例外処理を検査するテストが
実行経路に基づく欠陥限局手法に与える影響の調査

吉岡 遼[1]　肥後 芳樹[1]　柗本 真佑[1]　楠本 真二[1]　伊藤 信治[2]　Phan Thi Thanh Huyen[2]

概要：ソフトウェア開発において，デバッグ作業は多大なコストを要する．計算機によるデバッグ作業の支援により，デバッグコストの削減が期待される．デバッグ作業を支援する技術の 1 つに，欠陥限局と呼ばれる技術がある．これまでに数多くの欠陥限局手法が提案されており，中でも実行経路に基づく欠陥限局手法が注目を集める．実行経路に基づく欠陥限局手法は，入力として欠陥を含むプログラムとそのテストを与えると，欠陥箇所を推定し出力する．これまでの研究により，実行経路に基づく欠陥限局手法は与えるテストによって欠陥限局の正確さに差異が生じることが示唆されている．本研究では，例外処理を検査するテストに着目し，これらが実行経路に基づく欠陥限局手法の正確さに与える影響を調査する．例外処理は通常の制御フローと切り離されるため，例外処理を検査するテストと通常の制御フローを検査するテストでは実行経路が異なり，実行経路に基づく欠陥限局手法の正確さに大きな影響を与えると考えたためである．実際の開発過程で生じた欠陥とミューテーションツールで人工的に生成した欠陥を対象に調査し，失敗テストが全て例外期待テストの場合，実行経路に基づく欠陥限局手法の結果は信頼できる傾向にあることを確認した．

1. はじめに

ソフトウェア開発において，デバッグ作業は多大なコストを要する．ソフトウェア開発に要するコストの半分以上をデバッグ作業が占めるという報告もされている [1], [2]．計算機によりデバッグ作業を支援することで，デバッグコストの削減が期待される．

計算機によりデバッグ作業を支援する技術の 1 つに，欠陥限局がある．欠陥限局とは，計算機によりプログラム中に潜む欠陥箇所を推定する技術である．これまでに数多くの欠陥限局手法が提案されている [3], [4]．中でも，**S**pectrum-**b**ased **F**ault **L**ocalization (SBFL) は近年盛んに研究されている欠陥限局手法である [5]．SBFL は欠陥を持つプログラムとテストを与えると，テストの実行経路に基づきプログラム中に含まれる欠陥箇所を推定し出力する．

これまでの研究により，入力として与えるテストが SBFL の正確さに影響を及ぼすことが示唆されている [6]．本研究では，例外処理を検査するテスト (以降，例外期待テスト) に着目し，これらが SBFL に与える影響を調査する．例外期待テストに着目する理由を述べる．例外処理と通常の制御フローを切り離すことはプログラムの堅牢性を高める上で重要であり [7]，例外処理と通常の制御フローは明示的に分離すべきとの研究報告がある [8]．そのため，例外期待テストと通常の制御フローを検査するテストでは実行経路が異なり，SBFL の正確さに大きな影響を与えると筆者らは考えた．また，例外期待テストはプログラム中への欠陥の混入を抑制できることが報告されており [9]，SBFL のより正確な欠陥限局に有効であると筆者らは仮説を立てた．よって，例外期待テストが SBFL の正確さに与える影響を調査する．

本研究では実際のソフトウェアの開発過程で生じた欠陥とミューテーションツールを用いて人工的に生成した欠陥を対象に調査を行った．調査の結果，失敗テストが全て例外期待テストの場合，例外期待テストが全く含まれない場合と比べ，実際のソフトウェア開発過程で生じた欠陥では約 33%，人工的に生成した欠陥では約 66%，デバッグ時に人が確認する必要のあるプログラム文が少なく済むことを確認した．

また，例外期待テストが検査する例外の種類をカスタム例外と標準/サードパーティ例外に分類し，それぞれが SBFL の結果に与える影響を調査した．その結果，失敗テストに占める標準/サードパーティ例外の発生を検査する例外期待テストの割合が高い場合，特に SBFL の結果が正確な傾向にあることを確認した．

