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

ソースコードの編集差分を用いた高精度なSBFLの提案

川村　　颯†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{h-kawamr,shinsuke}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発におけるデバッグ支援技術の一つとして Spectrum-Based Fault Localization（SBFL）がある．SBFL では複数のテストの実行経路を用いて，欠陥の原因箇所を自動的に推定する．既存の SBFL 手法では一つ以上のテストが失敗する，つまり欠陥が混入した時点でのソースコードとテストを用いて欠陥箇所を推定する．しかし，実際のソフトウェア開発においては編集前のソースコードが常に存在している．この編集前の情報の活用によって SBFL による欠陥限局性能を改善できる可能性がある．本研究では SBFL の性能改善を目的として，ソースコードの編集差分を用いた新たな SBFL 手法を提案する．提案手法では，差分解析のツール GumTree を用いて編集前後のプログラムの抽象構文木（AST）を比較・解析し，構文レベルでの編集操作（追加，削除，移動，更新）を特定する．この AST 情報に基づき，既存 SBFL によって算出された各行の疑惑値に対して重み付けを行う．本稿では，提案手法を構成する 2 つの要素である編集状況の特定と差分を用いた SBFL 計算について紹介する．さらに提案手法の適用対象となる事例を対象として評価実験を行った結果，74.4% の事例で欠陥限局性能が向上し，平均検査コストを約 39.7% 削減できることを確認した．

キーワード　Spectrum-Based Fault Localization，SBFL，編集差分，GumTree

1. はじめに

ソフトウェア開発におけるデバッグ支援技術の一つとして，Spectrum-Based Fault Localization（SBFL）がある．SBFL は複数のテストケースの実行経路に基づき，ソースコード中の欠陥行の自動的な特定・限局を試みる．SBFL のアイデアは，失敗したテストケースが通過する行ほどバグ原因である可能性が高く，逆に成功したテストケースが通過する行ほど原因である可能性が低い，という発想に由来する．SBFL を用いることでソースコード中の全ての行に対して，バグ箇所の可能性の高さを表す疑惑値と呼ばれる値が算出される．開発者はこの疑惑値の高い順にソースコードを精査することで，効率的なバグ箇所の特定が可能となる．

SBFL のバグ限局性能の改善に関する研究が盛んに行われている [1]．条件確率等の統計的な視点を組み込んだ手法 [2] や，制御フロー等のソースコードの構造を考慮した手法 [3]，さらには，SBFL の入力となるテストケースを改善する方法 [4] も存在しており，様々な側面から SBFL の性能改善が続けられている．

しかしながら既存 SBFL は開発の文脈を十分に反映しておらず，さらなる性能改善の可能性がある．既存 SBFL の多くは，その入力データをある時点でのソースコードとテストケースに限定している．よって，バグが発生した開発の任意の時点において SBFL による疑惑値算出が可能であり，可搬性に優れた手法である．一方で人手によるバグ箇所の限局というシナリオを考えると，自身が最近編集した箇所を最優先で疑う，という発想は極めて自然である．バグがない状態からバグを含む状態への移行には，必ずソースコードの改変が伴うためである．近年ではテスト駆動開発の考えが広く浸透しており，常にテストを通過する状態を保つという開発スタイルも一般的となった [5]．よって SBFL に対し，どのような編集をどの部分に加えたか，という開発の文脈を組み込むことでバグ限局の性能改善が期待できる．

本研究では SBFL のバグ限局性能の改善を目的として，ソースコードの編集差分を考慮した新たな SBFL 手法，GumSBFL を提案する．本手法は，主に開発者のローカル環境において，リファクタリングやソースコード修正に伴って混入した回帰的な欠陥の特定を想定している．GumSBFL は，編集が加えられた行ほどバグ原因である可能性が高い，というアイデアに基づく．バグを含む編集後のソースコードだけでなく，バグを含まない編集前のソースコードの利用により，開発の文脈を SBFL に組み込む．また編集差分の算出においては，抽象構文木（Abstract Syntax Tree: AST）を用いた差分算出方法 GumTree [6] を用いる．GumTree の利用により，単純なテキスト差分では捉えきれない，より詳細な編集内容を活用する．

