社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

欠陥限局に適したテストスイートに関する考察

九間　哲士† 　肥後　芳樹† 　柗本　真佑† 　楠本　真二†

† 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
E-mail: †{t-kuma,higo,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発においてデバッグに要するコストを削減することを目的として，欠陥限局に関する研究が盛んに行われている．欠陥限局とは，プログラム中に存在する欠陥箇所を推測する技術である．既存手法として，プログラムとそのテストスイートを入力とし，各テストケースの成否と実行経路の情報を用いて欠陥限局を行う手法が提案されている．この手法には，入力として与えられるテストスイートによって欠陥限局の精度が大きく左右されるという特徴がある．テストスイートから得られる実行経路の情報が十分でない場合，欠陥限局の精度は低くなる．そこで本研究では，欠陥限局の精度を向上させるため，既存のテストスイートの経路網羅を高めるテストケースを生成する手法を提案する．提案手法は，欠陥を含むプログラムに対するテストケースを生成し，その中から既存のテストスイートの経路網羅を高めるテストケースのみを選択する．経路網羅とは，プログラムの全ての実行経路のうち，テストで実行された実行経路の割合である．経路網羅では，実行経路を文の実行順序や実行回数を考慮する文の列として考えるため，ループを含むプログラムでは実行経路の総数が無数になる．そのため，本研究では実行経路を文の列ではなく，文の実行順序や実行回数を考慮しない文の集合として考えることで実行経路の総数を有限に捉える．評価実験として，提案手法で生成したテストケースを既存のテストスイートに追加して欠陥限局を行った．その結果，70.6%の欠陥箇所において，既存のテストケースのみを用いる場合よりも提案手法により生成したテストケースを追加した方が限局の精度が向上した．

キーワード　欠陥限局，経路網羅, テスト生成

1. ま え が き

ソフトウェア開発において，デバッグは多大な労力を要する作業である．ソフトウェア開発コストのうち，半数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，デバッグ支援に関する研究が盛んに行われている．

デバッグ支援の 1 つに欠陥限局と呼ばれる技術がある．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り，これまで様々な欠陥限局手法が提案されている [3]〜[5]．その中でも，実行経路情報に基づいて欠陥限局を行う Spectrum-Based Fault Localization(以降，SBFL) は近年最も盛んに研究されている手法の 1 つである [6]．SBFL は失敗テストケースで実行された文は欠陥箇所である可能性が高く，成功テストケースで実行された文は欠陥箇所である可能性が低いという考えに基づいて欠陥限局を行う．SBFL は欠陥を含むプログラムとそのテストスイートを入力として受け取り，各テストケースの成否と実行経路の情報を用いて欠陥箇所を行う．SBFL はテストスイートの実行によって得られる情報のみを用いて欠陥限局を行うため，その精度は入力として与えられるテストスイートに依存するという特徴がある．

図 1 は，Apache Commons Lang(以降, Lang) に含まれるプログラムの一部である(注1)．図 1(a) に示されるプログラムは 4 行目が欠陥箇所である．4 行目は `return a;` となっているが，正しくは `return b;` である．このプログラムに対し Lang に含まれる全てのテストケースを実行すると図 1(b) に示されるテストケース t1 が失敗する．この欠陥を含むプログラムと Lang のテストスイートを入力として SBFL を適用すると t1 によって実行された 3，4，5，6，8，12，13，14，15 行目の文が欠陥箇所である可能性が高いと判定される．すなわち，プログラム中の全ての文が欠陥箇所である可能性が高いことを示し，SBFL は開発者が欠陥箇所を特定する助けになっていない．このように，既存のテストスイートだけを利用すると欠陥箇所をほとんど推測できないことがある．

