特別研究報告

題目

ソースコードの編集差分を用いたSBFLの精度改善

指導教員

楠本 真二 教授

報告者

川村 颯

令和 8 年 2 月 9 日

大阪大学基礎工学部情報科学科

令和 7 年度 特別研究報告

ソースコードの編集差分を用いた SBFL の精度改善

川村 颯

内容梗概

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に基づき，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（Abstract Syntax Tree: 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編集差分を用いた SBFL 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．

主な用語

Spectrum-Based Fault Localization，SBFL，編集差分，GumTree

目次

図目次

表目次

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，欠陥を含んでいる可能性の高さを示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]．その一つは疑惑値の計算式に関する研究である [2][3][4]．ソースコードの各行には，{成功した | 失敗した} テストケースが {通過した | 通過しなかった} 回数，という 4 種類の情報が記録される．この 4 種類の回数をどのように重ね合わせるかによって，SBFL の限局性能は大幅に変わることが知られている [5]．他にも，単なる回数の重ね合わせではなく条件確率等の統計的な視点を組み込んだ手法 [6][7] や，制御フロー等のソースコードの構造を考慮した手法 [8][9][10]，変更波及解析等の動的な解析を組み込んだ手法 [11] などが存在する．さらには，SBFL の入力となるテストケースを改善する方法 [12][13] 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スイートに限定している．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．欠陥のない状態から欠陥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（Test-Driven Development : TDD）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 [14][15]．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[16] 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．

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スイートを入力とし，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 はその直感的かつ単純な原理から可搬性が高く，特定のプログラミング言語や環境に依存しないという利点がある．そのため，C 言語や Java といった静的型付け言語だけでなく，Python のような動的型付け言語や，さらにはスプレッドシートに至るまで，多岐にわたる対象に適用されている [17][18]．また，自動プログラム修正（Automated Program Repair: APR）における修正箇所の特定フェーズに組み込まれる [19] など，デバッグ支援の枠を超えた応用もされている．

SBFL では，プログラムの構成要素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．疑惑値の算出粒度にはステートメントやメソッド等様々あるが，本研究においてはソースコードの行を疑惑値算出の単位とする．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[20] や Jaccard[21] など多数存在する [22] が，本研究では多くの先行研究で高い精度が報告されている Ochiai[5] 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[16] がある．GumTree の実行例を図 1 に示す．一般的な `diff` コマンドは，最長共通部分列（Longest Common Subsequence: LCS）に基づきテキスト上での差分を行単位で計算する．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，AST でのノード単位で差分を計算する．これにより，単純なテキスト差分では検出できない構造的な変更も検出できる．

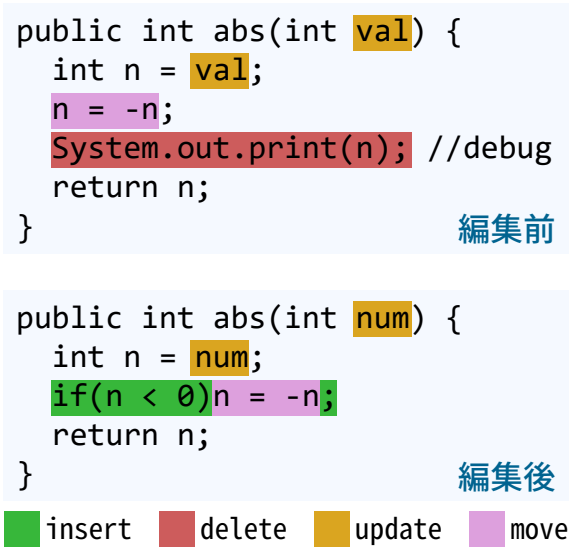

図 1　GumTree の実行例

それに加えて，改行や空白，コメント文などのプログラムの動作に関係がない要素の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操作列であり，その操作回数が最小となるように算出される．従来の `diff` コマンドは行の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 の処理は主にノードのマッチングと編集スクリプト生成の 2 つの工程で構成されている．特に重要なのはノードのマッチング処理であり，これは変更前後の AST のノードの対応関係を特定する工程である．マッチング処理は以下の 2 段階で行われる．