さらに提案手法の一部として，開発文脈の自動的な判定にも取り組む．これは編集差分が SBFL に有効な状況とそうでない状況があるためである．例えばリファクタリング時の機能破壊のような状況においては，編集差分は SBFL の改善に強く寄与する．一方，ソースコードとテストの両方を書き換える機能追加のような状況では，編集差分は逆に SBFL の性能低下に繋がる可能性が高い．このような状況を自動的に判定することで，実践的かつ高精度な SBFL を実現する．

This article is a technical report without peer review, and its polished and/or extended version may be published eleswhere.
Copyright ©2026 by IEICE

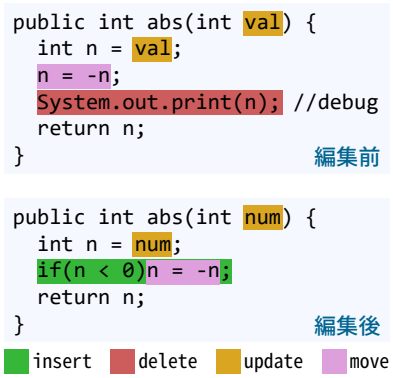

図 1: GumTree の実行例

2. 準　　備

2.1 Spectrum-Based Fault Localization（SBFL）

自動的な欠陥限局の手法の一つとして Spectrum-Based Fault Localization（SBFL）がある．SBFL はソースコードとテストを入力とし，テストケースの実行経路から欠陥箇所を自動で推定する．SBFL のキーアイデアは，失敗したテストで頻繁に実行され，かつ成功したテストではあまり実行されない箇所ほど欠陥を含む可能性が高いという点にある．

SBFL では，ソースコードの各行に疑惑値と呼ばれる数値を付与する．疑惑値とは欠陥を含む可能性を数値化した指標である．各行について，その行を実行した失敗テストの数と成功テストの数を集計し，計算式に適用することで疑惑値を算出する．計算式は Tarantula や Jaccard など多数存在する [7] が，本研究では先行研究で高い精度が報告されている Ochiai [8] を用いる．l 行目に対する疑惑値 $Susp(l)$ の，Ochiai の計算式は以下のように表される．

$$Susp(l) = \frac{e_f(l)}{\sqrt{(e_f(l)+n_f(l)) \times (e_f(l)+e_p(l))}}$$

ただし，

$e_f(l)$ ： l を実行した失敗テストケースの数
$e_p(l)$ ： l を実行した成功テストケースの数
$n_f(l)$ ： l を実行していない失敗テストケースの数
$n_p(l)$ ： l を実行していない成功テストケースの数

開発者は疑惑値の高い行からソースコードを検査することで，効率的に欠陥箇所を特定できる．

2.2 GumTree

ソースコードの構造に基づいて編集箇所を特定する手法として，Falleri らによって提案された GumTree [6] がある．GumTree の実行例を図 1 に示す．一般的な `diff` コマンドは行単位でテキストの差分を計算する．一方で，GumTree はソースコードを抽象構文木（Abstract Syntax Tree:AST）に変換し，ノード単位で差分を計算する．これにより，行単位では検出できない構造的な変更も検出できる．それに加えて，改行やコメント文などのプログラムの動作に関係がない変更に影響されずに差分計算が可能となる．

GumTree の出力は編集スクリプトである．編集スクリプトとは，編集前のソースコードの AST を編集後のソースコードの AST へと変換するための最小の操作列である．従来の `diff` コマンドは挿入と削除のみで差分を表現するが，編集スクリプトは以下の 4 種類の操作で構成されている．

insert : 新たなノードを追加する
delete : 既存のノードを削除する
update : ノードの値（変数名やリテラル値など）を変更する
move : 部分木を別の位置へ移動させる

これらの操作は，編集前後のノードの対応関係によって分類ができる．update と move は編集前と後の双方の AST に対応するノードが存在する対称的な操作である．一方で，delete と insert は，片方の AST にのみノードが存在する非対称的な操作である．編集差分の内容を正確に理解するには，図 1 のように編集前後両方のソースコードを確認する必要がある．

GumTree は move や update の検知精度に課題があることが報告されている [9]．例えば，本来は move や update として扱われるべき編集が，大量の delete と insert として扱われることがある．これは GumTree の処理が AST の構造のみに依存していることに起因する．ソースコードの編集により AST の構造的な特徴が大きく変化すると，GumTree はノードの対応関係を正確に追跡できない．