そこで著者らは既存のテストスイートにテストケースを追加すれば欠陥箇所をより正確に推測できるのではないかと考えた．図 1(c) に示す a1，a2，a3 のようなテストケースを既存のテストスイートに追加する．テストケースを追加した既存のテストスイートを用いて SBFL を適用すると，4 行目だけが欠陥箇所である可能性が高いと判定される．このとき，a2 と a3 が追加されることで欠陥箇所はより絞り込まれているが，a1 が追加されても欠陥箇所は 1 つも絞り込まれていない．これは，a1 の実行経

(注1)：説明を簡略化するため，一部のメソッドや文を省略，変更している．

サンプルプログラム		テストケース			
1	public class IEEE754rUtils {	t1	a1	a2	a3
2	float max(float a, float b) {				
3	if(Float.isNaN(a))	●	●	●	●
4	return a; //return b;	●	●		
5	else if(Float.isNaN(b))	●	●	●	●
6	return a;	●	●		●
7	else				
8	return Math.max(a, b);	●	●	●	
9	}				
10	float max(float[] array) {				
11	...				
12	float max = array[0];	●	●	●	●
13	for(int j = 1; j < array.length; j++)	●	●	●	●
14	max = max(array[j], max);	●	●	●	●
15	return max;	●	●	●	●
16	}				
17	}				
テストケースの成否(成功=P,失敗=F)		F	F	P	P
欠陥箇所を特定するまでに確認する最大の行数		9	9	2	1

(a) サンプルプログラムとテストケースの実行情報

既存のテストケースt1

```
1  @Test
2  void test_t1() {
3    float[] aF = new float[] { 1.2f, Float.NaN, 3.7f,
4        27.0f, 42.0f, Float.NaN };
5    assertEquals(42.0f, IEEE754rUtils.max(aF));
6    float[] bF = new float[] { Float.NaN, 1.2f, Float.NaN,
7        3.7f, 27.0f, 42.0f, Float.NaN };
8    assertEquals(42.0f, IEEE754rUtils.max(bF));
9  }
```

(b) 既存のテストケース

追加するテストケースa1, a2, a3

```
1  @Test
2  void test_a1() {
3    float[] aF = new float[] {1.2f, 3.7f, Float.NaN,0.5f};
4    assertEquals(1.2f, IEEE754rUtils.max(aF));
5  }
6  @Test
7  void test_a2() {
8    float[] aF = new float[] {1.2f, 3.7f};
9    assertEquals(3.7f, IEEE754rUtils.max(aF));
10 }
11 @Test
12 void test_a3() {
13   float[] aF = new float[] {1.2f, Float.NaN};
14   assertEquals(1.2f, IEEE754rUtils.max(aF));
15 }
```

(c) 追加するテストケース

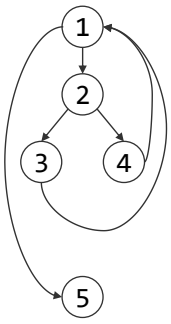
図 1: テストケースの追加により欠陥箇所を特定しやすくなる例

路が t1 と同じ実行経路であることが原因である．SBFL の精度を向上させるためには，a2，a3 のような既存のテストケースとは異なる実行経路のテストケースを追加することが重要である．また，欠陥限局は自動プログラム修正にも利用される．自動プログラム修正では，欠陥限局の結果からプログラムの修正箇所が決定される．自動プログラム修正の手法の中には，プログラムの修正が完了するまでに何度も欠陥限局を行う手法が存在する [7]．そのような手法では，欠陥限局に要する時間が増加すると，プログラムの修正を完了するまでの時間が大幅に増加する．テストケースの追加によって欠陥限局に要する時間を必要以上に増加させないためには，a1 のようなテストケースを追加しないことが重要である．