トップダウンフェーズ．AST を根から探索し，高さなどの特徴が完全に一致する最大の部分木を探し，同一のノードとして対応付ける．

ボトムアップフェーズ．トップダウンフェーズで対応付けられなかったノードに対し，子ノードの類似度に基づき対応付ける．

このマッチング結果に基づき，編集スクリプトが生成される．

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 [23]．例えば，本来は move や update として扱われるべき編集が，不適切なマッチングが原因で，大量の delete と insert として扱われることがある．これは GumTree のマッチング処理が 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 [24] 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 のマッチング精度を向上させる手法である．具体的には，ソースコードを行単位で「変更がある行」と「変更がない行」に分類し，「変更がない行」に含まれるノードを優先的にマッチングする．これにより，編集前後のノードの対応関係をより高精度に特定し，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 [25] 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，疑惑値の計算がテストの通過・非通過の情報のみに依存しているため，ソースコード内に条件分岐が少ない事やテストケース数が不足している事が原因で発生する．同じ疑惑値の行が多数存在すると，開発者はその中から手作業で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変更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．

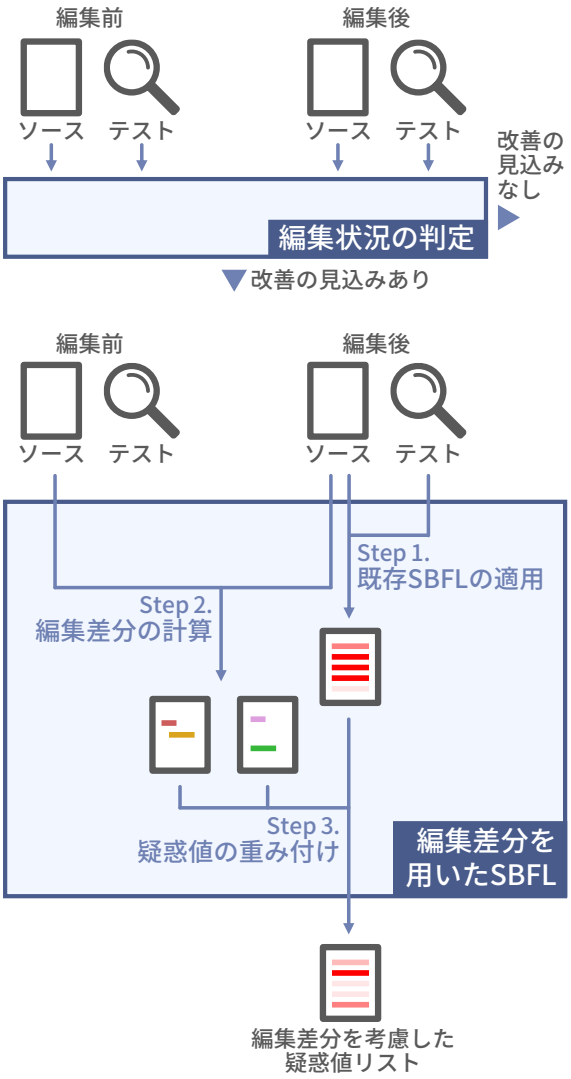

図 2　提案手法の概要図

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．3.1 節で述べたように，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，テストの仕様変更等が原因でテストに失敗するようになった場合で

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，現在のソースコード 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 : $\boldsymbol{S_{curr} = S_{prev} \wedge T_{curr} \neq T_{prev}}$

ソースコードに変更がなく，テストが変更された結果，テストに失敗するようになった状況である．具体的には，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．GumSBFL はソースコードの編集差分を活用する手法であるため，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 : $\boldsymbol{S_{curr} \neq S_{prev} \wedge T_{curr} = T_{prev}}$

