証研究の基盤として用いられてきた．しかし，BIC がバグを混入した原因は，変更内容そのものではなく，BIC 以前に確立された制約の違反にある場合があり，変更内容に着目する既存の枠組みでは説明しきれない．本研究では，このような事例を過去制約違反と呼び，68 件のオープンソースプロジェクトに含まれる 76 件の BFC–BIC ペアを対象に，LLM エージェントを用いて詳細に分析した．その結果，分析対象の 59.2% が過去制約違反であり，破られた制約の多くは，値や型，実行文脈に関する意味的な前提であった．さらにその 88.9% は，BIC 時点では機械的な検査の対象になっておらず，近傍コードや実行文脈を読み解いてはじめて把握できる制約であった．これらの結果は，変更が依存する過去の制約を変更時点で可視化する支援の，バグ予防への有効性を示唆する．

キーワード　バグ混入コミット，過去制約違反，LLM エージェント，ソフトウェアリポジトリマイニング

1. はじめに

ソフトウェアのバグは，既存機能の失敗や予期しない挙動を引き起こし，利用者や開発者に大きな影響を与える [1]．また，バグの修正には，失敗の再現，原因箇所の特定，関連する変更履歴の調査など，多くの時間と労力が必要となる [2]．そのため，バグを効率的に修正・予防するには，バグが発生した箇所だけでなく，そのバグがどの変更によって，どのような理由で混入されたのかの理解が重要である [3]．

バグを混入したと考えられる変更は，バグ混入コミット（bug-introducing commit; BIC）と呼ばれる．BIC は，バグがいつ，どのような変更によって導入されたのかを理解するための重要な手がかりである [4]．代表的な BIC 同定手法として，バグ修正コミット（bug-fixing commit; BFC）で修正された行の履歴を遡る SZZ 手法およびその派生手法が提案されてきた [5], [6]．同定された BIC は，バグ原因分析や Just-In-Time 欠陥予測など，バグに関する実証研究の基盤として利用されている [7]．

BIC がどのような原因でバグを混入したのかを理解するには，BIC 内の変更内容や BFC との対応関係だけでは不十分な場合がある．ある BIC は，新たに追加されたコード内で完結する局所的な誤りとして説明できる．別の BIC は，BIC 以前から存在していた前提，不変条件，API 利用契約，実行順序，あるいは近傍コードとの一貫性といった性質を破ることでバグを混入している．実際，リファクタリングやバグ修正が既存の前提を損なって新たなバグを生む事例や [8], [9]，BIC の変更自体は正しいが他の箇所がそれに適応できずにバグが生じる事例 [10] が報告されている．このような BIC の誤りは diff 単体の中に閉じているのではなく，過去に確立された性質との関係の中ではじめて説明できる．しかし，SZZ 手法やその派生手法は BFC の変更行を遡って BIC を同定し，既存の BIC 分析の多くも BIC の diff と BFC との対応に着目してきた [11], [12]．そのため，バグの原因が diff の内側ではなく文脈の側にある場合，変更内容そのものを見る枠組みでは，なぜその変更がバグを混入したのかを説明しきれない．

そこで本研究では，BIC 以前から存在していた制約を BIC が破った事例を，**過去制約違反**と呼ぶ．ここで制約とは，ソフトウェアが正しく動作するために満たされるべき性質のうち，次の 2 つの条件を満たす性質を指す．第一に，その制約は BIC よりも前の時点で既に確立されていた．第二に，その制約の違反は BIC の diff 単体の中には現れず，近傍コードや実行文脈といった BIC 外の情報との関係においてのみ説明できる．この 2 条件により，過去制約違反は，追加されたコード内で完結する局所的な誤りとは明確に区別される．本研究は，BIC を単に「バグを混入した変更」として扱うのではなく，BIC が破った制約に着目し，「過去に確立されたどの制約を，どのように破った変更なのか」という観点から分析する．