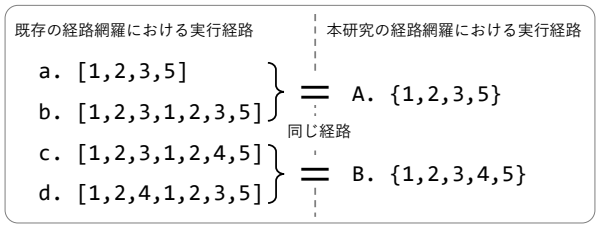
そこで本研究では，SBFL の精度を向上させるため，既存のテストスイートの経路網羅を高めるテストケースを生成する手法を提案する．提案手法は，欠陥を含むプログラムに対するテストケースを生成し，その中から既存のテストスイートの経路網羅を高めるテストケースのみを選択する．経路網羅とは，プログラムの全ての実行経路のうち，テストで実行された実行経路の割合で

```
1 while(…){
2   if(…)
3     a++;
    else
4     a--;
  }
5 return a;
```

(a) サンプルプログラム



(b) 制御フローグラフ



(c) 実行経路の違い

図 2: 既存の経路網羅における実行経路と本研究の経路網羅における実行経路の比較

ある．評価実験として，提案手法で生成したテストケースを既存のテストスイートに追加して SBFL を適用した結果と既存のテストスイートのみで SBFL を適用した結果を比較した．その結果，70.6%の欠陥箇所において，既存のテストスイートのみを用いる場合よりも提案手法によるテストケースを追加した方が SBFL の精度が向上した．

2. 準　　備

2.1 実行経路情報に基づく欠陥限局

欠陥限局とは，プログラムの欠陥箇所を推測する技術である．欠陥限局の手法の 1 つに実行経路情報に基づく欠陥限局手法 (SBFL(注2)) がある．SBFL は失敗テストケースで実行された文ほど欠陥箇所である可能性が高いという考えに基づいて欠陥限局を行う．SBFL は，欠陥を含むプログラムとそのテストスイートを入力として受け取り，各テストケースの実行経路情報と成否情報を収集する．実行経路情報とは，各テストケースでプログラム中のどの文が実行されたかを表す情報である．各テストケースの実行経路情報と成否情報を基にプログラムの各文に対し Suspiciousness (以降，疑惑値) を出力する．疑惑値とは，欠陥箇所である可能性の高さを表す数値である．疑惑値の計算手法はこれまでに数多く提案されている．Abreu らは，7 つの疑惑値の計算手法を比較し，Ochiai [8] が優れた手法であると結論付けている [9]．そのため，本研究でも疑惑値の計算には Ochiai を用いる．Ochiai の計算式は式 (1) で表される．