テストに変更がなく，ソースコードが変更された結果，テストに失敗するようになった状況である．具体的には，リファクタリングなどが該当する．この場合は直前のソースコードの編集が原因で欠陥が混入した可能性が高いと考えられる．したがって，GumSBFL が適用可能である．

Case 3 : $\boldsymbol{S_{curr} \neq S_{prev} \wedge T_{curr} \neq T_{prev}}$

ソースコードとテストの両方が変更された結果，テストに失敗するようになった状況である．この状況では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，テスト自体の不備や仕様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，その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 : $\boldsymbol{\neg Pass(S_{prev},T_{curr})}$

過去のソースコードでも現在と同様にテストに失敗するため，欠陥は今回の変更以前から潜伏していたか，あるいは仕様変更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，提案手法は適用せず，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．現在のソースコードに対して現在のテストを実行し，カバレッジ情報とテスト結果を収集する．得られた実行情報を Ochiai の計算式に適用し，プログラム内の各行に対する疑惑値を計算する．この段階で算出された，l 行目に対する疑惑値を $Susp_{base}(l)$ とする．

3.5 Step2．編集差分の計算

Step2 では編集前のソースコード S_{prev} から現在のソースコード S_{curr} へ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 [24] を採用する．精度の低い編集スクリプトは後続の重み付け処理に悪影響を及ぼすためである．

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 $Susp_{base}(l)$ 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，変更が加えられていない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

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

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．

以上のルールはその役割によって大きく 2 つのカテゴリに分類される．第一のカテゴリは「検査対象の絞り込み」である．これはルール 1 が該当する．本手法の仮定に基づき，欠陥を可能性が低い未編集の行の疑惑値を一律に下げる．それによりランキングの上位から未編集の行を排除し，開発者が検査すべき行の数を減らす．第二のカテゴリは「同順位の細分化」である．これはルール 2 からルール 6 が該当する．これらのルールでは，単純な編集操作よりも欠陥混入のリスクが高いと思われる，複雑な編集が行われた行の疑惑値を高くする．これによりランキングで同順位となっていた行の中に順位差を生み，より欠陥混入のリスクの高い行を上位へと浮上させる．step3 ではこの「検査対象の絞り込み」と「同順位の細分化」という 2 つのアプローチによって疑惑値を調整する．

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．具体的には，以下の 2 つの RQ を設定し，それぞれに対応する実験を行う．

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

実験 1 では，RQ1 に答えるために，GumSBFL と既存の SBFL の精度比較を行う．4.2 節で述べる実験題材に対し，提案手法と既存手法をそれぞれ適用し，4.3 節で定義する評価指標を用いて，欠陥箇所の特定における提案手法の有効性を定量的に評価する．

実験 2 では，RQ2 に答えるために，GumSBFL を構成する各ルールの貢献度の分析を行う．提案手法は複数のヒューリスティックルールによって構成されている．各ルールを単独で使用した際の欠陥限局精度の変化から，各ルールがそれぞれどの程度スコアの改善に寄与しているかを調査する．なお，実験題材と評価指標は実験 1 と共通のものを用いる．

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．

以上の条件をすべて満たす題材として，競技プログラミングサイトである AtCoder[*1]の提出データを

[*1] https://atcoder.jp

用いた．AtCoder では，提示された問題に対して参加者がソースコードを提出し，システムが用意したテストケース群によって正誤が判定される．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[26][27] 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，EXAM の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し，欠陥限局精度が高いことを表す．

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．Wong ら [22] が指摘しているように，このような状況では単一の順位のみを用いた評価だけでは検証として不十分である．そこで本実験では，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} とし，以下の 3 種類の順位を定義する．

$$\mathrm{Rank}_{bst} = N_{high} + 1 \tag{2}$$

$$\mathrm{Rank}_{wst} = \mathrm{Rank}_{bst} + N_{tie} \tag{3}$$

$$\mathrm{Rank}_{avg} = \frac{\mathrm{Rank}_{bst} + \mathrm{Rank}_{wst}}{2} \tag{4}$$