そこで本研究では，松本らによる先行研究 [10] のアプローチを採用する．これは，AST の情報だけでなく，行単位の差分情報を補助的に用いることで，GumTree の精度を向上させる手法である．具体的には，変更がない行を手がかりとして，編集前後のソースコードの対応関係をより高精度に特定する．これにより，従来の GumTree より短く正確な編集スクリプトの生成が可能となる．

3. 提 案 手 法

3.1 提案手法の概要

本研究では SBFL の精度向上を目的として，ソースコードの編集差分を用いた SBFL を提案する．本手法のキーアイデアは，欠陥は開発者の編集操作によって混入するという点にある．バグ修正や機能追加の際に開発者が編集した箇所は，欠陥が混入した可能性が高い．しかし，既存の SBFL は，ある時点でのソースコードとテスト結果だけに基づいて疑惑値を算出しており，この開発の文脈（開発コンテキスト）は全く使用していない．そこで本手法では，編集によって欠陥が混入したという仮定のもとで，開発コンテキストに基づいた疑惑値への重み付けを導入する．既存の SBFL ではできなかった欠陥箇所の絞り込みや，編集の意図を考慮した疑惑値の調整により，欠陥限局精度の向上を図る．

開発コンテキストの活用は，SBFL が本質的に抱えている同率問題 [11] に対しても有効な対策となる．同率問題とは，複数の行に対して同じ疑惑値が付与され，ランキングとしての区別性が低下する現象である．これは，ソースコードに条件分岐が少ない場合や，テストケース数が不足している場合に発生する．同じ疑惑値の行が多数存在すると，開発者はそ

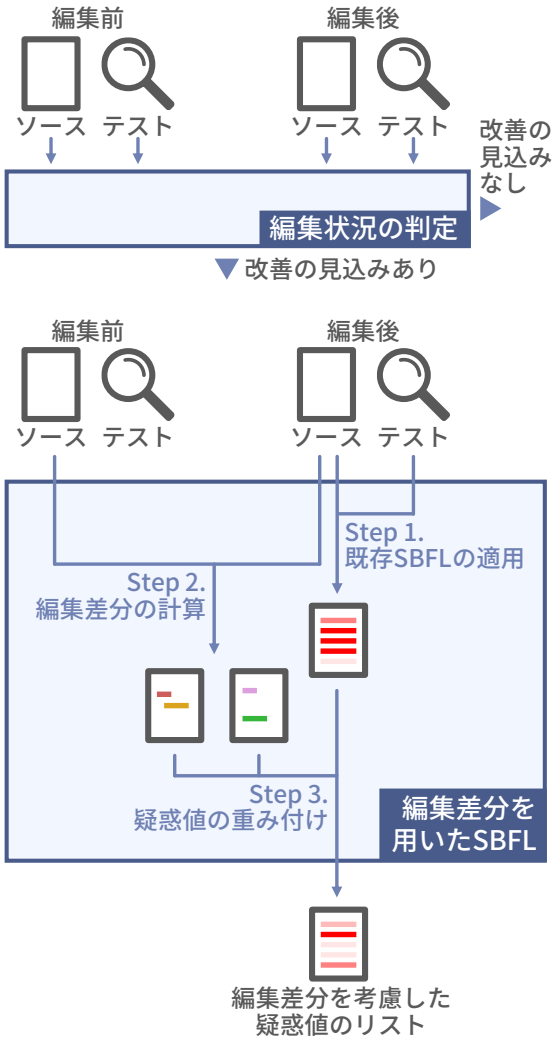

図 2: 提案手法の概要図

の中から真の欠陥行を探す必要があり，欠陥特定の効率が低下する．開発コンテキストを追加の手がかりとして利用することで，疑惑値に差異が生じ，同率問題の解消に寄与する．

さらに本手法では，本手法自体の適用可否の自動判定も導入する．なぜならば，編集内容に基づく重み付けは無条件に適用できないからである．本手法は，直近の編集が欠陥混入の原因であるという仮定が成立する場合において有効に機能する．欠陥が以前から潜伏していた場合や，テストの仕様変更が原因でテストに失敗するようになった場合にはこの仮定は成立しない．仮定が不成立の状況で本手法を適用すると，欠陥の混入と無関係な編集箇所を過剰に強調してしまい精度が悪化する恐れがある．したがって，本手法で精度向上を行うためには欠陥限局の対象が本手法に適しているかの判定が必要である．適用不可と判断された場合には本手法を適用せずに終了することで安定した欠陥限局精度を維持する．