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

式中の各変数は以下の意味を表す．

(注2)：Spectrum-Based Fault Localization

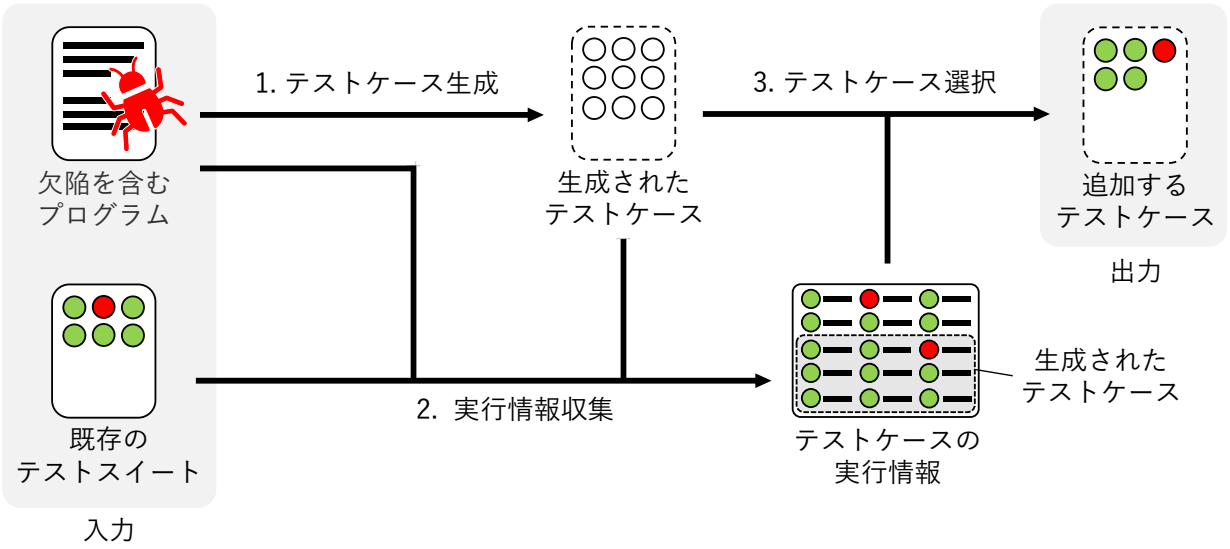


図 3: 提案手法の概要

s：疑惑値計算対象の文
$totalfail$：失敗テストの総数
$fail(s)$：s を実行した失敗テストの数
$pass(s)$：s を実行した成功テストの数

2.2 経路網羅

経路網羅は，プログラムの全ての実行経路のうちテストで実行された実行経路の割合である [10]．既存の経路網羅における実行経路とは，プログラムの開始から終了までに実行された文の列である．列は要素の出現回数や出現順序を考慮する．そのため，以下の実行経路は異なる実行経路とみなされる．

- 実行された命令は同じであるがループの回数が異なる実行経路
- 実行された命令は同じであるが実行された順序が異なる実行経路

例として図 2(a) のプログラムを考える．このプログラムの制御フローグラフは図 2(b) で表される．制御フローグラフから既存の経路網羅における実行経路を考えると，図 2(c) の a から d の実行経路が例として挙げられる．a，b は実行された命令は同じであるがループの回数が異なる実行経路であり，c，d は実行された命令は同じであるが実行された順序が異なる実行経路である．図 2(a) のプログラムはループを含むため a から d のような実行経路は無数に存在する．このようにループを含むプログラムでは実行経路の総数は無数になる．

そこで本研究では，実行経路の総数を有限に捉えるため実行経路をプログラムの開始から終了までに実行された文の集合として考える．実行経路を文の集合とすることで以下の 2 点を考慮せずに実行経路の総数を算出できる．

- 文の実行回数
- 文の実行順序

これにより，実行された命令は同じであるがループの回数が異なる実行経路や実行された命令は同じであるが実行された順序が異なる実行経路はそれぞれ同一の実行経路とみなされる．そのため，本研究の経路網羅における実行経路の総数は高々 $2^n(n = $ プログラム中の分岐の数) となる．表 1 に既存の経路網羅と本研究の経路網羅の相違点をまとめる．

3. 提案手法

本研究では，SBFL の精度を向上させるため，既存のテストスイートの経路網羅を高めるテストケースを生成する手法を提案する． テストスイートの経路網羅を高めることで，既存のテストケースでは実行されていない実行経路の情報が取得できるようになる．その結果，各文に対する疑惑値をより詳細に計算できるようになり，SBFL の精度が向上する．提案手法の概要を図 3 に示す．提案手法の入力は欠陥を含む Java プログラムとそのプログラムに対する既存のテストスイートである．出力は既存のテストスイートに追加するテストケースである．提案手法は以下の 3 つのステップで構成される．

ステップ 1： 欠陥を含むプログラムに対するテストケースの生成

ステップ 2： 既存のテストケースおよび生成されたテストケースの実行情報の解析

ステップ 3： 既存のテストスイートに追加するテストケースの選択

以降，各ステップについて説明する．

3.1 ステップ 1: テストケースの生成

テストケース自動生成ツールを用いて既存のテストスイートに追加する候補となるテストケースを生成する．テストケース自動生成ツールを用いる理由は，テストケースの生成にかかるコ

表 1: 既存の経路網羅と本研究の経路網羅の比較

	既存の経路網羅	本研究の経路網羅
ループ回数	考慮する	考慮しない
実行順序	考慮する	考慮しない
実行経路	実行された文の列	実行された文の集合
実行経路の総数	無数	$2^n(n = $ 分岐の数)