本研究では，過去制約違反の実態を明らかにするため，以下の 3 つの研究設問（RQ）を設定した．

RQ1: 過去に確立された制約を違反する BIC はどの程度存在し，どのような制約が違反されているか？

RQ2: 違反された制約は，履歴上のどこで・いつ・誰によって確立されたか？

RQ3: 違反された制約は，バグ混入時点でどのように明示されていたか？

これらの RQ に答えるため，68 件のオープンソースソフト

図 1　BigStitcher における過去の制約を違反したバグ混入コミット (acef72c) の例

ウェア（OSS）プロジェクトに含まれる 76 件の BFC–BIC ペア [13] を対象に，詳細な手作業による分析を実施した．

分析の結果，76 件の BIC のうち 45 件（59.2%）が過去制約違反として判定された．これらの違反は，データ制約，実行文脈上の制約，コード規約の 3 種類に分類された．また，その多くは BIC 時点で明示的な検査やドキュメントとして存在しておらず，近傍コードや大域的な実行文脈から読み取る必要があった．これらの結果は，バグの原因の相当数が，変更されたコードの内側ではなく，それ以前から確立された制約との関係に存在すると示している．この知見は，バグの修正において破られた制約とその確立箇所を併せた提示や，バグ予防のために変更が依存する暗黙の制約を検査対象とする手法の有効性を示唆する．

2. 動機付けの例

過去制約違反の具体例を図 1 に示す．この例では，BIC の誤りが diff 単体ではなく，過去に確立された制約との関係においてはじめて説明できる．本事例は，BigStitcher の GUI において，プレビューモードの終了時に global optimization を実行する処理を扱う．この処理では，プレビュー中に作成された設定オブジェクトを後続の global optimization へ受け渡す必要がある．そのため，呼び出し元である StitchingExplorerPanel が設定オブジェクトを savedFilteringAndGrouping に一時保存し，最適化側の OptimizeGloballyPopupExpertBatch がそれを読み取る構造になっていた．

BIC 以前は，global optimization が同期的に実行されていた．globalOptPopup.doClick() が返った時点で，最適化側は savedFilteringAndGrouping を既に読み終えている．呼び出し元は，doClick() の直後に savedFilteringAndGrouping を null とし，設定オブジェクトを破棄していた．すなわち，「設定オブジェクトを最適化処理が消費してから破棄する」という順序が保たれていた．

BIC では，global optimization が別スレッドで実行されるように変更された．doClick() は最適化処理の完了を待たず，ワーカースレッドを起動した直後に戻る．その結果，呼び出し元が従来どおり savedFilteringAndGrouping を null にすると，ワーカースレッドが設定オブジェクトを読み取る前に値が破棄される可能性が生じる．BIC は，「設定オブジェクトを最適化処理が消費してから破棄する」という既存の順序制約を破っていた．

BFC は，この破棄のタイミングを修正している．global optimization を起動する場合は呼び出し元では破棄せず，処理の完了後に，非同期に実行される最適化側が setSavedFilteringAndGrouping(null) を呼び出す．これにより，設定オブジェクトの破棄は，非同期処理の開始直後ではなく，その値の利用が完了した後に行われるようになった．

この例で重要なのは，BIC が行った非同期化そのものは，UI の応答性を改善する自然な変更だという点である．問題は，その変更が，同期実行を前提として成り立っていた順序制約を破った点にある．この制約は BIC の diff 内には現れておらず，BIC 以前のコードの実行文脈を読み解いてはじめて把握できる．本事例は，BIC 内で完結する局所的な実装誤りではなく，過去に確立された制約を後続の変更が破った過去制約違反として解釈できる．

3. 実験設計

3.1 データセット

本研究では，Rosa らが構築した開発者情報付きオラクルデータセットに含まれる BFC–BIC ペアを使用した [13]．このデータセットは，SZZ 手法による機械的な推定ではなく，バグ修正コミットのメッセージ中で開発者がバグ混入コミットを明示した事例を抽出し，手動フィルタリングで曖昧な事例を除外して構築されている．そのため，BFC と BIC の対応関係について高い信頼性を持つ．

データセットには 2,304 件の BFC–BIC ペアが含まれる．本研究では，このうち Java プロジェクトに属し，かつリポジトリが利用可能な 76 件を分析対象とした [6]．Java は SZZ 手法や Just-In-Time 欠陥予測などの研究で広く対象とされており [5], [7]，既存研究との比較や知見の位置付けが行いやすい．また，本研究は各事例について BFC，BIC，およびその周辺履歴を詳細に確認する必要があるため，対象を Java に限定し，手作業による分析が可能な範囲に絞った．

