特別研究報告

題目

スペクトラムに基づく欠陥限局に適したプログラム構造の再調査

指導教員

楠本 真二 教授

報告者

久保 光生

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

スペクトラムに基づく欠陥限局に適したプログラム構造の再調査

久保 光生

内容梗概

プログラムに含まれる欠陥箇所を自動で推測する方法として, スペクトラムに基づく欠陥限局（SBFL）がある．SBFL では，各テストケースの成否と実行経路情報をもとに欠陥箇所を特定する．同じ機能を持つプログラムでも，その構造によって SBFL の欠陥限局精度には差が生じる．よって，プログラム構造を SBFL に適する形に変更することで，機能を保ちつつ SBFL の精度向上が期待される．先行研究では SBFL に適するプログラム構造を発見するために，SBFL スコアが提案された．SBFL スコアはプログラムが SBFL にどの程度適しているかを評価する指標の 1 つである．また，先行研究では，同じ機能を持つが構造の異なるプログラムペアを対象として SBFL スコアを計測し，SBFL スコアが高くなるプログラム構造を得ている．しかし，実験対象のプログラム数が 10 個，ミューテーション演算子が 11 種類と少ないことが課題であった．そこで本研究では，実験対象のプログラム数を約 36 倍，ミューテーション演算子の種類数を約 2.5 倍にして実験を行った．実験の結果，新たに SBFL スコアを高めるプログラム構造が 3 つ特定できた．

主な用語

SBFL，ミューテーションテスト，ソフトウェア品質モデル

目次

図目次

表目次

1　はじめに

ソフトウェア開発において，デバッグは多くの労力とコストを必要とする作業である．ソフトウェア開発に必要なコストのうち，半分以上をデバッグ作業が占めているという報告もある [1, 2]．そのため，デバッグを支援するための研究が盛んに行われている．デバッグ支援に関する研究分野の 1 つに欠陥限局がある．欠陥限局とは，プログラム中の欠陥箇所を推測する技術である．中でも近年，スペクトラムに基づく欠陥限局 (Spectrum-Based Fault Localization, 以降 SBFL) に関する研究が盛んに行われている [3]．SBFL では，テストがどの文を実行し，どの文を実行しなかったかという情報を用いて，プログラムの欠陥と疑われる箇所を自動的に特定する．SBFL の基本的なアイデアは，多くの失敗テストが通過する行ほど欠陥である可能性が高く，成功テストが多く通過する行ほど欠陥である可能性が低いと判断することである．

SBFL の精度は，欠陥自体の性質やテストの内容など，様々な要因に左右される [4]．その中でも，佐々木らによる先行研究 [5] では，プログラム構造に着目した．この先行研究では，プログラムがどの程度 SBFL に適するかを表す SBFL 適合性が提案されている．また，SBFL 適合性の評価指標として，SBFL スコアが提案されている．SBFL スコア計測の基本的なアイデアは，すべてのテストを通過するプログラムについて，ミューテーションテストを用いて，様々な箇所に意図的に欠陥を発生させることである [5]．これらの欠陥が SBFL によってどの程度正確に特定できたか計測することにより，元のプログラムの SBFL 適合性を評価できる．先行研究では同じ機能を持つが構造の異なる 5 組のプログラムペアを用いて SBFL スコアを計測することにより，SBFL に適するプログラム構造が調査された．しかし，計測対象のプログラム数が 10 個と少なく，SBFL に適するプログラム構造が十分に明らかになったとはいえない．また，プログラムに意図的に欠陥を発生させるためのミューテーション演算子の種類が 11 個と少なく，SBFL スコアの信頼性が低いことが課題であった．

そこで本研究では，365 個のプログラムを用いて SBFL スコアを計測することにより，SBFL に適する新たなプログラム構造の発見を目指す．また，SBFL スコアの信頼性を高めるために，16 種類の新たなミューテーション演算子を定義する．これにより，合計で 27 種類のミューテーション演算子をプログラムに適用する．

本研究では，SBFL 適合性が高いプログラム構造を明らかにする．まず，RQ 1 では，SBFL スコアの信頼性の向上を検証する．次に，RQ 2 では，考察対象プログラム数の増加を検証する．その後，本研究の主題である，SBFL 適合性が高いプログラム構造について RQ 3 を設定して調査する．

RQ 1：ミューテーション演算子の追加により SBFL スコアの信頼性は高まったか？

RQ 1 では，新たなミューテーション演算子の実装により，先行研究の課題である SBFL スコアの信頼性の低さが解決できたか確かめる．SBFL スコアの信頼性は，ミュータントの数が多いほど高まる．よって，プログラムの規模に対するミュータント数はどの程度増加するか検証する．検証の結果，約 98% のプログラムでミュータント数は増加し，信頼性が高まったことが確認できた．

RQ 2：SBFL スコアに変化が生じたグループ数はいくつか？

RQ 2 では考察対象プログラム数を計測する．データセットの利用により，実験対象プログラム数は増加した．しかし，プログラム構造が異なる場合でも，SBFL スコアが同じになる場合が考えられる．よって，実験対象プログラムの SBFL スコアを計測し，同機能グループのうち，SBFL スコアに差が生じるグループはいくつであるか確認する．結果，差が生じるグループ数は 130 組であり，考察対象プログラムは増加した．

RQ 3：SBFL スコアが高くなるプログラム構造とは何か？

RQ 3 では，SBFL に適するプログラム構造を調査する．調査の結果，新たに SBFL スコアが高くなる 3 つのプログラム構造を発見し，そのうち 1 つが先行研究の結果に反することを確認した．さらに，調査の結果得られた SBFL に適するプログラム構造をもとに，SBFL 適合性を高めるプログラムの変換方法を提案した．