ストを削減するためである．既存のテストスイートの経路網羅を高めるテストケースを人間が生成するためには対象のプログラムの処理を理解する必要があり，大きな労力を要する．

テストケース自動生成ツールでテストケースを生成する方法は 2 つ考えられる．

- 入力値のみツールで生成し期待値を手動で入力する．
- 入力値と期待値の両方をツールで生成する．

前者は，欠陥を含むプログラムをテストケース自動生成ツールに入力してテストケースを生成できるが，手作業で期待値を入力する分テストケースの生成にコストがかかる．後者はテストケースの生成を全て自動で行えるが，欠陥のないプログラムをテストケース自動生成ツールに入力する必要がある．欠陥のないプログラムが必要な理由は，テストケース自動生成ツールでは欠陥を含むプログラムから正しい期待値を生成できないためである．

本研究では，テストケースの生成に要するコストを削減するため入力値と期待値の両方をテストケース自動生成ツールで生成する．テストケース自動生成ツールに入力する欠陥のないプログラムとして欠陥が発生する直前のリビジョンのプログラムを利用する．

3.2 ステップ 2: 実行情報の収集

欠陥を含むプログラムに対し，既存のテストケースと生成したテストケースを全て実行する．実行時に各テストケースの実行情報を収集する．実行情報には以下の 2 つの情報が含まれる．

- 実行経路情報
- 成否情報

3.3 ステップ 3: テストケースの選択

ステップ 2 で収集した実行情報を基に，ステップ 1 で生成されたテストケースの中から既存のテストスイートの経路網羅を高めるテストケースのみを選択する．これにより経路網羅を高めないテストケースを削除する．冗長なテストケースを削除することでテストスイートの肥大化を防ぎ，実行時間が必要以上に増加しないようにする．

生成されたテストケースから追加するテストケースを選択するために必要な条件は 3 つある．

（1） 実行時に既存の失敗テストケースで呼び出されたメソッドのいずれかを呼び出すテストケースであること

（2） 既存のテストケースや既に追加するテストケースとして選ばれたテストケースとは異なる実行経路のテストケースであること

（3） 失敗テストケースであること

(1) の条件を満たすテストケースのうち，(2)，(3) のいずれかの条件を満たすテストケースを既存のテストスイートに追加するテストケースとする．

(1) の条件を設ける理由は，(1) の条件を満たさないテストケースを追加しても SBFL の精度が全く変化しないからである．失敗テストケースで実行されていないメソッドに含まれる全ての文の疑惑値は 0 である．すなわち，テストケースの実行時に既存の失敗テストケースで呼び出されたメソッドがいずれも呼び出されないことは疑惑値が 0 の文だけが実行されることを意味する．疑惑値が 0 の文だけを実行するテストケースを追加しても実行された文の疑惑値はそれ以上変化しない．(2) の条件を設ける理由は，経路網羅を高めるためである．(3) の条件を設ける理由は，Kucuk らの研究 [11] で失敗テストケースが多いほど SBFL の精度が高い傾向があると結論付けられているからである．

サンプルプログラム		テストケース					
1	`public class IEEE754rUtils {`	t1	c1	c2	c3	c4	c5
2	`  float max(float a, float b) {`						
3	`    if(Float.isNaN(a))`	●		●	●	●	●
4	`      return a; //return b;`	●		●		●	
5	`    else if(Float.isNaN(b))`	●		●	●	●	●
6	`      return a;`	●		●		●	
7	`    else`						
8	`      return Math.max(a, b);`	●		●	●		●
9	`  }`						
10	`  float max(float[] array) {`						
11	`    ...`						
12	`    float max = array[0];`		●		●	●	●
13	`    for(int j = 1; j < array.length; j++)`		●		●	●	●
14	`      max = max(array[j], max);`				●	●	●
15	`    return max;`		●		●	●	●
16	`  }`						
17	`}`						
テストケースの成否(成功=P,失敗=F)		F	P	F	P	F	P
追加するか否か(追加する=1,追加しない=0)		-	0	1	1	1	0