3.2 データアノテーション

各 BFC–BIC ペアに対し，表 1 に示すように，過去制約違反，局所的誤り，判定困難，BIC 誤ラベルのいずれかのラベルを付与する．そして，過去制約違反に該当する事例を特定し，判定

の過程で参照した情報をもとにこれらを詳細に分析する．

アノテーションは，著者が LLM エージェントの支援を受けて行った．まず，各事例について BFC，BIC，および git 履歴の調査を Claude Opus 4.8 に補助させ，コーディングルールに沿った一次的なラベル付けを得た．このとき Claude には，実行したコマンドとその出力を記録するツールを与え，参照した出力を著者が同一の形で確認できるようにした．そのうえで著者が，これらのログと収集された証拠を一件ずつ確認し，コーディングルールに従って証拠が収集されているか，および BIC 以前の証拠に基づくかを検証した．証拠が妥当でない事例では，著者自身が git 履歴を再調査し，最終的なラベルは著者が確定した．

各事例の分析は，以下の探索手順に従って行った．

（1）**BFC の調査**：BFC の diff とコミットメッセージから，バグを防ぐために満たされるべき性質 X を特定する．

（2）**BIC の調査**：BIC の diff とコミットメッセージから，BIC が加えた変更を把握する．

（3）**対応付け**：BFC が変更した行と BIC が変更した行を対応付ける．

（4）**事前存在証拠の探索**：X が BIC 以前から存在していたことを示す独立証拠を探索する．

（5）**制約導入元の同定**：`git log -S` や `git blame` を用いて X を確立した制約導入コミットの候補を特定し，それが BIC の祖先であることを確認する．

（6）**反証の試行**：過去制約違反という判断を覆す可能性を能動的に検討する．すなわち，X が事前に存在しなかった，欠陥が BIC 以前から存在していた，あるいは誤りが BIC 内で完結している，のいずれかではないかを検証する．

（7）**関連 issue の確認**：コミットが issue や PR を参照している場合，それらを確認する．

過去制約違反の判定では，BFC の内容を言い換えただけの説明を排除することを重視した．そのため手順 4 の独立証拠には，X を確立した祖先コミット，BIC 以前のテストや assert，コメントやドキュメント，外部 issue や仕様，近傍コードパターンを用い，BFC の説明の言い換えや BIC 以後に追加された証拠は採用しない．さらに手順 6 の反証により，採用した証拠が偶然の一致や事後的な再構成でないことを確認した．

各ペアのラベルは，これらの手順における判定結果に基づいて決定する．手順 4 で X の事前存在を示す独立証拠が得られ，かつ手順 6 の反証がいずれも成立しない場合，その事例を過去制約違反とする．手順 6 において，修正対象の欠陥が BIC 以前から存在していたと判明した場合は BIC 誤ラベル，誤りが BIC 内で完結すると判明した場合は局所的誤りとする．また，X は記述できるものの，その事前存在を示す独立証拠が得られない場合は判定困難とする．

表 1　各事例の分類ラベル

ラベル	定義
過去制約違反	BIC 以前から存在していた制約を BIC が破った事例
局所的誤り	BIC 内で完結する実装誤りとして説明できる事例
判定困難	性質 X は説明できるが，その事前存在を示す証拠が不十分な事例
BIC 誤ラベル	修正対象の欠陥が BIC 以前から存在していた事例

3.3　研 究 設 問

本研究では，前節で付与したラベルに基づき，以下の 3 つの研究設問に答える．

RQ1：過去に確立された制約を違反する BIC はどの程度存在し，どのような制約が違反されているか？

分析対象の 76 件のうち，過去制約違反と判定された事例を数え，その割合を算出する．さらに，これらの事例について，手順 1 で特定した性質 X と手順 4 で得た証拠をもとに，どのような制約が違反されているかを分析する．この分析を通じて制約の種別を作成し，各事例をその種別に分類して分布を集計する．

RQ2：違反された制約は，履歴上のどこで・いつ・誰によって確立されたか？

過去制約違反と判定された各事例について，手順 5 で特定した制約導入コミット，すなわち破られた制約を確立したコミットを分析する．具体的には，制約導入コミットと BIC の変更箇所の関係，制約導入コミットと BIC の時間差，および制約を確立した開発者自身がそれを破る自己制約違反であるかを調べる．