Rank_{bst} は同じ疑惑値のグループ内で，欠陥行が最初に発見される最良の場合を想定した順位である．Rank_{wst} は同じ疑惑値のグループ内で，欠陥行が最後に発見される最悪の場合を想定した順位である．Rank_{avg} は同じ疑惑値のグループがランダムな順序で検査された場合の順位であり，期待値である．各 $Rank$ での EXAM は以下の式 (5) で算出される．

$$\mathrm{EXAM} = \frac{\mathrm{Rank}}{\text{全行数}} \times 100\% \tag{5}$$

以降では, Rank_{avg}, Rank_{wst}, Rank_{avg} を用いて算出した EXAM を EXAM_{bst}, EXAM_{wst}, EXAM_{avg} と呼ぶ.

4.4 実験 1: 提案手法により欠陥限局の精度は向上するか

4.4.1 実験設計

実験 1 では，GumSBFL の有効性を検証するために，GumSBFL と SBFL で欠陥限局精度の比較を行う．まず，4.2 節の実験題材に対し，既存 SBFL と GumSBFL 両方で欠陥行の Rank_{bst}，Rank_{wst}，Rank_{avg} を計算する．その後，それぞれの Rank について 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．

4.4.2 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_{bst}，EXAM_{wst}，EXAM_{avg} 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 4 に示す.

EXAM_{wst} と EXAM_{avg} は精度の向上が確認できた．図 3 を確認すると，EXAM_{wst} と EXAM_{avg} において GumSBFL の箱ひげ図が既存 SBFL よりも小さい値へと推移しており，欠陥限局の精度が向上していることがわかる．図 4 の，件数に基づく比較においては，EXAM_{wst} では全データの 80.5%,