本研究の主な貢献として以下の 3 点が挙げられる．

[1] 大阪大学大学院情報科学研究科 大阪府吹田市
[2] 日立製作所 東京都千代田区

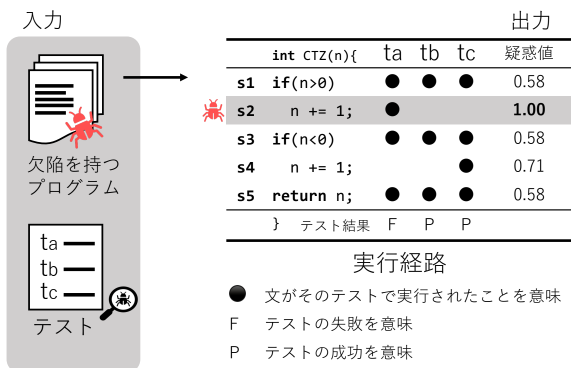


図 1: SBFL の入力から出力までの流れ

- 例外期待テストが SBFL の結果に与える影響を調査した初めての研究である．
- 失敗テストが全て例外期待テストの場合に SBFL の結果が正確な傾向にあることを確認した．
- 失敗テストが全て標準/サードパーティ例外を検査する例外期待テストの場合，特に SBFL の結果が正確な傾向にあることを確認した．

2. 準備

2.1 Spectrum-based Fault Localization (SBFL)

SBFL とは，欠陥を含んだプログラムから欠陥箇所を推定する技術である．SBFL は，多くの失敗テストによって実行されるプログラム文は欠陥を含む可能性が高く，多くの成功テストによって実行されるプログラム文は欠陥を含む可能性が低いというアイデアに基づく．SBFL の入力から出力までの流れを図 **1** に示す．入力として欠陥を含むプログラムとそのテストを与えると，各テストの成否と実行経路を取得し，出力として疑惑値を算出する．疑惑値は各プログラム文に割り振られ，その文が欠陥箇所である可能性の高さを表す．

疑惑値の算出式には様々な種類があり，Abreu らにより Ochiai が優れた算出式であると結論づけられる [10]．そのため，本研究では Ochiai を算出式に用いる．Ochiai の算出式は以下の式 (1) で表される．

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

s：疑惑値計算対象の文
$susp(s)$：文 s の疑惑値
$totalfails$：失敗テストの総数
$fail(s)$：文 s を実行する失敗テストの数
$pass(s)$：文 s を実行する成功テストの数

2.2 例外期待テストと非例外期待テスト

例外期待テストとは，例外処理を検査するテストであり，意図した通りに例外が発生するか検証する．本研究では Java 言語で記述されたプロジェクトを対象に調査を進める．これまでの研究 [11] に従い，例外期待テストは表 **1** に記した例外の検出手法を用いるテストとする．

```
@Test(expected=ArithmeticException.class)
public void testIncrementToIntegerMaxValue() {
  Math.incrementExact(Integer.MAX_VALUE);
}
```

図 2: 例外期待テストの一例

```
@Test
public void testIncrementExact() {
  int inc = Math.incrementExact(10);
  assertEquals(inc,11);
}
```

図 3: 非例外期待テストの一例

例外期待テストの例を図 **2** に記す．図 2 のテストメソッドは，意図した通りに `ArithmeticException` が発生するか検証するテストである．`Math.incrementExact()` は引数に与えられた `int` 型の数値にインクリメントした値を返すメソッドであり，インクリメントした結果オーバーフローする場合，`ArithmeticException` をスローする．図 2 のテストでは `Math.incrementExact()` の引数に `Integer.MAX_VALUE` を与えているためオーバーフローを起こし，`ArithmeticException` をスローする．

例外はカスタム例外と標準/サードパーティ例外に分けられる．カスタム例外とは，開発者が独自に実装した例外であり，標準/サードパーティ例外とは，標準/サードパーティライブラリで実装された例外である．例外期待テストのうち，意図した通りにカスタム例外が発生するか検証するテストをカスタム例外期待テスト，標準/サードパーティ例外が発生するか検証するテストを標準/サードパーティ例外期待テストと呼ぶ．

また，例外処理以外を検査するテストを非例外期待テストとする．例外期待テスト以外のテストは全て非例外期待テストである．非例外期待テストを図 **3** に記す．図 3 のテストは，`Math.incrementExact()` の引数に 10 を与え，返り値に 11 を期待するテストである．

表 1: 先行研究 [11] が利用した例外処理を検査するテストフレームワークと例外の検出手法

フレームワーク	例外の検出手法
JUnit	assertThrows の利用
	@Test の expected 属性での指定
	ExpectedException の利用
TestNG	@Test の expectedExceptions 属性での指定
AssertJ	assertThatThrownBy の利用
	assertThatExceptionOfType の利用
	assertThatIOException の利用