3.2 提案手法の流れ

提案手法の概要図を図 2 に示す．本手法は「編集状況の判定」と「編集差分を用いた SBFL」の 2 つの要素から構成される．入力として，編集前ソースコードとテスト，編集後ソースコードとテストの 4 種類を受け取り，最終的に疑惑値のリストを出力する．なお，編集差分を用いた SBFL は以下の 3 つのステップからなる．

Step1. 既存 SBFL の適用
Step2. 編集差分の計算
Step3. 疑惑値の重み付け

以降では既存の SBFL に対して，本稿が提案する編集差分を用いた SBFL を GumSBFL と呼ぶ．

3.3 編集状況の判定

提案手法の最初のステップとして，欠陥限局対象への GumSBFL の適用が効果的であるかの判定を行う．本手法は，直前の編集が原因で新たに欠陥が混入した状況を仮定している．元から欠陥が潜伏していた場合や，仕様変更が原因の場合には仮定が成立しない．仮定が不成立の場合には，GumSBFL の適用で欠陥限局精度が悪化する可能性があるため，GumSBFL は使わずに既存の SBFL の結果を出力とする．

そこで本研究では，編集後のソースコード S_{curr} とテスト T_{curr}，及び，編集前のソースコード S_{prev} とテスト T_{prev} の関係に基づき，以下の 3 つのケースに分類して適用可否を決定する．ここで，$Pass(S,T)$ はソースコード S に対してテスト T を実行し，全テストケースが成功した際に真を返す関数と定義する．なお，前提として過去のバージョンではテストが成功し，現在のバージョンではテストが失敗している，つまり $Pass(S_{prev},T_{prev}) \wedge \neg Pass(S_{curr},T_{curr})$ である．

Case 1 : $\boldsymbol{S_{curr} = S_{prev} \wedge T_{curr} \neq T_{prev}}$
テストの追加や修正による潜在的な欠陥の顕在化などが該当する．ソースコードの差分が存在しない場合は GumSBFL を適用することはできない．よって既存の SBFL を用いる．

Case 2 : $\boldsymbol{S_{curr} \neq S_{prev} \wedge T_{curr} = T_{prev}}$
リファクタリングなどが該当する．直前のソースコードの編集が欠陥混入の原因であると考えられる．したがって，GumSBFL が適用可能である．

Case 3 : $\boldsymbol{S_{curr} \neq S_{prev} \wedge T_{curr} \neq T_{prev}}$
テストの失敗の原因が，ソースコードの編集によるものなのか，テストの変更によるものなのかを区別する必要がある．そこで本手法では，さらに詳細に状況を分類するために過去のソースコード S_{prev} に対して現在のテスト T_{curr} を実行し，結果に応じて以下の 2 通りに処理を分岐させる．

Case 3a : $\boldsymbol{Pass(S_{prev},T_{curr})}$
過去のコードでも現在のテストが通過するため，過去のコードには欠陥がなかった場合である．ソースコードの編集によって欠陥が混入したと判断でき，GumSBFL が適用可能である．

Case 3b : $\boldsymbol{\neg Pass(S_{prev},T_{curr})}$
編集以前から欠陥が潜伏していたか，あるいは仕様変更等によりテスト条件が満たせない場合である．GumSBFL の適用で精度が悪化する可能性があるため，既存の SBFL を用いる．

以上の判定プロセスにより，GumSBFL は直近の編集が直接的な原因で欠陥が混入したと推定される状況である Case2 と

Case3a に対してのみ選択的に適用される．それ以外の場合は既存手法に切り替えることで，常に安定した限局精度を維持する．また，この自動判定機構により，GumSBFL の適用判断を開発者が手動で行う必要がなくなる．

3.4 Step1. 既存 SBFL の適用

Step1 では，2.1 節の既存の SBFL を用いて各行の疑惑値の算出を行う．編集後のソースコードに対して編集後のテストを実行し疑惑値のリストを作成する．この段階で算出された疑惑値を $Susp_{base}(l)$ とする．