RQ3：違反された制約は，バグ混入時点でどのように明示されていたか？

過去制約違反と判定された各事例について，破られた制約が BIC 時点でどのように明示されていたかを分析する．具体的には，手順 4 で得た証拠や，コミットメッセージ・issue の記述を手作業で調べ，制約の明示形態を分類する．そのうえで，明示形態と RQ1 で得た制約の種別との関係を分析する．

4.　RQ1：過去に確立された制約を違反する BIC はどの程度存在し，どのような制約が違反されているか？

4.1　過去制約違反の割合

分析対象とした 76 件の BFC–BIC ペアのうち，45 件（59.2%）が過去制約違反であった．判定ラベルの内訳を表 2 に示す．過去制約違反が 45 件，局所的誤りが 22 件，BIC 誤ラベルが 8 件，判定困難が 1 件であった．過去制約違反が全体の約 6 割を占めることは，バグの原因の半数以上が，BIC の変更内に閉

表 2　判定ラベルの分布

ラベル	件数	割合
過去制約違反	45	59.2%
局所的誤り	22	28.9%
BIC 誤ラベル	8	10.5%
判定困難	1	1.3%
合計	76	100.0%

表 3　違反された制約の種類と分布

種類	定義	件数	割合
データ制約	値の範囲，nullability，識別子，状態など，データが満たすべき性質	22	48.9%
実行文脈上の制約	呼び出し順序，実行モード，並行実行，ライフサイクルなど，実行文脈に依存する制約	17	37.8%
コード規約	API 利用，リソース管理，静的解析規則，既存コードとの一貫性などに関する制約	6	13.3%
合計		45	100.0%

表 4　制約種類別の確立位置・時期・著者の傾向

		確立位置					
種類	件数	同一関数	同一ファイル	同一モジュール	別モジュール	期間中央値	自己制約違反
データ制約	22	16 (72.7%)	4 (18.2%)	0 (0.0%)	2 (9.1%)	227 日	13 (59.1%)
実行文脈上の制約	17	14 (82.4%)	0 (0.0%)	2 (11.8%)	1 (5.9%)	282 日	5 (29.4%)
コード規約	6	4 (66.7%)	0 (0.0%)	1 (16.7%)	1 (16.7%)	540 日	1 (16.7%)
全体	45	34 (75.6%)	4 (8.9%)	3 (6.7%)	4 (8.9%)	265 日	19 (42.2%)

じた誤りではなく，BIC 以前から存在していた制約との関係に存在することを示している．

BIC 誤ラベルとした 8 件では，欠陥が BIC 以前から存在していたにもかかわらず，開発者が BIC を混入コミットとして言及していた．たとえば，潜在的な欠陥が BIC 以前から存在し，BIC がその欠陥を通る実行経路を新たに導入したことで顕在化した事例があった．これは，「バグを混入したコミット」という概念自体が，欠陥の作り込みと顕在化のいずれを指すのかという点で曖昧さを含むことを示している．

4.2　違反された制約の種類

違反された制約のうち，値や型に関するデータ制約と，呼び出し順序などの実行文脈上の制約が全体の約 87% を占めた．45 件の過去制約違反について，違反された制約を表 3 に示す 3 種類に分類した．その結果，データ制約が 22 件（48.9%）と最も多く，次いで実行文脈上の制約が 17 件（37.8%），コード規約が 6 件（13.3%）であった．大半を占めたデータ制約と実行文脈上の制約は，いずれもプログラムの動作の正当性を支える意味的な不変条件であり，BIC が破る制約が形式的な記述規約よりも意味的な不変条件に偏在することを示唆している．

RQ1 への回答

分析対象 76 件の BFC–BIC ペアのうち，45 件（59.2%）が過去制約違反であり，変更内に閉じた誤りではなく，BIC 以前に確立された制約を破ることでバグを混入していた．違反された制約はデータ制約，実行文脈上の制約，コード規約の 3 種類に分類でき，値や型に関するデータ制約と呼び出し順序などの実行文脈上の制約が全体の約 87% を占めた．