2 準備

2.1 スペクトラムに基づく欠陥限局（SBFL）

デバッグを支援する技術の 1 つに，欠陥限局がある．欠陥限局とは，プログラム中の欠陥箇所を推測する技術である．テストを用いた自動的な欠陥限局方法の 1 つに，スペクトラムに基づく欠陥限局 (Spectrum-Based Fault Localization, SBFL) がある．SBFL におけるスペクトラムとは，テスト実行時にどの文を通過したかという実行経路情報である．SBFL の基本的なアイデアは，多くの失敗テストが通過する行ほど欠陥である可能性が高く，成功テストが多く通過する行ほど欠陥である可能性が低いと判断することである．

SBFL による欠陥箇所の特定方法について説明する．まず，すべてのテストを実行し，テストの成否と実行経路情報を記録する．次に，これらの情報を利用して，疑惑値と呼ばれる，欠陥である可能性の高さを示す値を文ごとに算出する．疑惑値の算出方法には様々あるが，Abreu らが SBFL で用いられる計算式の有効性を評価した結果，Ochiai の計算式 [6] が最も優れていると結論づけている [7].

Ochiai の計算式における疑惑値 $susp(s)$ の算出方法を (1) に示す．ここで，$fail(s)$ は文 s を実行した失敗テストの数，$pass(s)$ は文 s を実行した成功テストの数，$totalFail$ はすべての失敗テストの総数である．

$$susp(s) = \frac{fail(s)}{\sqrt{totalFail \times (fail(s) + pass(s))}} \tag{1}$$

この $susp(s)$ をすべての文 s に対して算出し，その値が高い文ほど，欠陥の原因箇所である可能性が高いと推測する．

なお，SBFL の疑惑値の算出においては，失敗テストの実行経路情報が最も重要な要素となる．失敗テストがどのような文を通過し，どのような文を通過しなかったかが，テストの失敗，すなわち欠陥の原因箇所に対する大きな手がかりとなるためである．上記 Ochiai 式では，分子が $fail(s)$ であることから，失敗テストの実行経路情報の重要さが見て取れる．

2.2 SBFL 適合性

先行研究 [5] で提案された SBFL 適合性について述べる．SBFL 適合性とは，プログラムが持つ品質特性の 1 つであり，プログラム自体が SBFL にどの程度適しているかを表す．プログラムの機能やテストスイートが同じでも，プログラム構造が異なれば SBFL を用いた欠陥限局の精度に違いが生じることがある．

プログラム構造の違いによる SBFL 適合性の変化について，例を用いて説明する．図 1 に示すプログラム (a) とプログラム (b) は，機能が同じだが，構造が異なる．図 1 に示すテスト (c) を用いて両方

プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
s_1: `boolean result = false;`	0.50	✓	✓	✓	✓
s_2: `if (0 < a)`	0.50	✓	✓	✓	✓
s_3: `  result = true;`	0.00	✓	✓		
🔥s_4: `if (0 <= b) //correct: 0 < b`	0.50	✓	✓	✓	✓
s_5: `  result = true;`	0.50	✓	✓	✓	✓
s_6: `return result;`	0.50	✓	✓	✓	✓
テスト結果：		P	P	P	F

(a)リファクタリング前

プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
s'_2: `if (0 < a)`	0.50	✓	✓	✓	✓
s'_3: `  return true;`	0.00	✓	✓		
🔥s'_4: `if (0 <= b) //correct: 0 < b`	0.71			✓	✓
s'_5: `  return true;`	0.71			✓	✓
s'_6: `return false;`	-				
テスト結果：		P	P	P	F

(b)リファクタリング後

テストケース	入力 (a, b)	期待値	実際の値
t_1:	(1, 1)	true	true
t_2:	(1, 0)	true	true
t_3:	(0, 1)	true	true
t_4:	(0, 0)	false	true

(c)テストスイート

図 1　プログラム構造の違いにより SBFL 適合性が変化する例

のプログラムに SBFL を実行すると，各文の疑惑値が算出される．プログラム (a) では，欠陥がある箇所と同じ疑惑値を持つ文が 4 個存在する．一方，プログラム (b) では，欠陥がある箇所と同じ疑惑値を持つ文は 1 個である．欠陥がある箇所と同じ疑惑値を持つ文が少ないほど，確認するべき文が少なくなるため，SBFL による欠陥限局の精度が高いといえる．よって，この場合，(b) の方が SBFL 適合性が高い．

2.3　SBFL スコア

SBFL 適合性と同時に先行研究 [5] で提案された SBFL スコアについて述べる．SBFL スコアは，SBFL 適合性の評価指標の 1 つとして提案された．SBFL スコアを計測するための基本的なアイデアは，与えられたプログラムに意図的に変更を加え，人工的な欠陥を生成することである．これらの人工的な欠陥を SBFL でどの程度正確に特定できるか計測することにより，プログラム全体がどの程度 SBFL に適するかを測定できる．