3.5 Step2. 編集差分の計算

Step2 ではソースコードの編集差分を計算する．この差分計算には，2.2 節の GumTree を用いる．前述の通り，AST 構造のみに依存する従来の GumTree の精度課題を解決するため，松本らの先行研究のアプローチ [10] を採用する．

3.6 Step3. 疑惑値の重み付け

Step3 では，Step2 で特定された編集差分に基づき，Step1 で算出したベース疑惑値 $Susp_{base}(l)$ に対して重み付けを行う．本手法では，単に変更の有無だけではなく，どのような編集が行われたかという開発コンテキストを考慮するために，ヒューリスティックな重み付けルールを定義する．

各行 l に対する最終的な疑惑値 $Susp_w(l)$ は，以下の式 (1) で定義する．ただし $r_k(l)$ は重み係数であり，l 行目がルール k の条件を満たす場合にルールで定められた固有の値をとり，満たさない場合は 1.0 とする．ルールは重複して適応されることがある．そのため $Susp_w(l)$ は，適用される全ルールの係数を乗算して算出される．

$$Susp_w(l) = Susp_{base}(l) \times \prod_k r_k(l) \tag{1}$$

各ルールの条件と重みは表 1 に示す．各ルールの詳細について説明する．

ルール 1 では，未編集の行に対して重み付けする．本手法は直近の編集が欠陥の原因であるという仮定に基づいている．編集されていない箇所に欠陥が含まれる可能性は低いと判断し，疑惑値を大幅に低下させる．

ルール 2 では，新機能が追加された行に対して重み付けする．既存コードの move を含まない insert は完全に新しい機能の実装である．検証済みの既存のコードを利用している箇所より欠陥を含む可能性が高いと考えて疑惑値を向上させる．

ルール 3 では，ソースコードの移動と書き換えが同時にされた行に対して重み付けする．単純な move は既存の機能を保ったまま位置を変える操作である．一方で，move の内部で update や insert が発生している場合，移動先に合わせた書き換えや書き足しが行われている．このような操作は単純な move より欠陥を含む可能性が高いと考えて疑惑値を向上させる．

ルール 4 では，構造の変更と同時にリテラルや変数名などの変更がされた行に対して重み付けする．他の編集操作を伴わない単独の update は，変数名のリネームなどの単純なリファクタリングである事が多い．しかし，move や insert の内部で行われる update は機能の変更を含む可能性が高いと考え，疑惑値を向上させる．

ルール 5 では，ソースコードの再利用がされた行に対して重み付けする．編集前のソースコードで delete の内部にあった move は，削除された機能の一部を切り出して再利用している．このような操作は，単純な move に比べて欠陥を混入させるリスクが高いと判断し，疑惑値を向上させる．

ルール 6 では，機能が消失した可能性がある行に対して重み付けする．編集前のソースコードにおいて move の内部に delete が存在していた場合，move の際になにかしらの機能が消失してしまっている．このとき，編集後のソースコードでも必要となる処理まで誤って削除してしまっている可能性があるため，疑惑値を向上させる．

表 1: 疑惑値に対する重み付けルール

	適用条件	重み
ルール 1.	l 行目が編集されていない	0.1
ルール 2.	l 行目に move を内包しない insert がある	1.1
ルール 3.	l 行目に insert や update を内包する move がある	1.1
ルール 4.	l 行目に insert や move を外延する update がある	1.1
ルール 5.	l 行目に move があり，その move は編集前ソースコードにおいて delete を外延していた	1.1
ルール 6.	l 行目に move があり，その move は編集前ソースコードにおいて delete を内包していた	1.1

4. 実　　験

4.1 実験概要

本節では提案手法の有効性を確認するために実施した 2 つの実験について述べる．提案手法は 3.2 節で述べた通り，「編集状況の判定」と「編集差分を用いた SBFL」の 2 つの要素から構成される．本稿では後者の，編集差分を用いた疑惑値の重み付けの有効性に焦点を当てる．したがって「編集状況の判定」は実験の対象外とし，提案手法の適用対象であると事前に判明しているデータのみを題材として用いることとする．具体的には，以下の 2 つの RQ を設定した．