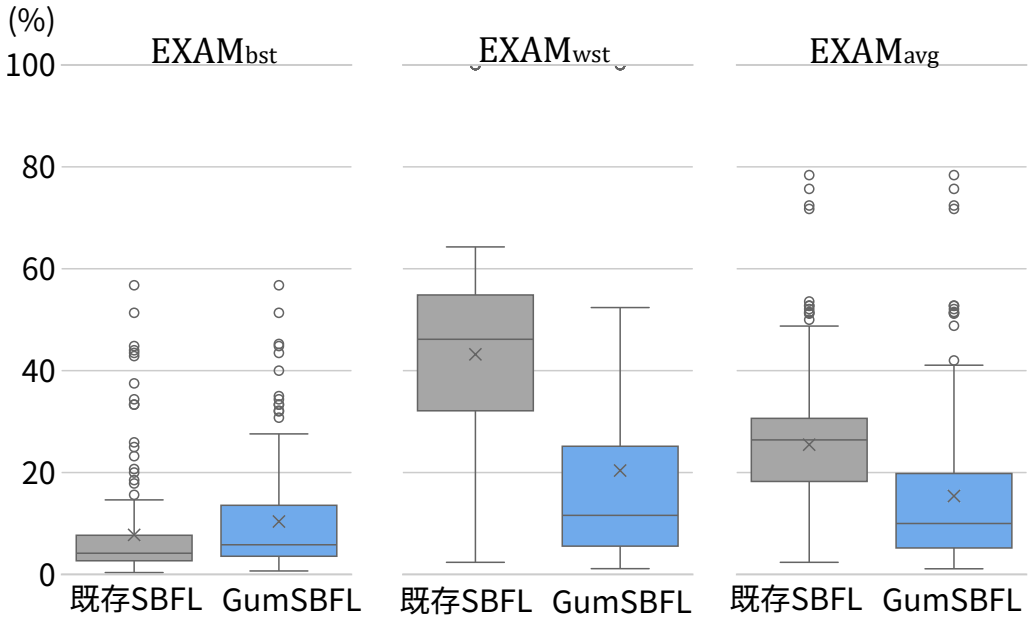

図 3　欠陥限局精度の比較

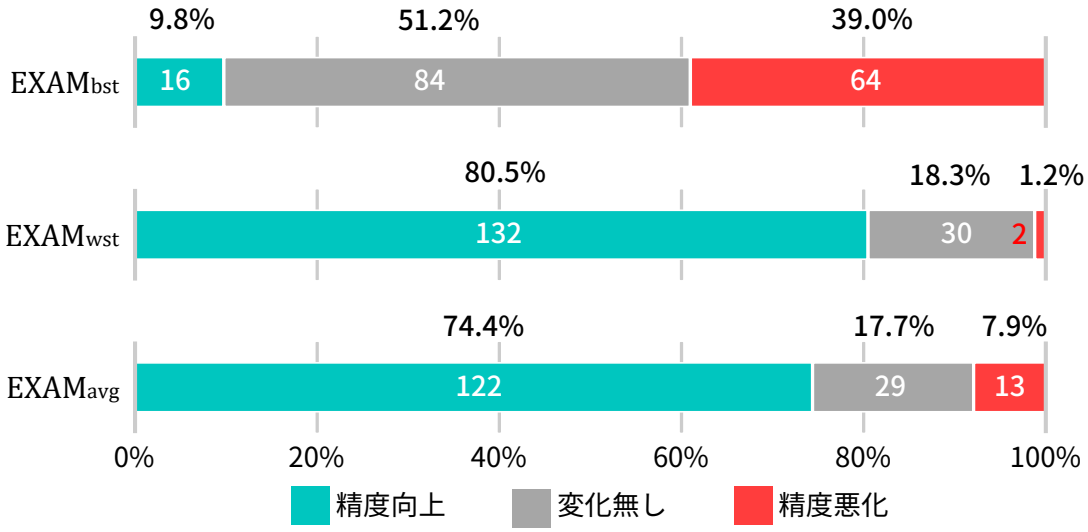

図 4　EXAM が変化した事例の割合

表 2　既存 SBFL と GumSBFL の平均 EXAM スコアおよび改善率の比較

	平均 EXAM(%)		改善効果	
	既存 SBFL	GumSBFL	減少量	改善率
EXAM_{bst}	7.8	10.4	−2.6	−33.7%
EXAM_{wst}	43.3	20.4	22.9	52.9%
EXAM_{avg}	25.5	15.4	10.1	39.7%

※ EXAM は小さいほど高精度．改善率は大きいほど高精度．

EXAM_{avg} では 74.4% の事例で精度が向上した．また，表 2 に示す平均 EXAM の比較においても，EXAM_{wst} は 52.9%，EXAM_{avg} は 39.7% という大幅な改善率を記録している．

一方で，EXAM_{bst} に関しては異なる傾向が見られた．図 3 で EXAM_{bst} は GumSBFL の値が既存 SBFL の値より大きい値に分布した．図 4 においても，EXAM_{bst} は 39.0% の事例で精度が低下し，精度の向上は 9.8% に留まった．これらのことより，最悪ケース Rank_{wst} および平均ケース Rank_{avg} の観点では GumSBFL が優れているが，最良ケース Rank_{bst} の観点では既存 SBFL が優れているという結果が得られた．

EXAM_{wst} の大幅な改善は，GumSBFL が同率問題を解消した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，欠陥行と同順位の行数の分布を図 5 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，Rank_{wst} の値を悪化させていた．3.6 節で述べたように GumSBFL では「検査対象の絞り込み」や「同順位の細分化」を行った．図 5 で同順位の行数は大幅に削減されていることがわかる．これにより同順位のグループが解体され，真の欠陥