5.　RQ2：違反された制約は，履歴上のどこで・いつ・誰によって確立されたか？

過去制約違反と判定された 45 件すべてについて，破られた制約を確立した制約導入コミットを git 履歴上で特定した．BIC と制約導入コミットの位置関係，制約が確立された時期，および制約を確立した開発者の 3 つの観点から分析する．これら 3 つの観点について，全体および制約の種類ごとに集計した結果を表 4 に示す．

5.1　制約が確立された場所

過去制約違反の 75.6% は，BIC と制約導入コミットが同一の関数を変更している．45 件のうち 34 件（75.6%）では両コミットが同一の関数を変更しており，同一ファイル内の事例を含めると 38 件（84.4%）に達する．残る 7 件（15.6%）では，制約が別のファイルや別のモジュールで確立されており，BIC とは離れた箇所で確立された制約が破られていた．この傾向は制約の種類によらず共通しており，いずれの種類でも同一関数内での確立が中心であった．

5.2　制約が確立された時期

制約が確立されてから BIC で破られるまでの期間は，数日から数年まで幅広く分布する．制約導入コミットから BIC までの期間は，0 日から 3,317 日にわたって分布し，その中央値は 265 日である．確立から 30 日以内に破られた事例が 10 件ある一方で，1 年以上を経て破られた事例も 21 件（うち 3 年以上が 9 件）あった．すなわち過去制約違反は，導入されて間もない制約が破られる場合から，長期間維持されてきた制約が後年の変更によって破られる場合まで，幅広い時間スケールで生じている．種類別では，データ制約と実行文脈上の制約の期間中央値が比較的短い（227 日，282 日）のに対し，コード規約は 540 日と長い．

5.3　制約を確立した開発者

制約を確立した開発者自身がその制約を破る事例も見られた．制約を確立した開発者と BIC の著者が一致した事例は 45 件中 19 件（42%）であった．長く当該コードを保守してきた開発者であっても，自らが確立した制約を破りうる．ただし，この 19 件のうち 15 件は制約導入コミットの著者が当該ファイルの唯一の編集者であった事例であり，自己違反の多くはファ

イルの編集者が限られる状況で生じている．編集者からランダムに違反者が選ばれると仮定した場合の期待値は約 69% であり，観測された 42% はこれを下回る．したがって，この割合から制約の知識と違反のしやすさの関係を読み取ることはできず，著者性は本データにおいて過去制約違反を特徴づける明確な傾向を示さなかった．

RQ2 への回答

過去制約違反の 75.6% では，制約導入コミットと BIC が同一の関数を変更しており，破られた制約は BIC の近傍に確立される傾向が強かった．制約が確立されてから BIC で破られるまでの期間は数日から数年まで幅広く分布した（中央値 265 日）．制約を確立した開発者自身による違反も 42% に見られたが，ファイルの編集者の集中から期待される水準を下回り，著者性に特筆すべき傾向は認められなかった．

6. RQ3：違反された制約は，バグ混入時点でどのように明示されていたか？

過去制約違反と判定された 45 件について，破られた制約が BIC 時点でどのように明示されていたかを，表 5 に示す 6 つのカテゴリ（A〜F）に分類した．分類結果を全体および制約の種類ごとに表 6 に示す．

6.1 全体の明示形態の傾向

過去制約違反の 88.9% は，BIC 時点では静的解析やテストによる機械的な検査の対象になっておらず，コードや実行文脈を読み解いてはじめて把握できる制約であった．静的解析やコンパイラ，あるいは assert や実行時検査によって機械的に検出できる形で存在していた制約（A・B）は，わずか 5 件（11.1%）にとどまった．残る 40 件（88.9%）は，BIC 時点ではこうした機械的な検査の対象になっていない制約である．とりわけ，近傍のコードパターンから推論する必要がある制約（D）が 25 件（55.6%）と最も多く，全域的な解析や外部仕様の理解を要する制約（E・F）も 9 件（20.0%）あった．過去制約違反の多くは，変更時点で機械的に検出可能な形では存在しておらず，周辺のコードや実行文脈の読解を通じてはじめて認識できる．