2.3.1　SBFL スコアの計測方法

先行研究 [5] で提案された，SBFL スコアの計測方法の概要を示す．プログラム P の SBFL スコア計測時の入力は以下の 2 つである．

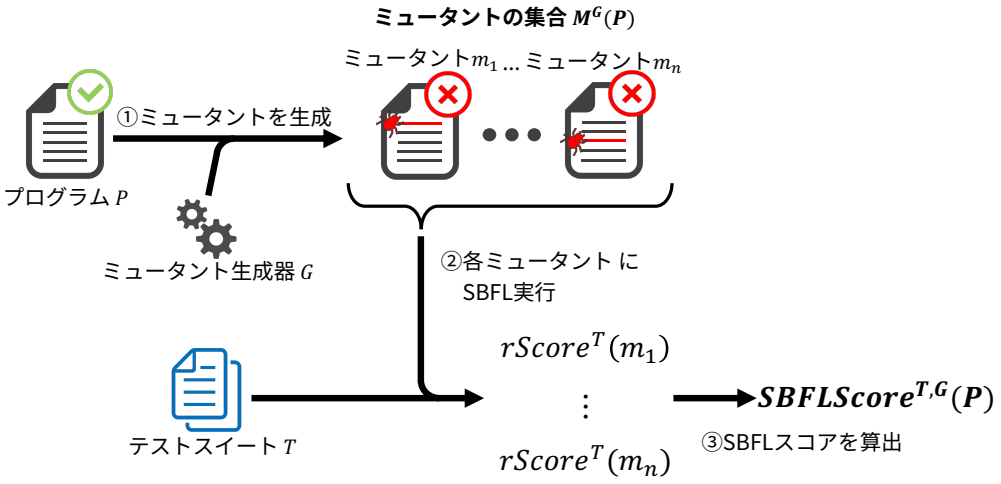

図 2　SBFL スコアの計測方法の概要

- ミュータント生成器 G
- テストスイート T

計測の大まかな流れを図 2 に示す．SBFL スコアの計測は以下の 3 つのステップで構成される．

Step1. プログラムに対してミュータントを生成

Step2. ミュータントに対して SBFL を実行

Step3. SBFL スコアを算出

以降では，各ステップの詳細を述べる．

Step1. プログラムに対してミュータントを生成

対象プログラムとテストスイートを用意する．プログラムはすべてのテストに通過することを前提とする．このプログラムに対して，ミュータント生成器を用いてミュータントを生成する．このとき，各ミュータントは元のプログラムに対して 1 箇所だけが変更されるように生成する．プログラムには変更可能な箇所が複数存在するため，ミュータントは複数生成される．ミュータント生成器 G によって生成されたミュータントの集合を $M^G(P)$ と定義する．

Step2. ミュータントに対して SBFL を実行

$M^G(P)$ に含まれる各ミュータントに対して，テストスイート T を用いて SBFL を実行し，文ごとの疑惑値を算出する．ここで，あるミュータント $m \in M^G(P)$ に含まれる各文 s について，以下を定義する．

- $susp^T(s)$：文 s の疑惑値
- $rank^T(s)$：文 s の疑惑値の順位
- $rScore^T(s)$：文 s の疑惑値の正規化順位

疑惑値の算出には Ochiai の計算式を用いる．疑惑値の順位は，疑惑値の高い順に文を並べた際に，欠陥が存在する文を発見するまでに最大で確認しなければならない文の総数とする．例えば，疑惑値 1.0 の文が 2 つ，0.8 の文が 1 つ存在する場合は，疑惑値 1.0 の文はどちらも 2 位と扱い，疑惑値 0.8 の文は 3 位とする．疑惑値の順位は，文の総数により異なる価値を持つ．例えば，10 個の文のうちの 10 位と，100 個の文のうちの 10 位とでは，後者の方が順位としての価値が高い．そこで，各文が文全体の中でどの程度上位に位置するかを表すため，順位を 0 以上 1 以下の範囲に線形に正規化する．文 s の正規化順位 $rScore^T(s)$ を以下の (2) の通り算出する．1 が最も価値が高く，0 が最も価値が低いことを表す．(2) において，$totalStatements^T$ はテストスイート T によって実行される文の数である．

$$rScore^T(s) = 1 - \frac{rank^T(s) - 1}{totalStatements^T - 1} \tag{2}$$

また，ミュータント m に含まれる欠陥の疑惑値の正規化順位を $rScore^T(m)$ と定義する．各ミュータントに含まれる欠陥は 1 箇所であるため，この値はミュータントごとに一意である．ミュータント m の欠陥を含む文を s^m_{fault} とすると，$rScore^T(m)$ は文 s^m_{fault} の疑惑値の正規化順位である．

$$rScore^T(m) = rScore^T(s^m_{fault}) \tag{3}$$

ミュータントの $rScore$ が高いほど，欠陥箇所を正確に特定できることを意味する．

Step3. SBFL スコアを算出

対象プログラムから生成された各ミュータントの $rScore$ の平均をとることにより，SBFL スコアが算出される．なお，$|M^G(P)|$ は，生成されたミュータントの総数を表す．

$$SBFLScore^{T,G}(P) = \frac{1}{|M^G(P)|} \sum_{m \in M^G(P)} rScore^T(m) \tag{4}$$

2.4 先行研究の調査方法

先行研究 [5] では，Java で実装されたプログラムを対象として，プログラム構造の違いにより SBFL スコアがどのように変化するか確認するための実験が行われた．

2.4.1 実験対象プログラム・テストスイート

先行研究 [5] では，単一のメソッドで構成されるプログラムに対して SBFL スコアを計測する実験が行われた．実験対象は，同じ入力が与えられると同じ出力を返すが，構造の異なる 5 種類のプログラムペアである．以降，同じ入力が与えられると同じ出力を返すが，構造の異なるメソッドを同機能異構造メソッドと呼ぶ．5 種類のプログラムペアは，佐々木によってリファクタリングを題材として作成された．リファクタリングには多くのパターンが存在するが，Fowler によって「条件記述の単純化」に分類されるリファクタリングパターンの中から選定した単一のメソッド内で完結する以下の 5 種類のリファクタリングが題材となった [8]．

- 変数の切り出し（「条件記述の分解」の代用）：条件式を他の変数に切り出す．
- 重複した条件記述の断片の統合：条件式のすべての分岐先に同じ処理がある場合，その処理を条件式の外側に移動する．
- 条件式の統合：同じ処理を持つ一連の条件式がある場合，それらを 1 つの条件記述にまとめる．
- 制御フラグの削除：処理の流れを制御するフラグを削除し，`break`，`continue`，`return` を利用する．
- ガード節による入れ子条件記述の置き換え：後続の処理の対象外となる条件が満たされた場合に `return` する処理を先頭に記述する．