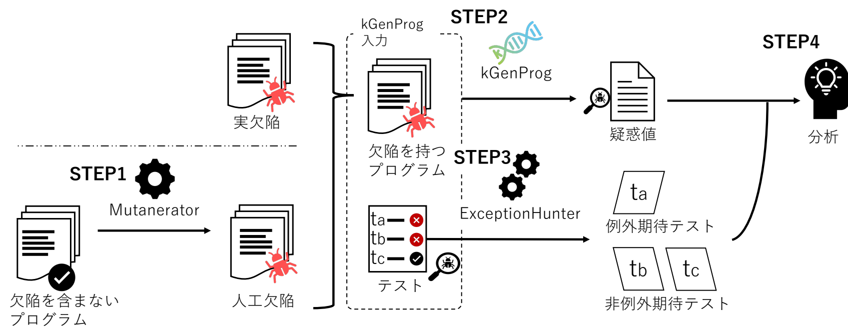


図 4: 実験の流れ

3. Research Questions

本研究では，例外期待テストが SBFL に与える影響を調査するため，以下の Research Question を設定した．

RQ1: 例外期待テスト，非例外期待テストが SBFL の正確さに与える影響は異なるか？

テストに占める例外期待テストの割合とその成否により SBFL の正確さにどのような差が生じるか調査する．SBFL の正確さによって差が生じる場合，SBFL の結果が信頼できるかという事前の判断や，どういったテストを記述するべきかを開発者へ示唆できると考え調査する．

RQ2: 例外期待テストと非例外期待テストの実行経路に違いはあるか？

例外期待テストと非例外期待テストの実行経路の違いを調査する．例外処理と通常の制御フローは切り離すべきとの研究報告がある [8] ため，例外期待テストと非例外期待テストの実行経路には，長さや欠陥箇所を実行する失敗テストの割合に違いがあると筆者らは考えた．

RQ3: カスタム例外期待テスト，標準/サードパーティ例外期待テストが SBFL の正確さに与える影響は異なるか？

例外の種類がカスタム例外か標準/サードパーティ例外かによって SBFL の正確さに与える影響が異なるか調査する．カスタム例外期待テストと標準/サードパーティ例外期待テストにより SBFL の正確さに差が生じる場合，開発者にどちらのテストを積極的に書くべきか示唆できると考え，本 RQ を調査する．

4. 実験設定

4.1 実験で利用するツール

本実験では以下のツールを利用する．

kGenProg[*1]

kGenProg は Higo らによって開発された自動欠陥修正ツールである [12]．本研究では，kGenProg が対象のプログラムから実行経路を取得し疑惑値を算出する部分のみを限定して利用する．

[*1] https://github.com/kusumotolab/kGenProg

ExceptionHunter[*2]

ExceptionHunter は Francisco らによって開発された，Java プログラムの静的解析ツールである [11]．あるテストが例外期待テストかの判別に利用する．

Mutanerator[*3]

Mutanerator は Java プログラムを対象としたミュータント生成ツールである．表 2 に記したミュータント演算子を適用できる．

4.2 実験対象

本研究では以下の 2 つを実験対象とする．

- 実世界のソフトウェア開発で生じた欠陥 (以降，実欠陥)
- Mutanerator を利用し人工的に生成した欠陥 (以降，人工欠陥)

実欠陥として Defects4J[13] を用いる．Defects4J は実際の Java プロジェクトの開発過程で生じた欠陥を収集したデータセットであり，多くの先行研究で実験対象として利用されている [14], [15]．本実験では，これまでの研究 [16] を参考に Defects4J に含まれる Math，Chart，Lang，Jsoup，JacksonCore，Codec の 6 個のプロジェクトに含まれる欠陥を実験対象とする．なお，いくつかの欠陥は筆者らの環境で想定通りに動作しなかったため取り除いた．利用した欠陥はオンライン上[*4]に掲載する．

人工欠陥には，実欠陥として用いた Defects4J の 6 個の

表 2: 実験で用いるミュータント演算子

ミュータント演算子	説明
Conditional Boundary	関係演算子の境界を変更する
Increments	インクリメント/デクリメントの入れ替え
Invert Negatives	負数を正数に書き換える
Math	算術演算子を書き換える
Negate Conditionals	関係演算子を書き換える
Void Method Calls	void 型のメソッド呼び出しを削除
Primitive Returns	プリミティブ型の戻り値を 0 に書き換える

[*2] https://github.com/easy-software-ufal/exceptionhunter
[*3] https://github.com/kusumotolab/Mutanerator
[*4] https://github.com/Haur514/ses2023

プロジェクトから欠陥を取り除き，新たにミューテーションツールを用いて生成した欠陥を利用した．

4.3 実験の流れ

実験の流れを図 4 に記す．図中のテスト t_a，t_b は失敗テスト，t_c は成功テストである．

STEP1 欠陥を含まないプログラムにミューテーションツールを適用し人工欠陥を生成する．欠陥を含まないプログラムとは，実欠陥の Defects4J から欠陥を取り除き，全てのテストに通る状態としたプログラムである．ミューテーションツールでこれらのプログラムに変更を加え，1 つ以上のテストに失敗するプログラムのみ人工欠陥に加える．