RQ1: GumSBFL は既存の SBFL と比較して，欠陥限局の精度を向上させることができるか

RQ2: 各重み付けルールは精度の向上にどの程度寄与しているか

4.2 実験題材

実験に用いる題材の条件は以下の通りである．

1. 欠陥混入前後のソースコードがある
2. 欠陥混入前後のテストがある
3. 真の欠陥箇所が既知である

条件 1 および 2 は，提案手法を適用するための前提である．条件 3 は SBFL の欠陥限局精度の評価を行うために必要となる．

以上の 3 つの条件に加えて，欠陥が人工的でなく自然であることも重要である．GumSBFL は開発者の具体的な編集操作の内容に基づいて重み付けを行う．しかし，従来の SBFL

研究で用いられるミューテーションツール等による人工的な欠陥は，人間の編集操作が招く欠陥とは異なる．そのため，開発コンテキストを活用する GumSBFL の適切な評価のためには，より自然な欠陥が必要となる．

以上の条件を満たす題材として，競技プログラミングサイトである AtCoder(注1) の提出データを用いた．AtCoder では提出履歴が保存されており，条件 1 を満たすソースコードを収集可能である．また，これらは実際の参加者の試行錯誤によって生じたコードであるため，自然な欠陥であるという要件も満たしている．なお，条件 2 のテストは問題文や入出力例に基づき著者らで作成した．条件 3 に該当する真の欠陥箇所の特定も著者らが人手で行った．

本実験では，AtCoder 上で提出された Java 言語のソースコードを収集対象とした．収集したデータの中から，3.3 節で定義した Case2 のリファクタリングに該当する事例 164 件を抽出して使用した．具体的には以下の条件を満たすソースコードの組とテスト結果を実験題材とした．

- 直前のソース S_{prev} は全てのテストを通過している
- 現在のソース S_{curr} では 1 つ以上のテストに失敗している
- テストの変更はない

4.3 評価指標

提案手法による欠陥限局精度の変化を評価するために，SBFL の先行研究で頻繁に用いられている EXAM [12] を指標として採用する．EXAM とは，疑惑値のランキングに従ってソースコードを検査した際に，真の欠陥行に到達するまでにプログラム全体の何 % を調べる必要があるかを示す指標である．したがって，値が小さいほどより少ない検査コストで欠陥が特定できたことを意味する．

SBFL の欠陥限局精度の評価には同率問題を考慮する必要がある．3.1 節で述べたように SBFL では複数の行の疑惑値が等しくなることがあり，同じ疑惑値のグループ内での検査順序によって EXAM の値は変動する．そこで本実験では，同じ疑惑値の行はランダムな順序で検査されると仮定し，その順位の期待値である Rank_{avg} を用いる．真の欠陥行よりも疑惑値が高い行の数を N_{high}，真の欠陥行と疑惑値が等しい行の数（欠陥行自身を含む）を N_{tie} とするとき，Rank_{avg} は以下の式 (2) で定義される．

$$\mathrm{Rank}_{avg} = \mathrm{N}_{high} + \frac{\mathrm{N}_{tie} + 1}{2} \tag{2}$$

これは，同じ疑惑値のグループ内で欠陥行が最初に見つかる場合の順位（$\mathrm{N}_{high} + 1$）と，最後に見つかる場合の順位（$\mathrm{N}_{high} + \mathrm{N}_{tie}$）の平均値に相当する．最終的な評価指標である EXAM の値は以下の式で算出される．

$$\mathrm{EXAM} = \frac{\mathrm{Rank}_{avg}}{\text{全行数}} \times 100\% \tag{3}$$

4.4 実験設計

本実験では，RQ1 と RQ2 に答えるために以下の手順で分析を行う．まず，RQ1 に答えるために，GumSBFL と SBFL で欠陥限局精度の比較を行う．4.2 節の実験題材に対し，既存 SBFL と GumSBFL 両方で欠陥行の Rank_{avg} を求め，EXAM を計算する．GumSBFL の EXAM の値が，既存の SBFL の EXAM の値から減少している場合，提案手法によって疑惑値が改善され，欠陥限局の精度が向上したと判定する．