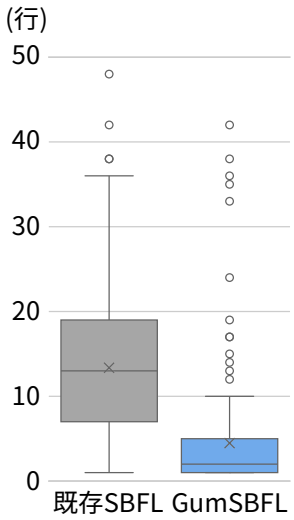

図 5　欠陥行と同順位の行数の分布

行がグループから抜け出して上位に浮上したことで EXAM_{wst} が改善された．

EXAM_{bst} の悪化は，重み付けルールの「同順位の細分化」の性質によると考えられる．提案手法によって各行に細分化された順位がつくと，EXAM_{bst} は悪化として扱われてしまう可能性がある．例えば，既存 SBFL において，欠陥行が多くの行と同順位で 1 位タイであった場合，同順位の行数に関係なく Rank_{bst} は 1 位となる．しかし，GumSBFL は同順位のグループを解体し，各行に明確な順位をつける．その結果，欠陥行が同順位であった他の行よりわずかでも下位にランク付けされてしまうと，順位は 2 位や 3 位となり数値上では精度の低下として扱われる．すなわち，同順位の行数が多いことに起因して Rank_{bst} が高順位となっていた事例において，同順位グループの解体によって順位が下がり，EXAM_{bst} が悪化することがある．

実用的な観点において最も重視すべきは EXAM_{avg} である．実際のデバッグ作業において同じ疑惑値の行群を常に最良の順番 Rank_{bst} で検査できるとは限らず，また常に最悪の順序 Rank_{wst} で検査することもないからである．したがって，その期待値である EXAM_{avg} が現実的な検査コストを反映しているといえる．表 2 において EXAM_{avg} は平均して 39.7% 改善され，図 4 において全事例の 74.4% で精度の向上が確認された．これは GumSBFL が開発者の平均的な検査コストを削減できることを表している．

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．GumSBFL の適用によって EXAM_{wst} は改善され，EXAM_{bst} は悪化する傾向にある．しかし，開発者の平均的な検査コストを示す EXAM_{avg} においては全データの 74.4% で精度が向上し，平均 EXAM は約 39.7% 改善された．

4.5 実験 2: 各ルールは欠陥限局の精度向上にどう貢献しているか

4.5.1 実験設計

実験 2 では，GumSBFL を構成する 6 つの重み付けルールが欠陥限局の精度向上にそれぞれどの程度寄与しているのかの調査を行う．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．評価指標には，EXAM_{avg} を用いる．算出された 6 通りの EXAM_{avg} を，既存 SBFL の EXAM_{avg} と比較し，その改善量および改善した事例数を集計する．

4.5.2 結果と考察

表 3 より，全ルールにおいて適用時の平均 EXAM_{avg} 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ることがわかった．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 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．全ルール適用時のの EXAM_{avg} と，ルール 1 のみ適用時の EXAM_{avg} に改善率に約 10% の

表 3　各ルール単独適用時の平均改善率と向上件数

	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．

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，GumTree を用いて開発コンテキストの詳細に基づいた重み付けをすることの有用性を示している．

RQ2 への回答：重み付けルールは全て欠陥限局精度向上に寄与している．特にルール 1 は平均改善率が 29.4% と最も高い．また，ルール 2 からルール 6 は，ルール 1 の単独使用時と比較してさらに約 10% の精度向上をもたらした．これは，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　妥当性への脅威

本稿の実験では AtCoder の提出コードを用いた．これらは数百行程度の小規模なプログラムであり，アルゴリズムの実装に特化している．そのため，大規模な GUI アプリケーションや Web システムなど，より複雑な構造や依存関係を持つ実際のソフトウェア開発において，GumSBFL が同等の有効性を示すことは保証されない．また，本実験では Java 言語のみを対象とした．GumSBFL は AST 解析を用いるため，重み付け効果は言語仕様に依存する部分がある．Python や C++ などの他のプログラミング言語でも本稿で提案した重み付けルールが同様に有効であるかは検証の余地が残されている．