また，各プログラムのテストスイートは佐々木によって，以下を満たすように作成された．

- すべてのミュータントはいずれかのテストに失敗する．
- 条件網羅率が 100% になる．

2.4.2 ミュータント生成器

先行研究 [5] では，表 1 に示す 11 種類のミューテーション演算子が用いられた．これらは，オープンソースのミューテーションテストツールである PIT [9] の基本ミューテーション演算子を参考に実装された．PIT は，ミューテーションテストの分野でミュータントの生成に広く用いられている [10]．

2.5 先行研究の調査結果

先行研究 [5] での調査の結果，同一条件分岐先で実行される文が少ないほど SBFL 適合性が高いことが明らかになった．また，SBFL 適合性を高めるプログラム構造の変換方法として，以下の 2 つが得られている．

```
int result = 0;
if (x > 0)
    result = -10;
else if (y > 0)
    result = -20;
else if (y>0)
    result = -30;
return result;
```

(a) 変換前

```
if (x>0)
    return -10;
if (y>0)
    return -20;
return -30;
```

(b) 変換後

図 3　SBFL 適合性を高めるために return 文を用いて条件分岐を早期終了させる例

```
int result = 0;
int tmp = 0;
if (x > 0)
    tmp = y * 2;
else
    tmp = y * 3;
result = y + tmp;
return result;
```

(a) 変換前

```
int result = 0;
int tmp = 0;
if (x > 0)
    tmp = y * 2;
    result = y + tmp;
else
    tmp = y * 3;
    result = y + tmp;
return result;
```

(b) 変換後

図 4　SBFL 適合性を高めるために文を移動する例

- return 文を用いて条件分岐を早期終了させる（図 3）
- 同一の条件分岐先の文の数が多い方から少ない方へ文を移動する（図 4）

表 1　先行研究で利用されたミューテーション演算子

ミューテーション演算子	説明	変換例	
		変換前	変換後
(CB) Conditional Boundary	関係演算子の境界を変更	`a<b`	`a<=b`
(INC) Increments	インクリメントとデクリメントを入替	`n++`	`n-`
(INV) Invert Negatives	負の数を正の数に置換	`-n`	`n`
(MA) Math	算術演算子を置換	`a+b`	`a-b`
(NC) Negate Conditionals	関係演算子を置換	`a==b`	`a!=b`
(VM) Void Method Calls	戻り値のないメソッド呼び出しを削除	`method();`	`;`
(PR) Primitive Returns	プリミティブ型の戻り値を 0 に置換	`return 5;`	`return 0;`
(ER) Empty Returns	戻り値の型に応じて空を表す値に置換	`return "str";`	`return "";`
(FR) False Returns	戻り値を false に置換	`return true;`	`return false;`
(TR) True Returns	戻り値を true に置換	`return false;`	`return true;`
(NR) Null Returns	戻り値を null に置換	`return object;`	`return null;`

3　先行研究の課題とその解決策

先行研究 [5] には，以下の 2 つの課題が存在する．

- ミューテーション演算子の種類が少ない．
- 調査対象プログラム数が少ない．

以下では，課題の詳細と，その解決策について述べる．

3.1　ミューテーション演算子の種類が少ない

先行研究で使用されたミューテーション演算子は 11 種類であった．先行研究で実験対象となったプログラムは，これらのミューテーション演算子が多くの箇所に適用できるように実装されたため，十分な量のミュータントが生成できていた．しかし，それ以外のプログラムを用いる場合，十分な量のミュータントを生成できない場合がある．この状態で計測された SBFL スコアは，プログラム全体の SBFL 適合性を表さないため，信頼性が低い．

この問題の解決のために，本研究では，新たに表 2 に示す 16 種類のミューテーション演算子を実装した．なお，新たに追加したミューテーション演算子は，既存のミューテーション演算子と同じミュータントを生成する可能性がある．例えば，プログラムに `return false;` という文がある場合，Change Boolean Lieteral と，False Return が同じミュータントを生成する．よって，同じミュータントは複数生成されないようにミュータント生成器を実装した．

表 2　本研究で新たに用いるミューテーション演算子

ミューテーション演算子	説明	変換例	
		変換前	変換後
(CS) Change String Literal	文字列リテラルを別のリテラルに変更	`"String"`	`"String1"`
(CI) Change Instanceof	`instanceof` 文の右辺を変更	`a instanceof A`	`a instanceof B`
(NMC) Nonvoid Method Calls	戻り値を持つメソッド呼び出しを `null` に変更	`a = method()`	`a = null`
(CC) Constructor Calls	コンストラクタ呼び出しを null に変更	`a = new A()`	`a = null`
(CO) Compund Operator	複合代入演算子を変更	`a += 1`	`a -= 1`
(CNL) Change Numeric Literal	数リテラルを別のリテラルに変更	`if(x<0)`	`if(x<1)`
(CBL) Change Boolean Literal	真偽値リテラルを反転	`true`	`false`
(CUO) Change Unary Operator	単項演算子を削除	`!ismethod()`	`ismethod()`
(ANO) Add Not Operator	boolean 型の変数やメソッド呼び出しに`!`を追加	`if(b)`	`if(!b)`
(MSI) More Specific If	`&&`でつながれた条件式を変更	`if(a&&b)`	`if(a) , if(b), if(a||b)`
(LSI) Less Specific If	`||`でつながれた条件式を変更	`if(a||b)`	`if(a), if(b), if(a&&b)`
(CBC) Change Break and Continue	`break` 文と `continue` 文を入替	`break;`	`continue;`
(NA) Null Assignment	代入文の右辺を `null` に変更	`Object o1 = o2;`	`Object o1 = null;`
(EA) Empty Assignment	代入文の右辺を空値に変更	`String s = a.toString();`	`String s = "";`
(PA) Primitive Assignment	プリミティブ型変数の代入文の右辺を 0 に変更	`int a = b;`	`int a = 0;`
(CTE) Change Throw Exception	throw 文の対象を変更	`Throw new A;`	`Throw new B;`