次に，RQ2 に答えるために，GumSBFL を構成する 6 つの重み付けルールについて，それぞれの貢献度を調査する．具体的には各ルール $k(k = 1, \ldots, 6)$ のみを単独で適用し，それ以外の重みを 1.0 とした 6 通りの設定で GumSBFL を実行する．それぞれの設定で算出された EXAM を既存 SBFL と比較することで，各ルールが精度の向上にどの程度寄与しているかを確認する．

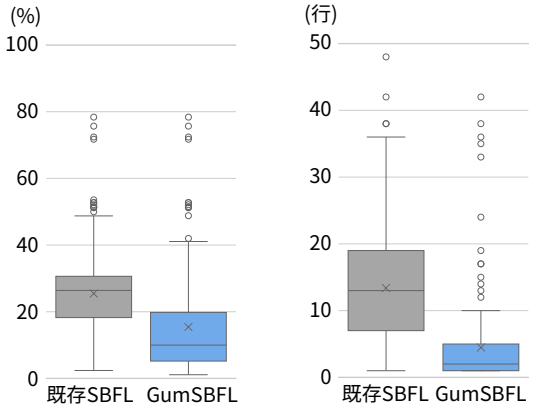

図 3: 欠陥限局精度の比較

図 4: 欠陥行と同じ疑惑値をもつ行数の比較

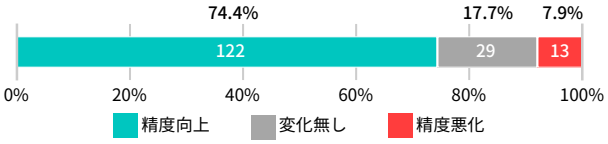

図 5: EXAM が変化した事例の割合

4.5 結果と考察

164 件の AtCoder 提出データに対し，既存 SBFL と GumSBFL を適用して算出された EXAM の分布を図 3 に示す．また，各指標において GumSBFL によって EXAM の値がどう変化したかの内訳を図 5 に示す．図 3 を確認すると，GumSBFL の箱ひげ図が既存 SBFL よりも小さい値へと推移している．これは提案手法によって，多くの事例でより少ない検査コストで欠陥を特定できるようになったことを示している．具体的な数値を確認すると，EXAM の平均値は 25.5% から 15.4% へと減少し，39.7% 改善された．中央値は 26.4% から 10.0% へと大幅に減少し，改善率は 62.1% を記録した．図 5 の個別の事例に着目する．精度の向上は全 164 件中 122 件（74.4%）で確認された．一方で，精度が低下した事例は 13 件（8.5%）に留まっている．これらの結果は，GumSBFL が開発者の平均的なデバッグ作業の負担を確実に軽減できることを示唆している．

EXAM の大幅な改善は，GumSBFL が同率問題を解消した

(注1)：https://atcoder.jp

ことに起因すると考えられる．ここで既存 SBFL と GumSBFL における，欠陥行と同じ疑惑値をもつ行数の分布を図 4 に示す．既存 SBFL では同率問題が原因でランキング上で大きな同順位のグループを形成して Rank_{avg} の値を悪化させていた．GumSBFL では重み付けルールにより検査対象の絞り込みや同順位の細分化を行った．図 4 で疑惑値が等しい行数が大幅に削減されていることがわかる．同じ疑惑値のグループが解体され，真の欠陥行が疑惑値リストの上位に浮上したことで EXAM が改善された．

RQ1 への回答：GumSBFL は既存の SBFL よりも高精度な欠陥限局を実現した．全データの 74.4% で精度が向上し，平均 EXAM は約 39.7% 改善された．

各ルールを単独で適用した場合の平均 EXAM，平均改善率，および精度が向上した件数を表 2 に示す．なお，平均改善率は，既存 SBFL の平均 EXAM に対する，各手法適用後の平均 EXAM の減少割合として求めた．表 2 より，全ルールにおいて適用時の平均 EXAM が既存 SBFL を下回っており，各ルールがそれぞれ欠陥限局精度の向上に寄与できている．特にルール 1 は平均改善率が 29.4% と最も高く，向上件数も 120 件であり全事例の約 73% に達した．ルール 2 からルール 6 の中ではルール 5 の貢献度が最も高く，平均改善率は 17.3% であり，適用件数は 56 件であった．最も貢献度が低かったルールはルール 6 であり，改善率は 9.3% であった．