図 4: テストケース選択の例

図 4 はテストケースの選択の例である(注3)．t1 は既存のテストケースであり，c1 から c5 はステップ 1 で生成されたテストケースである．c1 から順に上記の条件を満たすか確認すると結果は以下のようになる．

c1： (1) の条件を満たさない．

c2： (1)，(3) の条件を満たす．

c3： (1)，(2) の条件を満たす．

c4： (1)，(2)，(3) の条件を満たす．

c5： (1)，(2) の条件を満たすが，c3 と同じ実行経路である．

よって，図 4 の例では，c2，c3，c4 が既存のテストスイートに追加されるテストケースとなる．

4. 実　　験

4.1 実験概要

提案手法の評価実験として，OSS を対象に実験を行った．既存のテストスイートに提案手法で生成したテストケースを追加して SBFL を適用し，既存のテストスイートのみで SBFL を適用した場合と比較して欠陥限局の精度がどのように変化するか確認した．

4.2 実験対象

本実験では，Defects4J [12] に含まれる Apache Commons Math(以降，Math) を実験対象とする．Defects4J は，OSS プロジェクトの開発中に発生した欠陥の情報をまとめたデータセットである．欠陥の情報には，プログラムのどこが欠陥箇所で，どのテストケースに失敗したかなどの情報が含まれる．本実験では，Math に含まれる 90 個の欠陥の情報を用いる．90 個の欠陥の情報には，230 個の欠陥箇所が存在する．Math を実験対象とした理由は，欠陥限局の論文においてベンチマークとして広く利用されているためである [13]．

4.3 実験設定

本実験では，テストケースの生成に EvoSuite [14] を利用した．

(注3)：対象のプログラムは図 1(a) と同じであるが，説明を簡略化するためテストケースを変更している

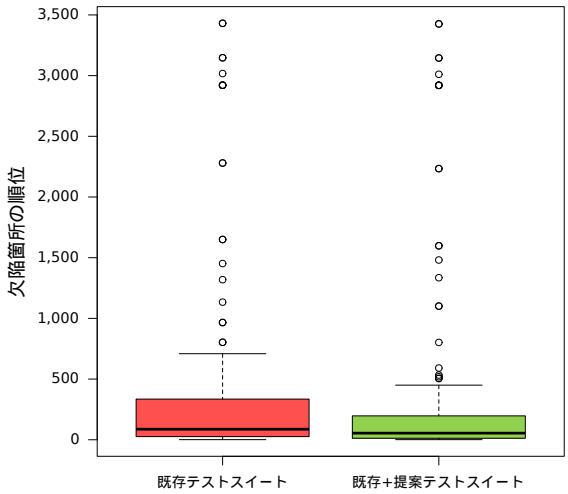


図 5: 欠陥箇所の順位の比較

EvoSuite は，Java を対象としたテストケース自動生成ツールであり，入力として与えられたクラスに対するテストケースを出力する．EvoSuite の実行には，Defects4J で提供されているスクリプトを利用した．EvoSuite によるテストケースの生成の制限時間はスクリプトにおけるデフォルト値の 100 秒とした．100 秒間に生成された全てのテストケースを提案手法のステップ 1 で生成されたテストケースとして利用した．

4.4 評 価 指 標

本実験では，欠陥箇所の順位を評価指標とした．プログラムの文を疑惑値が高い順に並べた時に，欠陥箇所の順位が高いほど欠陥限局の精度が高いと評価する．同じ疑惑値の文が複数存在する場合，欠陥箇所の順位はそれらの中で最も低い順位となるように評価する．例えば，欠陥箇所の疑惑値が 3 番目に大きい値で欠陥箇所と同じ疑惑値の文が他に 2 つ存在する場合，欠陥箇所の順位は 5 位として評価する．この指標は疑惑値の高い文から順に確認した場合，欠陥箇所を特定するまでに最大で何個の文を確認する必要があるかを表す．