	プログラム (入力: a)	t_1	t_2	t_3
s_1:	`if (a == 0)`	✓	✓	✓
s_2:	`  return 0;`			✓
s_3:	`else if (a > 0)`	✓	✓	
s_4:	`  return 1;`	✓		
s_5:	`return -1;`		✓	✓

(a)

	プログラム (入力: a)	t_1	t_2	t_3
s'_1:	`if (a > 0)`	✓	✓	✓
s'_2:	`  return 1;`	✓		
s'_3:	`else if (a < 0)`		✓	✓
s'_4:	`  return -1;`		✓	
s'_5:	`return 0;`			✓

(b)

図 5　実行経路が異なるプログラムペアの例

	プログラム (入力: a,b)	t_1	t_2	t_3	t_4
s_1:	`if (a > 0)`	✓	✓	✓	✓
s_2:	`  return 1;`	✓			
s_3:	`if (b < 0 && a != 0)`		✓	✓	✓
s_4:	`  return -1;`		✓		
s_5:	`return 0;`			✓	✓

(a)

	プログラム (入力: a,b)	t_1	t_2	t_3	t_4
s'_1:	`if (a > 0)`	✓	✓	✓	✓
s'_2:	`  return 1;`	✓			
s'_3:	`else if (a < 0 && b < 0)`		✓	✓	✓
s'_4:	`  return -1;`		✓		
	`else`				
s'_5:	`  return 0;`			✓	✓

(b)

図 6　実行経路が同じであるプログラムペアの例

3.2　実験対象プログラム数が少ない

実験対象プログラムは 5 組のプログラムペアに含まれる 10 個のプログラムであった．よって，調査結果の一般化可能性に欠ける．また，SBFL スコアが高くなるプログラム構造を十分に調査しきれていない．

この問題の解決のために，本研究では，Java で実装された同機能メソッドを含むデータセット [11] を用いる．本データセットは，単一のメソッドからなるプログラムを 728 個含む．これらはオープンソースソフトウェアから収集された．また，メソッドは機能によって 276 組のグループに分類されている．各グループは，同機能メソッドを 2 個から 12 個含む[*1]．各プログラムのテストスイートは Evosuite[12] によって自動生成された．

なお，本研究で用いるデータセットの中には，記述方法は異なるが，テストスイートを実行した際の実行経路が同じになるグループが含まれる可能性がある．「実行経路が同じ」とは，2 つのプログラムが以下の 2 つの条件を満たすことを指す．ここで，s_i は 1 つ目のプログラムの上から i 番目の文を，s'_i は 2 つ目のプログラムの上から i 番目の文を表す．

- テストスイートによって実行される行の総数 m が同じである．
- テストスイートに含まれるすべてのテストにおいて，$1 \leq i \leq m$ について，文 s_i における実行の有無が，文 s'_i における実行の有無と等しい．

[*1] 実際には，同機能でないメソッドが存在する．

実行経路が異なるプログラムペアの例を図 5 に，実行経路が同じであるプログラムペアの例を図 6 に示す．2 つのプログラムの実行経路が同じになる場合，適用されるミューテーション演算子の違いのみによって，SBFL スコアに差が生じてしまう．このようなプログラムは実験対象として適切でないと考えたため，実験対象から除外する必要がある．よって，本研究においては，プログラム構造を「テストスイートを実行した際の実行経路」と定義する．記述方法が異なっても，テストスイートを実行した際の実行経路が同じ場合には，プログラム構造は同じであるとみなす．

4 Research Questions

本研究では，SBFL 適合性が高いプログラム構造をするために，3 つの Research Question(RQ) を設定した．まず，RQ 1 では，SBFL スコアの信頼性の向上を検証する．次に，RQ 2 では，考察対象プログラム数の増加を検証する．その後，本研究の主題である，SBFL 適合性が高いプログラム構造について RQ 3 を設定して調査する．

RQ 1：ミューテーション演算子の追加により SBFL スコアの信頼性は高まったか？

RQ 1 では，先行研究の課題であった SBFL スコアの信頼性の低さを解決するために，新たなミューテーション演算子を実装する．ミューテーション演算子の追加前と追加後のそれぞれについて，実際のプログラムで生成されたミュータントの個数を比較し，ミュータントの増加数を検証する．このとき，プログラムの規模を考慮する必要がある．なぜなら，プログラムの規模が大きくなると，ミュータントの個数は増加する傾向にあるからである．規模に対するミュータントの増加数を評価するための指標として，IMPL (Increased Mutants Per LOC) を求める．IMPL は式 (5) で定義される．ここで，論理 LOC とは，プログラムのうち，空行や括弧のみの行，コメントのみの行を除いた行数を指す．IMPL の値が大きいほど，SBFL スコアの精度が高まったことを表す．

$$IMPL = \frac{\text{ミュータントの増加数}}{\text{論理 LOC}} \tag{5}$$

RQ 2 :SBFL スコアに変化が生じたグループ数はいくつか？

RQ 2 では，考察対象プログラム数の増加を検証する．データセットの利用により，実験対象プログラム数は増加した．しかし，プログラム構造が異なる場合でも，SBFL スコアが同じになる場合が考えられる．よって，プログラム構造が異なるグループのうち，SBFL スコアに差が生じるグループの数を確認し，考察対象プログラム数の増加を検証する．また，SBFL スコアが同じになる要因を考察する．