STEP2 欠陥を持つプログラムとそのテストを kGenProg に与える．ここで，欠陥を持つプログラムとは 1 つ以上のテストに失敗するプログラムを指す．kGenProg はプログラムをテストに通し，SBFL に基づき疑惑値を算出する．

STEP3 テストを ExceptionHunter に与え，例外期待テストと非例外期待テストに分類する．図 4 では簡略化のため省略したが，ExceptionHunter は例外期待テストをさらにカスタム例外期待テストと標準/サードパーティ例外期待テストに分類可能である．

STEP4 得られた疑惑値と例外期待テストの情報から例外期待テストが SBFL に与える影響を調査する．

4.4 評価指標

本研究では Rank と rTop-N を評価指標に用いる．

Rank

Rank を説明するにあたり，まず欠陥箇所の順位を定義する．プログラム文を疑惑値が高い順に並べた時，欠陥箇所を含むプログラム文が何番目に位置するかを欠陥箇所の順位とする．欠陥箇所と同じ疑惑値を付与されたプログラム文が複数ある場合，欠陥箇所の順位はそれらの順位タイの平均順位とする．

ある欠陥に存在する欠陥箇所の中で，最も高い欠陥箇所の順位を Rank とする．欠陥の修正において，1 つ目の欠陥箇所の特定が重要だからである [14], [15]．例えば，欠陥 A に含まれる 3 つの欠陥箇所の順位がそれぞれ 2，5，10 位であった場合，欠陥 A の Rank は 2 となる．

rTop-N

rTop-N とは Top-N を参考に作成した評価指標である．Top-N とは，欠陥箇所の順位が N 位以内であるような欠陥の個数を表す．Top-N は SBFL の性能を評価する上で有効な指標であり，多くの研究で評価指標として利用されている [14], [15]．しかし，Top-N は標本の大きさが異なる欠陥の比較には適さない．例えば，1,000 個の欠陥で Top-5 が 100 であるのと，200 個の欠陥で Top-5 が 100 であるのでは意味が異なる．本研究ではテスト中の例外期待テストの割合により欠陥を分類しており，分類後の欠陥の個数にばらつきがあるため，標本の大きさが異なる欠陥を比較する必要がある．そのため，Top-N をそのまま利用するのではなく，Top-N を標本の大きさで正規化した rTop-N を指標に用いる．rTop-N は式 (2) により定義される．

$$\text{rTop-N} = \frac{\text{Rank が } N \text{ 以内の欠陥の数}}{\text{標本の大きさ}} \quad (2)$$

例えば，欠陥 A の Rank が 2 であり，欠陥 B の Rank が 10 とする．この時，rTop-5 を求める．欠陥 A と欠陥 B において，Rank が 5 以内であるのは欠陥 A のみである．そのため，rTop-5 $= 1/2 = 0.5$ となる．

本研究では N の値に 5 を用いる．これまでの研究により，73.58%の開発者は欠陥限局手法が出力する上位 5 個の要素しか見ないことが報告されているためである [17]．

5. 実験結果と考察

実験結果から RQ1-RQ3 に回答する．

5.1 RQ1: 例外期待テスト，非例外期待テストが SBFL の正確さに与える影響は異なるか？

本 RQ では，成功/失敗テストに含まれる例外期待テストの割合が SBFL の正確さに与える影響を調査する．失敗テストに占める例外期待テストの割合を `rEEFail`，成功テストに占める例外期待テストの割合を `rEEPass` と定義する．例えば，図 4 にて失敗テストは t_a，t_b の 2 つであり，例外期待テストは t_a のみであるため `rEEFail` は $1/2 = 0.5$ となる．

rEEFail が SBFL の正確さに与える影響の調査

`rEEFail` によって欠陥を以下のように分類し調査を進める．

- **rEEFail = 0**
  失敗テスト中に例外期待テストが含まれない場合．
- **0 < rEEFail < 1**
  失敗テスト中に例外期待テスト/非例外期待テストが