4.5 実 験 結 果

どの程度の欠陥箇所で順位が向上したか

既存のテストスイートに提案手法で生成したテストケースを追加した場合と既存のテストスイートのみを用いた場合の各欠陥箇所の順位を比較した．その結果，順位が向上した欠陥箇所，順位が変化しなかった欠陥箇所，順位が低下した欠陥箇所がそれぞれ何個あったかを表 2 に示す．表 2 より提案手法で生成したテストケースを追加した場合に順位が向上した欠陥箇所が 162 個 (70.4%)，順位が変化しなかった欠陥箇所が 45 個 (19.6%)，順位が低下した欠陥箇所が 23 個 (10.0%) あったことがわかる．提案手法で生成したテストケースを追加することで，90.0%の欠陥箇所において既存のテストスイートのみを用いた場合と同等以上の精度で欠陥箇所を推測できた．

欠陥箇所の順位がどの程度向上したか

欠陥箇所の順位の箱ひげ図を図 5 に示す．縦軸が欠陥箇所の順位を表しており，値が小さいほど順位が高いことを表す．図 5 より，提案手法で生成したテストケースを追加したテストスイートを用いた方が欠陥箇所の順位が高いことが確認できる．中央値を比較すると，提案手法で生成したテストケースを追加したテストスイートを用いた場合の順位は 54 位，既存のテストスイートのみを用いた場合の順位は 76 位であった．提案手法で生成したテストケースを追加することで，欠陥箇所の順位は中央値で 22 向上した．

Fraction.java

```
1  private Fraction(double value, …, …, …)
2    throws FractionConversionException
3  {
4    long overflow = Integer.MAX_VALUE;
5    double r0 = value;
6    double a0 = (long) FastMath.floor(r0);
7    if(a0 > overflow) {  //if(FastMath.abs(a0) > overflow)
8      throw new FractionConversionException(value, a0, 1l);
9    }
10   ...
11   ...
12 }
```

テストケース	t1	a1
4	●	●
5	●	●
6	●	●
7	●	●
8	●	●
10	●	●
11	●	
テストケースの成否(成功=P,失敗=F)	F	P

(a) 対象のプログラム

既存のテストケースt1

```
1 @Test
2 void test_t1() {
3   try {
4     new Fraction(-1.0e10);
5     Assert.fail("an exception should have been thrown");
6   } catch (CovergenceException e) { }
7 }
```

(b) 既存のテストケース

追加したテストケースa1

```
1 @Test
2 void test_a1() {
3   Fraction fraction0 = new Fraction(0.0);
4   assertEquals(0, fraction0.getDenominator());
5 }
```

(c) 追加したテストケース

図 6: テストケースを追加して精度が低下した例

4.6 考　　察

提案手法で生成したテストケースを既存のテストスイートに追加して SBFL を適用した場合に，既存のテストスイートのみを用いて SBFL を適用するよりも精度が低下した欠陥箇所についての考察を述べる．

提案手法で生成したテストケースを追加したことで精度が低下した例 (Math の欠陥 ID26) を図 6 に示す．図 6(a) のプログラム Fraction は，引数で与えられた値の絶対値が Integer の最大値より大きい場合に例外を投げるという処理を含む．7 行目が欠陥箇所であり，引数で与えられた値の絶対値を計算する処理が抜けている．このプログラムに対し，図 6(b) に示す既存のテストケース t1 と図 6(c) に示す追加したテストケース a1 を実行する．t1 における Fraction への入力は $-1.0e10$ であり，その絶対値は Integer の最大値より大きい．本来であれば例外が投げられるべきであるが，絶対値を計算しないため例外は投げられず，t1 は失敗する．一方，a1 における Fraction の入力は 0 である．0 の絶対値は 0 であり，Integer の最大値より小さい．そのため，