RQ 3：SBFL スコアが高くなるプログラム構造とは何か？

RQ 3 では，すべての実験対象プログラムの SBFL スコアを計測する．計測結果やテストの実行経路情報をもとに，SBFL スコアが高いプログラム構造を調査し，その要因を考察する．さらに，調査の結果得られた SBFL 適合性が高いプログラム構造をもとに，SBFL 適合性を高めるプログラム構造の変換方法を提案する．

5　実験

実験では，どのようなプログラム構造が SBFL に適するか明らかにする．また，得られた結果をもとに，SBFL 適合性を高めるプログラムのを提案する．

5.1　実験対象プログラム・テストスイート

実験対象プログラムは，3.2 節で示したデータセットに含まれる 728 個のプログラムのうち，以下の 3 つの条件を満たす 133 組のグループに含まれる 365 個のプログラムである．

- グループ内のプログラムをテストスイートに通したときの条件網羅率が 100% になる．
- グループ内のプログラムが同機能である．
- グループ内のプログラム間で，プログラム構造が互いに異なる．

条件の確認は以下の 2 つのステップで構成される．

Step1. テストスイートを手動で修正する．
Step2. プログラムと実行経路情報を目視で確認し，条件を満たさないプログラムを除外する．

以下，これらの手順の詳細を述べる．

Step1. テストスイートを手動で修正する．

各プログラムのテストスイートは Evosuite[12] によって自動生成された．自動生成されたテストスイートでは，条件網羅率が 100% にならない場合がある．よって，テストスイートを手動で修正し，条件網羅率が 100% になるようにした．

Step2. プログラムと実行経路情報を目視で確認し，条件を満たさないプログラムを除外する

条件の確認のために，すべてのプログラムをテストスイートに通し，網羅率や等価性，実行経路を確認した．その結果，合計で 143 組のグループと 363 個のプログラムが実験対象から除外された．除外された要因とそれに当てはまるプログラムの数を表 3 に示す．

5.2　実験方法

実験は以下の 2 つのステップで構成される．

Step1. 実験対象プログラムの SBFL スコアを計測する．

Step2. 計測結果をもとに，SBFL スコアが高くなる要因を考察する．

以下では，これらの詳細を述べる．

Step1. 実験対象プログラムの **SBFL** スコアを計測する

すべての実験対象プログラムについて，計測ツールを用いて SBFL スコアを計測する．計測ツールは SBFL スコアを計測するために，自動欠陥修正ツールである kGenProg[13] を用いて実行経路情報を取得する．kGenProg は GenProg [14] という自動欠陥修正ツールを Java で実装したツールである．kGenProg はプラグインとして JaCoCo [15] を呼び出し，実行経路情報を得ている．各ミュータントに対して SBFL を実行した際の実行経路情報はファイルに保存され，後から確認できる．

また，計測ツール内のミュータント生成器に，2.4.2 節で示した 11 種類のミューテーション演算子と，3.1 節で示した 16 種類のミューテーション演算子を実装した．さらに，使用するミューテーション演算子を切り替えられるようにした．これにより，ミューテーション演算子の追加前後におけるミュータント数の変化を記録できる．

Step2. 計測結果をもとに，**SBFL** スコアが高くなる要因を考察する

各ミュータントに対して SBFL を実行した際の実行経路情報を目視で確認し，SBFL スコアが高くなる要因を考察する．また，先行研究の結果に反する事例がないか確認する．

表 3　条件にあてはまらないプログラムの数

条件	プログラム数
条件網羅率が 100% にならない	18
グループ内のプログラムが等価でない	90
グループ内のプログラムの実行経路が同じである	255

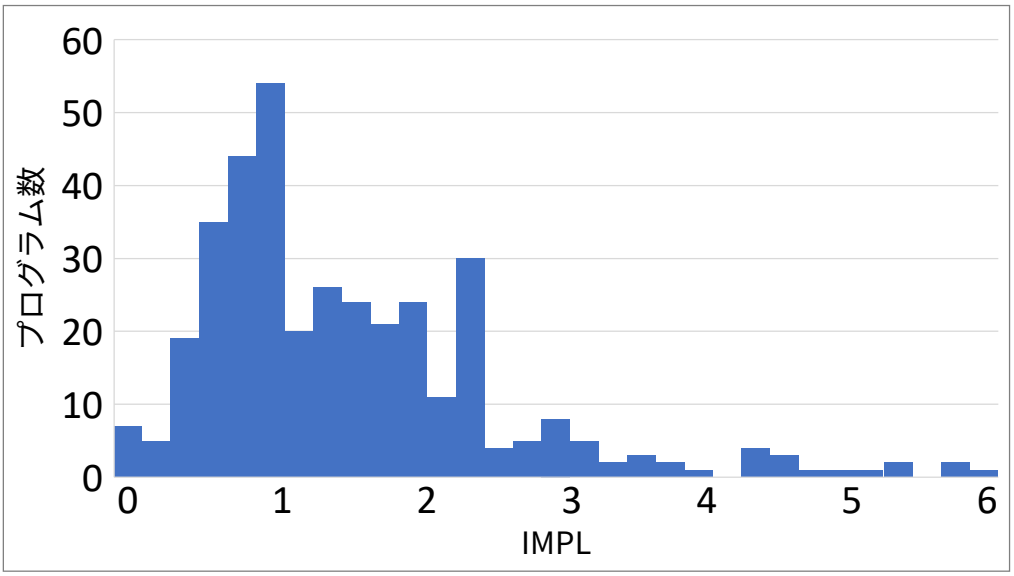

図 7　各プログラムの IMPL の分布

6　実験結果と考察

RQ 1：ミューテーション演算子の追加により SBFL スコアの信頼性は高まったか？

各プログラムについて IMPL を計測した結果を図 7 に示す．IMPL が 0 より大きいプログラムは 365 個中 358 個であった．よって，ほとんどのプログラムにおいて，SBFL スコアの信頼性は向上した．