6.2 種類ごとの明示形態の傾向

機械的に検出できる制約はコード規約に偏り，データ制約と実行文脈上の制約はコードの読解を要するものが大半であった．機械的に検出できる 5 件（A・B）のうち 3 件はコード規約であり，コード規約は 6 件中 3 件がツールで検出可能であった．一方，データ制約は 22 件中 21 件が，実行文脈上の制約は 17 件すべてが，機械的な検出には該当しない制約であった．とりわけ実行文脈上の制約は，17 件中 7 件が全域的な解析を要するカテゴリ E に分類され，呼び出し順序や実行モードといった非局所的な前提に依存するという性質を反映している．これは，現行のツールが機械的に強制する形式的な規約ほどバグとして残りにくく，逆にツールが検査しない意味的な制約（データの不変条件や実行文脈上の前提）が過去制約違反として顕在化しやすいことを示唆している．

表 5　制約の明示形態の分類

カテゴリ	明示形態	定義
A	静的検出可能	静的解析ツールやコンパイラで検出可能
B	実行時検査	assert，実行時検査，警告として明示
C	文書化	コメントやドキュメントに記述
D	近傍コード	近傍コードパターンから推論可能
E	全域的文脈	全域的なコード解析により把握可能
F	暗黙	外部仕様の理解を要する，または非明示

RQ3 への回答

過去制約違反の 88.9% は，BIC 時点では静的解析やテストによる機械的な検査の対象になっておらず，近傍コードや実行文脈を読み解いて初めて把握できる制約であった．種類別では，機械的に検出可能な形で存在していた制約がコード規約に偏る一方，データ制約と実行文脈上の制約はその大半がコードの読解を要した．これは，ツールが機械的に検査する形式的な規約よりも，そうした検査の対象とならない意味的な制約が過去制約違反として残りやすいことを示唆している．

7. 考　　察

過去制約違反で破られた制約の多くは，BIC と近接した履歴上に存在している．過去制約違反の 75.6% では，破られた制約を確立した制約導入コミットが BIC と同一の関数を変更しており，同一ファイル内を含めると 84.4% に達した．また，すべての事例で制約導入コミットを git 履歴上で特定できた．この結果は，過去制約違反が必ずしも大域的な依存関係だけに由来するのではなく，多くの場合，変更対象と同じ関数やファイルの履歴に埋め込まれた制約の見落としとして生じる．したがって，リポジトリ全体を対象とした重い解析だけでなく，変更箇所近傍の履歴を優先的に提示する軽量な支援でも，一定数の違反を防げる可能性がある．

破られた制約の多くは明示的に記述されておらず，変更時に開発者が自力で気づくのは難しい．過去制約違反の 88.9% は静的解析やテストでは機械的に検出できず，さらに 42% は制約を確立した開発者自身による違反であった．制約に関する知識を持つ開発者でさえ見落とすほど，制約は変更時点で明

表 6　制約種類別の明示形態の分布

種類	件数	A	B	C	D	E	F
データ制約	22	0	1	5	14	1	1
実行文脈上の制約	17	1	0	0	9	7	0
コード規約	6	2	1	1	2	0	0
全体	45	3	2	6	25	8	1

示されておらず，開発者の注意や記憶のみに委ねられている．この明示性の欠如が，過去制約違反が繰り返し生じる一因である．

変更が依存する過去の制約を変更時点で可視化することは，バグの予防に重要である．破られた制約は，履歴をたどれば到達できる位置に存在しながらも，変更時点では明示されていない．したがって，変更しようとする箇所に関連する制約導入コミットや近傍コードから暗黙の前提を抽出し，変更がそれらを破る可能性を警告できれば，過去制約違反の混入を未然に防げる．このような支援は，diff の局所に閉じた従来のレビューや静的解析では捉えにくいバグの予防に寄与する．

8. 妥当性への脅威

本研究の構成概念妥当性，内的妥当性，外的妥当性に関する脅威とその緩和策を以下に述べる．

構成概念妥当性 本研究では，制約の事前存在性とその違反という操作的定義に基づいて過去制約違反を判定している．この定義が制約という概念を十分に捉えられていなければ，判定結果が本来の意図とずれる恐れがある．この脅威を緩和するため，性質 X の事前存在を独立証拠によって裏づけ，BFC の内容を言い換えただけの説明を排除する手順を設けた．