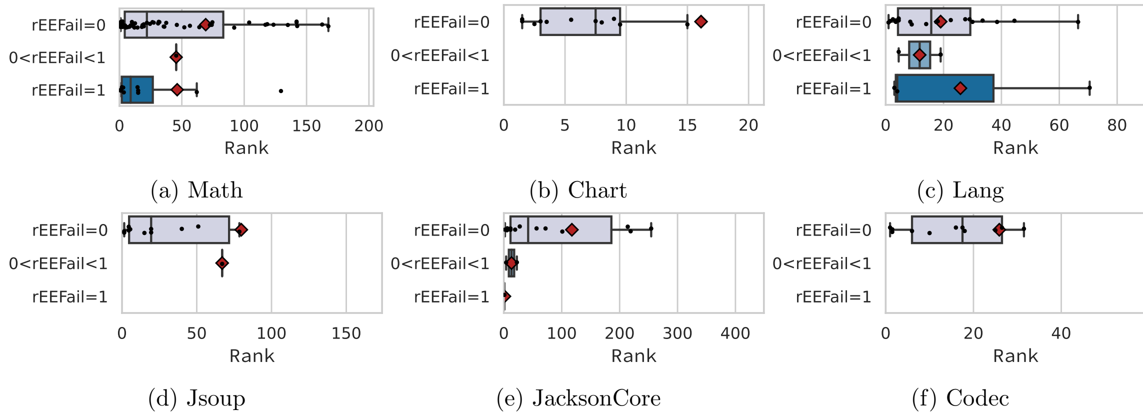
表 3: `rEEFail` による区分ごとの該当する欠陥の個数

実欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	67	13	20	14	14	13
0 < rEEFail < 1	1	0	2	1	2	0
rEEFail = 1	12	0	3	0	1	0
欠陥数合計	80	13	25	15	17	13

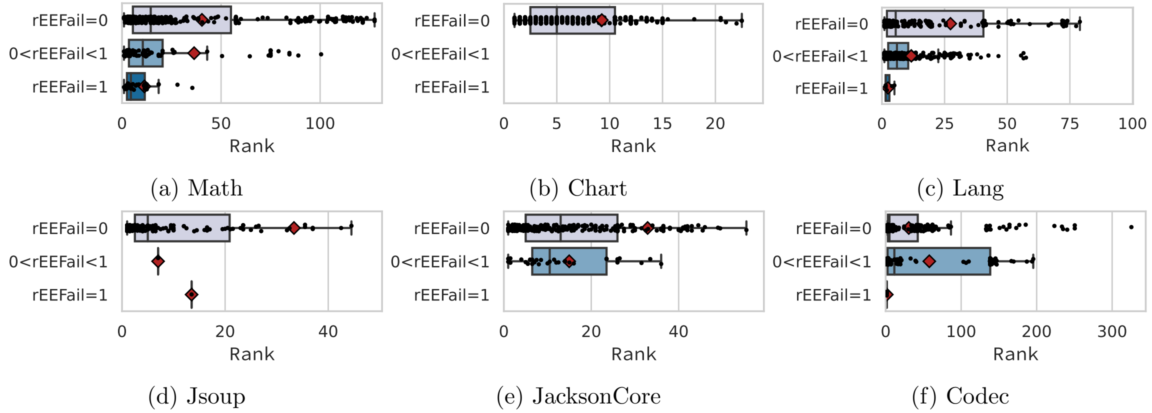
人工欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	399	488	277	166	307	272
0 < rEEFail < 1	93	0	566	1	33	150
rEEFail = 1	39	0	7	1	0	4
欠陥数合計	531	488	850	168	340	426



(a) Math (b) Chart (c) Lang
(d) Jsoup (e) JacksonCore (f) Codec

図 5: 実欠陥における Rank の分布



(a) Math (b) Chart (c) Lang
(d) Jsoup (e) JacksonCore (f) Codec

図 6: 人工欠陥における Rank の分布

混在する場合．

- **rEEFail = 1**
  失敗テストが全て例外期待テストの場合．

それぞれの場合に該当する欠陥の個数を**表 3** に記す．上側の表が実欠陥の個数を，下側の表が人工欠陥の個数を表す．表 3 から，欠陥の多くが rEEFail $= 0$ に分類されるとわかる．また，実欠陥において $0 <$ rEEFail < 1 となる欠陥は 6 個の実験対象プロジェクトで合計 6 個と少ない．そのため，実欠陥での rTop-5 と Rank の比較は rEEFail $= 0$ と rEEFail $= 1$ でのみ行う．

rEEFail と rTop-5 の関係を**表 4** に記す．表中の横線 “—” は，rEEFail の条件に該当する欠陥が存在しないことを，太文字で書かれた数値は各プロジェクトで最も良い rTop-5 を意味する．

表 4: rEEFail と rTop-5

実欠陥 rTop-5

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail $= 0$	0.30	**0.38**	0.35	**0.29**	0.07	**0.23**
$0 <$ rEEFail < 1	0.00	—	0.50	0.00	0.50	—
rEEFail $= 1$	**0.50**	—	**0.67**	—	**1.00**	—