RQ 2 :SBFL スコアに変化が生じたグループ数はいくつか？

実験対象である 133 組のグループのうち，120 組のグループで SBFL スコアに変化が見られた．よって，考察対象プログラム数は増加した．

SBFL スコアに変化が生じないグループは 11 組存在した．変化が生じない理由は，以下の 2 つであった．

- グループ内のいずれのプログラムにも分岐が全く存在せず，例外も発生しない．
- 実行経路が入れ替わっているのみで，実行される文の数は同じである．

RQ 3：SBFL スコアが高くなるプログラム構造とは何か？

SBFL スコアが高くなる要因の分類

SBFL スコアが高くなる要因として，新たに以下の 3 つが特定できた．

```
long max = strings.stream().mapToLong(String::length).max().orElse(Long.MIN_VALUE);
return max;
```

(a)

```
long max = Long.MIN_VALUE;
for (String string : strings) {
  long length = string.length();
  if (length > max)
    max = length;
  }
return max;
```

(b)

図 8　if 文や for 文の有無を示すプログラムペアの例

- 分岐がない処理の代わりに if 文や for 文を用いている
- early return[16] のための if 文を追加している
- if 文の条件式に論理演算子を用いず，別の文に分けている

また，先行研究で発見された要因である「同一条件分岐先で実行される文が少ない」に当てはまるグループも存在した．SBFL スコアが高くなる要因とそれに当てはまるグループの数を表 4 に示す．

以下では，新たに発見した要因について，具体例を用いて説明する．

分岐がない処理の代わりに if 文や for 文を用いている

図 8 の (b) のプログラムでは，if 文と for 文が，リストに含まれる各要素の処理に用いられている．よって，テストを実行した際の実行経路に差が生じるため，SBFL スコアが 0 よりも大きくなる．一方，(a) のプログラムでは，処理にストリームが用いられており，if 文と for 文が存在しない．if 文や for 文がプログラム中にない場合，各行の疑惑値に差が生じないため，各ミュータントの $rScore$ が 0 になり，SBFL スコアも 0 になってしまう．よって，分岐がない処理の代わりに if 文や for 文を用

表 4　SBFL スコアが高くなる要因とそれに当てはまるグループの数

要因	グループ数
分岐がない処理の代わりに if 文や for 文を用いている	21
early return のための if 文を追加している	15
if 文の条件式に論理演算子を用いず，別の文に分けている	12
同一条件分岐先で実行される文が少ない	62
不明	10

```
1 int sum = 0;
2 for (int i = pValues.length - 1; i >= 0; i++)
3     sum += pValues[i];
4 return sum;
```

(a)

```
1 if (add.length < 1)
2   return 0;
3 int ret = 0;
4 for (int i = 0; i < add.length; i++)
5   ret += add[i];
6 return ret;
```

(b)

図 9　if 文による early return の有無を示すプログラムの例

```
1 if (i < start || i > end) {
2     throw new IllegalArgumentException();
3 }
4 return i;
```

(a)

```
1 if (i < start) {
2     throw new IllegalArgumentException();
3 } else if (i > end) {
4     throw new IllegalArgumentException();
5 }
6 return i;
```

(b)

図 10　if 文の条件式における論理演算子の有無を示すプログラムペアの例

いることで，SBFL スコアは向上する．

early return のための if 文を追加している

プログラムの入力によっては，長い処理を行う前に出力がわかる場合がある．例えば，図 9 の (a) では，配列の内容の処理に for 文が用いられている．一方，(b) では if 文を追加することにより，配列の長さが 0 である場合には for 文による処理を行わずに 0 を返す．if 文がない場合，テストスイートの実行経路に差が生じにくい．一方で，if 文による early return を追加することにより，テストスイートの実行経路を分散させることができる．これにより，その後に続く文に欠陥がある場合に疑惑値を上昇させられるため，SBFL スコアが向上する．

if 文の条件式に論理演算子を用いず，別の文に分けている

図 10 の (a) のプログラムでは，条件式を||でつなぎ，1 つの if 文を用いている．一方，(b) のプログラムでは，条件式を分け，2 つの if 文を用いている．条件式を||でつないだ場合，条件式に欠陥がある場合でも，その行を成功テストが多く通過する．成功テストが多く通過すると，その行の疑惑値が低下する．欠陥箇所の疑惑値が低下すると，他の文の疑惑値の順位が上昇する．その結果，$rScore$ が低下し，SBFL スコアも低下する．反対に，条件式を別の if 文に分けると，SBFL スコアは向上する．

```
1 int len = list.length;
2 int[] arr = new int[len];
```

(a) 変換前

```
1 int[] arr = new int[list.length];
```

(b) 変換後

図 11　変数のインライン化を行う例

```
1 int i = 0;
2 for (int elem : list) {
3     array[i] = elem;
4     i++;
5 }
```

(a) 変換前

```
1 for (int i = 0; i < list.size(); i++) {
2     array[i] = list.get(i);
3 }
```

(b) 変換後

図 12　配列の処理に拡張 `for` 文ではなく `for` 文を用いる例

先行研究の結果の検証

先行研究では，同一条件分岐先で実行される文が少ないほど SBFL スコアが高くなると結論付けられている．しかし，`if` 文の条件式に論理演算子を用いている場合，この結果に反することがわかった．