表 2: 順位が向上した欠陥箇所，変化しなかった欠陥箇所，低下した欠陥箇所の数

	欠陥箇所の数 (全 230 個)	割合
順位向上	162 個	70.4%
変化なし	45 個	19.6%
順位低下	23 個	10.0%

欠陥を含んでいても正しいプログラムと同じように例外は投げられず，a1 は成功する．結果として，a1 は欠陥箇所を実行して成功したため欠陥箇所の疑惑値は低下し，精度が低下したと考えられる．

このように追加したテストケースが欠陥箇所を実行して成功してしまう場合，精度が低下する可能性があると考えられる．

5. 妥当性への脅威

5.1 テストケース生成

提案手法では，既存テストスイートに追加するテストケースの生成に EvoSuite を用いた．他のテストケース自動生成ツールや手動でテストケースを生成した場合，異なる結果が得られる可能性がある．また，EvoSuite によるテストケースの生成には乱択による操作が含まれる．本実験と同様の設定でテストケースを生成した場合でも，異なるテストケースが生成され，異なる結果が得られる可能性がある．

5.2 疑惑値の計算

本研究では，疑惑値の計算には Ochiai を用いた．他の疑惑値計算手法を用いた場合，異なる結果が得られる可能性がある．

5.3 実 験 対 象

本実験では，Defects4J に含まれる Math を実験対象とした．他のプロジェクトを対象に実験を行った場合，異なる結果が得られる可能性がある．

6. あ と が き

本研究では，SBFL の精度を高めるために，既存のテストスイートの経路網羅を高めるテストケースを生成する手法を提案した．評価実験として，提案手法で生成したテストケースを既存のテストスイートに追加して SBFL を適用し，既存テストスイートのみを用いて SBFL を適用した結果と比較した．その結果，70.4%の欠陥箇所において，既存のテストスイートのみを用いる場合よりも提案手法により生成したテストケースを追加した方が SBFL の精度が向上した．また，欠陥箇所の順位は中央値で 22 向上した．

今後の取り組みとして，追加で実験を実施し，より詳細な考察をする必要があると考えている．本稿の実験では，既存テストスイートのみで SBFL を適用した結果と比較した．一方で，テストケースを選択する条件を変更して生成したテストケースを追加して SBFL を適用した結果との比較が実施できておらず，重要な課題となっている．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：17H01725）の助成を得て行われた．

文 献

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, no.1, pp.4–12, 2002.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software - quantify the time and cost saved using reversible debuggers,” 2013.

[3] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff, “Sober: Statistical model-based bug localization,” Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.286–295, ESEC/FSE-13, 2005.

[4] B. Korel, “Pelas-program error-locating assistant system,” IEEE Transactions on Software Engineering, vol.14, no.9, pp.1253–1260, Sep. 1988.

[5] W. Jin and A. Orso, “F3: Fault localization for field failures,” Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp.213–223, ISSTA 2013, 2013.

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

[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” Proceedings of the 31st International Conference on Software Engineering, pp.364–374, ICSE ’09, 2009.

[8] A. daSilva, Meyer, A. Augusto, Franco Garcia, A. Pereira, de Souza, and C.L. deSouza, Jr., “Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l),” Genetics and Molecular Biology, vol.27, no.1, pp.83–91, 2004.

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

[10] E. Miller, Tutorial Program Testing Techniques: COMPSAC 77, November 8-11, 1977, Sheraton-O’Hare Motor Hotel, Chicago O’Hare Airport; the IEEE Computer Society’s First International Computer SOftware & Applications Conference, Chicago, IEEE Communications Society, 1977.

[11] Y. Küçük, T.A. Henderson, and A. Podgurski, “The impact of rare failures on statistical fault localization: the case of the defects4j suite,” 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)IEEE, pp.24–28 2019.

[12] R. Just, D. Jalali, and M.D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” In Proc. International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[13] J. Sohn and S. Yoo, “Fluccs: Using code and change metrics to improve fault localization,” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp.273–283, ISSTA 2017, 2017.

[14] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp.416–419, ESEC/FSE ’11, 2011.