人工欠陥 rTop-5

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail $= 0$	0.22	**0.52**	0.49	**0.51**	**0.25**	0.51
$0 <$ rEEFail < 1	0.38	—	0.34	0.00	0.18	0.43
rEEFail $= 1$	**0.54**	—	**1.00**	0.00	—	**1.00**

まず，rEEFail が rTop-5 に与える影響を調査する．実欠陥での，rEEFail $= 0$ と rEEFail $= 1$ の rTop-5 を比較する．実欠陥において，rEEFail $= 1$ となる欠陥を持つプロジェクトは，Math，Lang，JacksonCore の 3 つであった．表 4 からわかるように，3 つのプロジェクト全てで rEEFail $= 1$ の時，rEEFail $= 0$ よりも rTop-5 が良いことを確認した．次に，人工欠陥での rTop-5 を比較する．rEEFail $= 1$ となる欠陥を含む 4 個のプロジェクトのうち，Jsoup を除く Math，Lang，Codec の 3 プロジェクトで rEEFail $= 1$ の時，rTop-5 が最も良いことを確認した．Jsoup は rEEFail $= 1$ での rTop-5 が悪かったが，これは rEEFail $= 1$ となる欠陥が 1 個しかないためと考える．これらの結果から，rEEFail $= 1$ の時，rEEFail $= 0$ や，$0 <$ rEEFail < 1 より rTop-5 が良い傾向にあると結論づける．

次に，rEEFail が Rank に与える影響を調査する．**図 5** と**図 6** は，それぞれ実欠陥と人工欠陥における Rank の分布である．横軸が Rank を表す．箱髭図に重ねて黒色の点で Rank の分布をプロットしている．箱髭図の見やすさのため，Rank が外れ値となる欠陥はグラフから除外した．図

中の赤色の菱形は，Rank の平均値を表す．

実欠陥における Rank の分布を記した図 5 に着目する．rEEFail = 1 となる欠陥を含むプロジェクトは，Math，Lang，JacksonCore の 3 つである．Math や JacksonCore で，rEEFail = 1 の時，rEEFail = 0 より箱髭図における第一四分位数，中央値，第三四分位数，平均値が良いことを確認した．また，Lang で，rEEFail = 1 の平均値や第三四分位数は rEEFail = 0 より悪いが，第一四分位数や中央値は rEEFail = 0 より良い結果が得られた．これらの結果から，実欠陥において rEEFail = 1 は，rEEFail = 0 より Rank が良い傾向にあると考える．マンホイットニーの U 検定を用いて rEEFail = 1 と rEEFail = 0 間で Rank の分布に統計的有意差があるか調査した．なお，実欠陥において rEEFail = 1 となる欠陥の個数が少ないため，プロジェクトによる区分はしない．その結果，rEEFail = 0 と rEEFail = 1 間の p 値は 0.083 であり，統計的有意差は確認されなかった．統計的有意差がない理由に，rEEFail = 1 となる欠陥が少ないことが挙げられると筆者らは考える．

次に，人工欠陥における Rank の分布を記した図 6 に着目する．まず，rEEFail = 1 に注目する．rEEFail = 1 となる欠陥を含むプロジェクトは，Math，Lang，Jsoup，Codec の 4 つである．これらのうち Jsoup を除く 3 つのプロジェクト全ての箱髭図にて，rEEFail = 1 の時，その他の場合と比較して平均値，第一四分位数，中央値，第三四分位数が優れていた．Jsoup でのみ，rEEFail = 1 での第一四分位数や中央値は rEEFail = 0 や 0 < rEEFail < 1 を下回るが，これは rEEFail = 1 となる欠陥が 1 個しかないためと考える．次に，rEEFail = 0 と，0 < rEEFail < 1 を比較する．箱髭図から，Math や Lang，JacksonCore では 0 < rEEFail < 1 の方が良い Rank を取るが，Codec では rEEFail = 0 の方が良い Rank を取ることを確認した．そのため，rEEFail = 0 と 0 < rEEFail < 1 のどちらの Rank がより良いかは実験対象プロジェクトによって異なるため，本研究からは不明であった．以上の結果から，人工欠陥において，rEEFail = 1 は rEEFail = 0，0 < rEEFail < 1 よりも Rank が良い傾向にあると筆者らは考える．

マンホイットニーの U 検定を用いて，人工欠陥において rEEFail = 1 と rEEFail = 0 間，rEEFail = 1 と 0 < rEEFail < 1 間で Rank の分布に統計的有意差があるかを調査した．調査結果を表 **5** に記す．なお，rEEFail = 1 や 0 < rEEFail < 1 となる欠陥を持たないプロジェクトは表から除外している．表中の太字は，p 値が有意水準である 0.01 より小さいことを意味する．