6　おわりに

本稿では，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手法を提案した．提案手法では，GumTree を用いて編集操作を特定し，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．AtCoder の提出データを用いた評価実験の結果，提案手法は既存の SBFL と比較して平均的な検査コスト EXAM_{avg} を約 39.7% 改善し，その有効性を示した．また，単純な変更の有無だけでなく，AST 解析による詳細な編集内容の活用が精度向上に有効であることも確認した．

今後の課題として，3.3 節で述べた「編集状況の判定」の評価が挙げられる．本稿の実験では，提案手法の核となる重み付けの効果の確認のため，提案手法が有効に機能する状況のみを対象とした．したがって，実用化に向けては提案手法が適さない状況で，「編集状況の判定」が精度の低下をどの程度防げるのかを評価する必要がある．

また，重み付けルールの改善も課題である．現在のルールはヒューリスティックに定義されており，編集操作の種類のみに基づいている．しかし，同じ編集操作であっても，その対象が if 文のような条件式であるか，単なる代入文であるかによって，欠陥を混入する確率は異なると考えられる．AST のノード情報をより詳細に活用する等のアプローチを加えることでさらなる精度向上が可能になると考える．

謝辞

本研究の遂行にあたり、多大なるご指導とご協力を賜りました。ここに深く感謝の意を表します。

楠本真二教授には、研究の節目となる中間報告の場において、本質を突く鋭いご指摘をいただきました。先生の客観的な視点からのご助言は、独りよがりになりがちな自身の研究を再検討する機会となり、研究の質を一段と高めることができました。

柗本真佑准教授には、研究テーマの着想から本稿の完成に至るまで、終始熱心なご指導を仰ぎました。特に個人ミーティングや論文添削を通じて、研究者としての論理的思考力や姿勢を厳しくも温かくご教示いただきました。学生の主体性を尊重しつつ、重要な局面で的確な道筋を示してくださったことに、心より御礼申し上げます。

事務補佐員の橋本美砂子様には、物品の管理や各種手続きなど、多岐にわたる業務でサポートしていただきました。事務室での温かい対応は、日々の研究活動の励みとなりました。きめ細やかなサポートに心より御礼申し上げます。

楠本研究室のメンバーにも深く感謝しています。先輩方の背中を追うことで目標を見失わずに進むことができ、同期の仲間とは、研究の議論のみならず、プライベートでも多くの時間を共有し、精神的な支えとなりました。

最後に、これまでの学生生活を一番近くで支え、応援してくれた家族に改めて感謝の気持ちを伝えます。ありがとうございました。

参考文献

[1] Souza, de H. A., Chaim, M. L. and Kon, F.: Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges, *arXiv e-prints*, p. arXiv:1607.04347 (2016).

[2] Orso, A., Jones, J., Harrold, M. and Stasko, J.: GAMMATELLA: visualization of program-execution data for deployed software, in *Proc. International Conference on Software Engineering*, pp. 699–700 (2004).

[3] Ali, S., Andrews, J. H., Dhandapani, T. and Wang, W.: Evaluating the Accuracy of Fault Localization Techniques, in *Proc. International Conference on Automated Software Engineering*, pp. 76–87 (2009).

[4] Abreu, R., González, A., Zoeteweij, P. and Gemund, van A. J. C.: Automatic software fault localization using generic program invariants, in *Proc. Symposium on Applied Computing*, pp. 712–717 (2008).

[5] Abreu, R., Zoeteweij, P., Golsteijn, R. and Van Gemund, A. J.: A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792 (2009).

[6] Liu, C., Fei, L., Yan, X., Han, J. and Midkiff, S.: Statistical Debugging: A Hypothesis Testing-Based Approach, *Trans. Software Engineering*, Vol. 32, No. 10, pp. 831–848 (2006).