例として，図 10 を用いて説明する．図 10 中の (a) と (b) において，各条件分岐先で実行される文の数を表 5 に示す．表 5 から分かる通り，2 つの条件分岐先で (b) で実行される文の数のほうが多い．しかし，(b) のほうが SBFL スコアが高いため，先行研究の結果と反する．

SBFL スコアを向上させるプログラム構造の変換方法

以上の結果より，SBFL スコアを向上させるプログラム構造の変換方法として以下の 3 つが得られた．

- 分岐を用いない処理を `if` 文や `for` 文を用いて書き換える．
- `if` 文による early return を追加する．
- 論理演算子でつながれた条件式を別の `if` 文に分割する．

表 5　図 10 のプログラム中の各条件分岐先で実行される文数の比較

条件分岐先	(a) で実行される文の数	(b) で実行される文の数
`i < start`	2	2
`i > end`	2	3
`start <= i && i <= end`	2	3

また，同一条件分岐先で実行される文の数を少なくするためのプログラムの変形方法として，新たに以下の 2 つが得られた．

- 変数のインライン化を行う．(図 11)
- 配列のインデックスを扱う場合に，拡張 `for` 文ではなく `for` 文を用いる．(図 12)

7　妥当性への脅威

SBFL スコアの計測結果は，テストスイート及びミュータント生成器に影響を受ける．よって，異なるテストスイート及びミュータント生成器を使用すると，SBFL スコアの値が変化し，異なる結果が現れる可能性がある．

また，本研究とは異なるプログラムを実験対象とすると，今回の実験で得られた結果を否定する事例が現れる可能性がある．

8　おわりに

本研究では，先行研究の課題であるミューテーション演算子の種類の少なさと実験対象プログラム数の少なさを解決した．また，SBFL スコアの計測実験を行い，SBFL 適合性が高いプログラム構造を調査した．調査の結果，新たに 3 つの SBFL 適合性が高くなる要因が得られ，そのうち 1 つが先行研究の結果を否定することを確認した．さらに，SBFL 適合性を高めるプログラム構造の変換方法を提案した．

今後の課題としては以下が考えられる．

複数のメソッドからなるプログラムを用いた SBFL スコアの計測　複数のメソッドからなるプログラムと，複数のメソッドにまたがるミューテーション演算子を定義して SBFL スコアを計測し，SBFL 適合性が高いプログラム構造を新たに発見する．

SBFL 適合性を高めるようなプログラムの自動変換　SBFL 適合性を高める変換方法に基づいてプログラムを自動変換し，SBFL スコアの向上を確かめる．

SBFL 適合性と他の品質特性との関連性調査　SBFL 適合性が高くなる一方で，プログラムの保守性が低下する場合がある．よって，SBFL 適合性と他の品質特性との関連を明らかにすることは今後の重要な課題である．

謝辞

本論文の執筆においては，多くの方々にご協力いただきました．

肥後芳樹教授には，本研究の全体において，熱心な指導をしていただきました．テーマ選定から毎週のミーティング，添削に至るまで，たくさんの協力をいただきました．

楠本真二教授には，研究に関して有益なアドバイスをいただきました．特に，中間報告におけるアドバイスは，研究の要点を整理するのに役立ちました．

柗本真佑助教には，研究の方向性や，プレゼンに関する適切なアドバイスをいただきました．特に，中間報告での鋭い指摘は，スライドを改善するのみならず，研究の全体像を整理する助けにもなりました．

事務補佐員の橋本美砂子さんは，研究室運営のみならず，出張の手続きや差し入れに至るまで，様々な面での協力をいただきました．

楠本研究室の先輩方には，研究や発表に関するアドバイスを頂いたり，文章の添削をしていただきました．また，研究室での何気ない雑談は，充実した研究室生活を送る助けになりました．

楠本研究室の同期の皆様とは，卒論の執筆や大学院入試の勉強を共にすることで，より一層研究に励むことができました．

家族には，決して裕福とはいえない家庭ながらも，大学院進学に至るまで金銭面での支援をしていただきました．そのおかげで，お金や生活の心配をすることなく研究活動を行えました．

最後に，本研究を支えてくださった皆様に改めて感謝を申し上げます．

参考文献

[1] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[2] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. *Quantify the time and cost saved using reversible debuggers.* Technical report, Cambridge Judge Business School, 2012.

[3] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

[4] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the accuracy of fault localization techniques. In *Proc. International Conference on Automated Software Engineering*, pp. 76–87, 2009.

[5] 佐々木唯, 肥後芳樹, 柗本真佑, 楠本真二. プログラムに対する欠陥限局の適合性計測. 情報処理学会論文誌, Vol. 62, No. 4, pp. 1029–1038, 2021.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proc. Pacific Rim International Symposium on Dependable Computing*, 2006.

[7] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[8] Fowler Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[9] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proc. International Symposium on Software Testing and Analysis*, pp. 449–452, 2016.

[10] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proc. International Symposium on Foundations of Software Engineering*, pp. 52–63, 2014.

[11] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. Constructing dataset of functionally equivalent java methods. In *Proc. the International Conference on Mining Software Repositories*, 2022.

[12] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proc. Symposium and the European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[13] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kgenprog: A high-performance, high-extensibility and high-portability apr system. In *Proc. the Asia-Pacific Software Engineering Conference*, pp. 697–698, 2018.

[14] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[15] Hoffmann M.R. Jacoco java code coverage library. `https://www.eclemma.org/jacoco/`. [Online; accessed 2-February-2023].

[16] Mauricio A. Saca. Refactoring improving the design of existing code. In *Proc. IEEE Central America and Panama Convention*, pp. 1–3, 2017.