まずルール 1 の高い貢献度について考察する．この結果は欠陥は開発者の編集操作によって混入するという本手法のキーアイデアが多くの事例において成立したことを示している．つまり，探索対象を編集箇所に絞り込むことは欠陥限局精度の向上に有効である．

次に，変更の有無のみを用いる場合と，GumSBFL のように変更の内容まで考慮した際の差異について考える．全ルール適用時の GumSBFL の EXAM と，ルール 1 のみ適用時の EXAM に改善率には約 10% の性能差が確認された．この性能差はルール 2 からルール 6 によるものである．つまり，単に変更が行われたという情報だけでは不十分であり，開発コンテキストの詳細に基づいた重み付けの有効性を示している．

表 2: 各ルールの精度向上への貢献度

	平均 EXAM	平均改善率	向上件数
既存 SBFL	25.5%	—	—
GumSBFL(全ルール)	15.4%	39.7%	122 件
ルール 1 のみ	18.0%	29.4%	120 件
ルール 2 のみ	21.9%	14.1%	48 件
ルール 3 のみ	21.9%	14.4%	49 件
ルール 4 のみ	22.1%	13.5%	42 件
ルール 5 のみ	21.1%	17.3%	56 件
ルール 6 のみ	23.2%	9.3%	29 件

※ EXAM は小さいほど高精度．改善率は大きいほど高精度．

RQ2 への回答：全ルールが欠陥限局精度向上に貢献し，特にルール 1 は 29.4% の改善率であった．また，ルール 2 からルール 6 による約 10% の精度向上は，単なる編集の有無だけでなく，その詳細な内容の活用の有効性を示している．

5. おわりに

本稿では，ソースコードの編集差分を用いて SBFL を改良する手法を提案した．AtCoder の提出データを用いた実験の結果，平均的な検査コストを約 39.7% 改善し，手法の有効性と AST 解析による具体的な編集内容活用の重要性を示した．

今後の課題として，まず「編集状況の判定」の評価が挙げられる．実用化に向け，提案手法が適さない状況で，判定機能が正しく動作し，精度低下を防げるかの検証が必要である．また，重み付けルールの改善も課題である．編集操作の種類に加え，対象となる文の種類（if 文か代入文か等）を考慮する等，AST 情報をより詳細に活用することで更なる精度向上が期待できる．

謝辞 本研究の一部は，JSPS 科研費（JP25K15056, JP25K03102, JP24H00692）による助成を受けた．

文　献

[1] H.A. deSouza, M.L. Chaim, and F. Kon, “Spectrum-based software fault localization: A survey of techniques, advances, and challenges,” arXiv e-prints, p.arXiv:1607.04347, 2016.

[2] C. Liu, L. Fei, X. Yan, J. Han, and S.P. Midkiff, “Statistical debugging: A hypothesis testing-based approach,” Trans. Software Engineering, vol.32, no.10, pp.831–848, 2006.

[3] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, “Capturing propagation of infected program states,” Proc. Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering, pp.43–52, 2009.

[4] T. Wang and A. Roychoudhury, “Automated path generation for software fault localization,” Proc. International Conference on Automated Software Engineering, pp.347–351, 2005.

[5] E.M. Maximilien and L. Williams, “Assessing test-driven development at IBM,” Proc. International Conference on Software Engineering, pp.564–569, 2003.

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” Proc. International Conference on Automated Software Engineering, pp.313–324, 2014.

[7] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” Trans. Software Engineering, vol.42, no.8, pp.707–740, 2016.

[8] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J. Van Gemund, “A practical evaluation of spectrum-based fault localization,” Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.

[9] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, “Generating accurate and compact edit scripts using tree differencing,” Proc. International Conference on Software Maintenance and Evolution, pp.264–274, 2018.

[10] J. Matsumoto, Y. Higo, and S. Kusumoto, “Beyond GumTree: A hybrid approach to generate edit scripts,” Proc. International Conference on Mining Software Repositories, pp.550–554, 2019.

[11] Q.I. Sarhan and Á. Beszédes, “A survey of challenges in spectrum-based software fault localization,” Trans. IEEE access, vol.10, pp.10618–10639, 2022.

[12] L. Naish, H.J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” Trans. Software Engineering and Methodology, vol.20, no.3, pp.1–32, 2011.