[7] Arumuga Nainar, P., Chen, T., Rosin, J. and Liblit, B.: Statistical debugging using compound boolean predicates, in *Proc. International Symposium on Software Testing and Analysis*, pp. 5–15 (2007).

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

[9] Zhao, L., Wang, L., Xiong, Z. and Gao, D.: Execution-aware fault localization based on the control flow analysis, in *Proc. International Conference on Information Computing and Applications*, pp. 158–165 (2010).

[10] Yu, K., Lin, M., Gao, Q., Zhang, H. and Zhang, X.: Locating faults using multiple spectra-specific models, in *Proc. Symposium on Applied Computing*, pp. 1404–1410 (2011).

[11] Alves, E., Gligoric, M., Jagannath, V. and d'Amorim, M.: Fault-localization using dynamic slicing and change impact analysis, in *Proc. International Conference on Automated Software Engineering*, pp. 520–523 (2011).

[12] Wang, T. and Roychoudhury, A.: Automated path generation for software fault localization, in *Proc. International Conference on Automated Software Engineering*, pp. 347–351 (2005).

[13] Zhang, X., Gupta, N. and Gupta, R.: Locating faults through automated predicate switching, in *Proc. International Conference on Software Engineering*, pp. 272–281 (2006).

[14] Maximilien, E. and Williams, L.: Assessing test-driven development at IBM, in *Proc. International Conference on Software Engineering*, pp. 564–569 (2003).

[15] Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M. and Erdogmus, H.: What Do We Know about Test-Driven Development?, *Trans. IEEE Software*, Vol. 27, No. 6, pp. 16–19 (2010).

[16] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: Fine-grained and accurate source code differencing, in *Proc. International Conference on Automated Software Engineering*, pp. 313–324 (2014).

[17] Widyasari, R., Prana, G. A. A., Haryono, S. A., Wang, S. and Lo, D.: Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects, *Trans. Empirical Software Engineering*, Vol. 27, No. 6, p. 147 (2022).

[18] Getzner, E., Hofer, B. and Wotawa, F.: Improving Spectrum-Based Fault Localization for Spreadsheet Debugging, in *Proc. International Conference on Software Quality, Reliability and Security*, pp. 102–113 (2017).

[19] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-Performance, High-Extensibility and High-Portability APR System, in *Proc. Software Engineering Conference*, pp. 697–698 (2018).

[20] Jones, J. A., Harrold, M. J. and Stasko, J. T.: Visualization for fault localization, in *Proc. Workshop on Software Visualization*, pp. 71–75 (2001).

[21] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A. and Ernst, M. D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking, *Trans. Software Engineering*, Vol. 36, No. 4, pp. 474–494 (2010).

[22] Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A Survey on Software Fault Localization, *Trans. Software Engineering*, Vol. 42, No. 8, pp. 707–740 (2016).

[23] Frick, V., Grassauer, T., Beck, F. and Pinzger, M.: Generating Accurate and Compact Edit Scripts Using Tree Differencing, in *Proc. International Conference on Software Maintenance and Evolution*, pp. 264–274 (2018).

[24] Matsumoto, J., Higo, Y. and Kusumoto, S.: Beyond GumTree: A Hybrid Approach to Gen-

erate Edit Scripts, in *Proc. International Conference on Mining Software Repositories*, pp. 550–554 (2019).

[25] Sarhan, Q. I. and Beszédes, Á.: A survey of challenges in spectrum-based software fault localization, *Trans. IEEE access*, Vol. 10, pp. 10618–10639 (2022).

[26] Naish, L., Lee, H. J. and Ramamohanarao, K.: A model for spectra-based software diagnosis, *Trans. Software Engineering and Methodology*, Vol. 20, No. 3, pp. 1–32 (2011).

[27] Wong, W. E., Debroy, V., Golden, R., Xu, X. and Thuraisingham, B.: Effective software fault localization using an RBF neural network, *Trans. Reliability*, Vol. 61, No. 1, pp. 149–169 (2011).