まず，rEEFail = 0 と rEEFail = 1 間での p 値に着目する．Math と Codec で，p 値が有意水準を下回ることを確認した．Jsoup では有意差は確認できなかったが，rEEFail = 1 となる欠陥が 1 個しかないためだと考える．また，Lang においても p 値は有意水準を下回らなかったが，rEEFail = 1 の時，rEEFail = 0 より Rank の平均値が 24.93 良いことを確認した．これらの結果から，人工欠陥において，rEEFail = 1 の時，rEEFail = 0 よりも Rank は良い傾向にあると結論づける．

次に，0 < rEEFail < 1 と rEEFail = 1 間での p 値に着目する．Lang でのみ，p 値が有意水準を下回ることを確認した．Jsoup で有意差が確認できなかった理由は，0 < rEEFail < 1 となる欠陥が 1 個しかないためと考える．しかし，Math や Codec には，rEEFail = 1 となる欠陥がそれぞれ 39 個，4 個あるにもかかわらず，統計的な有意差は確認されなかった．そのため，rEEFail = 1 と 0 < rEEFail < 1 間で，どちらの Rank がより良いかは不明と筆者らは結論づける．

得られた実欠陥と人工欠陥での実験結果から，rEEFail = 1 の時，rEEFail = 0 と比べ SBFL の結果は正確な傾向にあると結論づける．rEEFail = 1 の時，rEEFail = 0 と比べ Rank の平均値は実欠陥では約 33%，人工欠陥では約 66%良く，これは欠陥箇所を発見するまでに調査すべきプログラム文の数が実欠陥では約 33%，人工欠陥では約 66% 少なく済むことを意味する．rEEFail = 1 の時に SBFL の

表 5: 人工欠陥における rEEFail ごとの Rank 間の p 値

	0 < rEEFail < 1 rEEFail = 0	0 < rEEFail < 1 rEEFail = 1	rEEFail = 0 rEEFail = 1
Math	**3.8E-04**	5.6E-02	**1.9E-06**
Lang	5.5E-02	**4.6E-03**	1.8E-02
Jsoup	7.6E-01	1.0	5.3E-01
Codec	5.1E-02	1.5E-02	**2.3E-03**

表 6: rEEPass による区分ごとの該当する欠陥の個数

実欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEPass = 0	21	11	11	14	16	5
0 < rEEPass < 1	59	2	14	1	1	8
rEEPass = 1	0	0	0	0	0	0

人工欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEPass = 0	81	325	8	105	202	13
0 < rEEPass < 1	450	163	842	63	138	413
rEEPass = 1	0	0	0	0	0	0

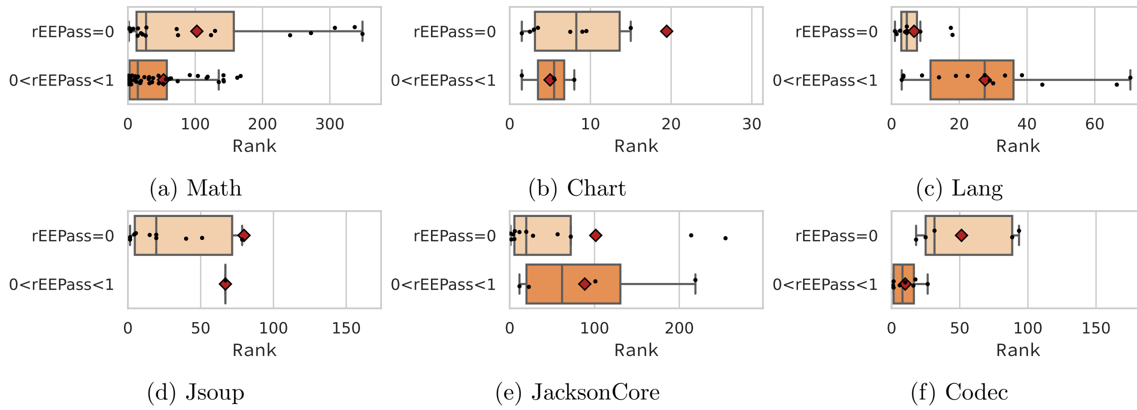
表 7: rEEPass と rTop-5

実欠陥 rTop-5

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEPass = 0	0.19	**0.45**	**0.64**	**0.29**	**0.19**	0.00
0 < rEEPass < 1	**0.37**	0.00	0.21	0.00	0.00	**0.38**
rEEPass = 1	—	—	—	—	—	—