内的妥当性 アノテーションに LLM を用いているため，判定結果が LLM の出力に左右される恐れがある．これに対しては，全事例について LLM の出力，実行コマンド，および収集された証拠を著者が確認し，証拠が不十分な事例では著者自身が git 履歴を再調査したうえでラベルを確定した．

外的妥当性 本研究の分析対象は，特定のデータセットに含まれる Java プロジェクトの 76 件の BFC–BIC ペアである．著者による詳細な手作業を要するため，SZZ 手法のように大規模な収集は行っていない．得られた割合や傾向が過去制約違反の実態をそのまま代表するとは限らず，他の言語やプロジェクト，より大規模な事例での検証は今後の課題である．

9. おわりに

本研究では，バグの原因が必ずしもバグ混入コミット（BIC）の変更内容のみでは説明できないという課題に対し，BIC を，BIC 以前に確立された制約を破った変更として捉える過去制約違反の観点を導入した．そして，68 件のオープンソースプロジェクトに含まれる 76 件の BFC–BIC ペアを対象に，LLM エージェントと著者による詳細な分析を行った．その結果，分析対象の 59.2%（45 件）が過去制約違反であり，バグの原因の多くが BIC の変更内に閉じた誤りではなく，過去に確立された制約との関係に存在すると明らかになった．破られた制約の多くは，値や型，実行文脈に関する意味的な前提であり，その 88.9% は BIC 時点で明示的に記述されておらず，近傍コードや実行文脈を読み解いてはじめて把握できる．

これらの結果は，バグの原因となる制約が，履歴をたどれば到達できる位置に存在しながらも，変更時点では明示されていないと示している．したがって，変更が依存する過去の制約を変更時点で開発者に提示する支援は，diff の局所に閉じた従来の手法では捉えにくいバグの予防に寄与すると期待できる．ただし本研究の分析は特定のデータセットの Java プロジェクト 76 件に限られるため，他の言語やより大規模な事例における検証が必要である．今後は，これらの検証とともに，変更が依存する過去の制約を明示化する支援手法の実現に取り組む．

謝辞 本研究の一部は, JSPS 科研費 JP24K02921, JP24H00692, JP25K03100, JP25K03102,JP25K22845, JP26K02889, JP26H02500 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] M. Zhivich and R.K. Cunningham, “The Real Cost of Software Errors,” IEEE Secur. Priv., vol.7, no.2, pp.87–90, 2009.

[2] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, “How Long Does a Bug Survive? An Empirical Study,” Proc. 18th Work. Conf. Reverse Eng., pp.191–200, 2011.

[3] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D.M. Germán, and J.M. Gonzalez-Barahona, “How bugs are born: a model to identify how bugs are introduced in software components,” Empir. Softw. Eng., vol.25, no.2, pp.1294–1340, 2020.

[4] S. Kim, E.J. Whitehead, and Y. Zhang, “Classifying Software Changes: Clean or Buggy?,” IEEE Trans. Softw. Eng., vol.34, no.2, pp.181–196, 2008.

[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] H. Watanabe, M. Kondo, E. Choi, and O. Mizuno, “Benefits and Pitfalls of Token-Level SZZ: An Empirical Study on OSS Projects,” Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng., pp.776–786, 2024.

[7] 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.

[8] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, “When does a refactoring induce bugs? an empirical study,” Proc. IEEE Int. Work. Conf. Source Code Anal. Manip., pp.104–113, 2012.

[9] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, “How do fixes become bugs?,” Proc. ACM SIGSOFT Symp. Found. Softw. Eng., pp.26–36, 2011.

[10] X. Song, Y. Wu, B. Chen, Z. Lu, S. Liu, and X. Peng, “Characterizing Regression Bug-Inducing Changes and Improving LLM-Based Regression Bug Detection,” Proc. Int. Conf. Softw. Eng., 2026. to appear.

[11] C. Williams and J. Spacco, “SZZ Revisited: Verifying When Changes Induce Fixes,” Proc. Workshop Defects Large Softw. Syst., pp.32–36, 2008.

[12] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, “Exploring and exploiting the correlations between bug-inducing and bug-fixing commits,” Proc. 27th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp.326–337, 2019.

[13] 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.