人工欠陥 rTop-5

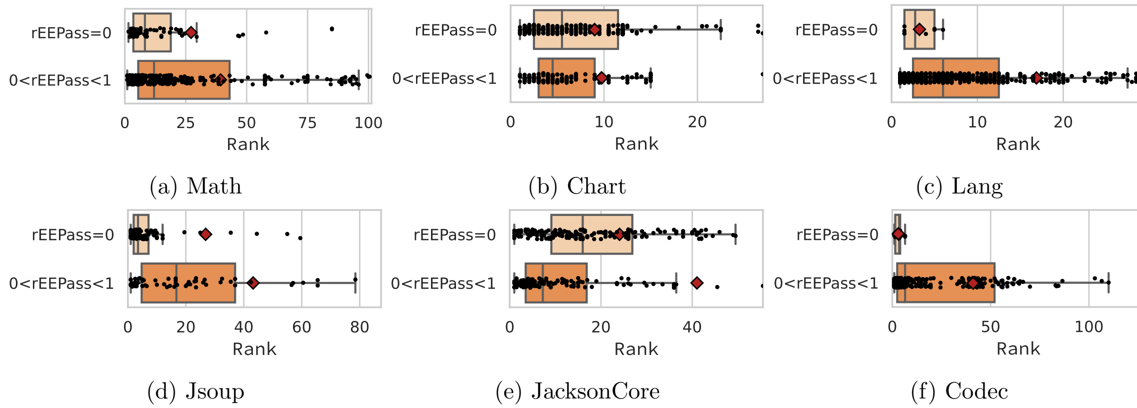
	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEPass = 0	**0.40**	0.49	**0.88**	**0.63**	0.12	**0.85**
0 < rEEPass < 1	0.25	**0.58**	0.39	0.30	**0.43**	0.48
rEEPass = 1	—	—	—	—	—	—



(a) Math (b) Chart (c) Lang

(d) Jsoup (e) JacksonCore (f) Codec

図 7: 実欠陥における Rank の分布



(a) Math (b) Chart (c) Lang

(d) Jsoup (e) JacksonCore (f) Codec

図 8: 人工欠陥における Rank の分布

結果が正確な傾向にある理由については，RQ2 における調査結果を示した際に考察する．

rEEPass が SBFL の正確さに与える影響の調査

rEEFail と同様に欠陥を rEEPass = 0，0 < rEEPass < 1，rEEPass = 1 に分類し調査を進める．それぞれの場合に該当する欠陥の個数を表 **6** に記す．実欠陥/人工欠陥合わせ，rEEPass = 1 となる欠陥は 1 つも存在しなかった．

rEEPass と rTop-5 を表 **7** に記す．実欠陥に着目すると，Math や Codec では 0 < rEEPass < 1 の方が rTop-5 が良いのに対し，Chart，Lang，Jsoup，JacksonCore では rEEPass = 0 の方が良い rTop-5 が得られた．人工欠陥でも同様に，rEEPass = 0 と 0 < rEEPass < 1 の rTop-5 を比較した時，どちらの方が良いかはプロジェクトによって異なった．そのため，rEEPass が rTop-5 に与える規則的な影響は確認されなかった．

rEEPass ごとの，実欠陥と人工欠陥における Rank の箱髭図を図 **7**，図 **8** に記す．まず実欠陥に着目する．Math や Chart，Codec では，0 < rEEPass < 1 の Rank は rEEPass = 0 よりも良いが，Lang や JacksonCore では rEEPass = 0 の Rank の方が良いことを確認した．そのため，rEEPass = 0 と 0 < rEEPass < 1 のどちらの Rank がより良いかは実験対象プロジェクトによって異なり不明であった．人工欠陥でも，実欠陥と同様に rEEPass = 0 と 0 < rEEPass < 1 のうち，実験対象プロジェクトによってどちらの Rank がより良いかは異なった．そのため，rEEPass = 0 と 0 < rEEPass < 1 のどちらの Rank がより良い傾向にあるかは本実験からは不明であった．

> **RQ1 への回答**: 失敗テストが全て例外期待テストの時，失敗テストに例外期待テストが全く含まれない場合と比べ SBFL の結果は正確であり，デバッグ時に欠陥箇所を見つけるまでに確認する必要のあるプログラム文が実欠陥では約 33%，人工欠陥では約 66%少なく済むことを確認した．

5.2 RQ2: 例外期待テストと非例外期待テストの実行経路に違いはあるか？

本 RQ では次の項目を調査する．SBFL の正確さに与える影響が特に大きいと筆者らが考えたためである．

- 失敗テストが実行するプログラム文の数
- 欠陥箇所を実行するテストに占める失敗テストの割合

RQ1 より，rEEFail が SBFL の正確さに影響を与えると判明した．そのため，本 RQ では rEEFail に着目し調査

を進める．

失敗テストが実行するプログラム文の数

SBFL は失敗テストで実行された文を欠陥箇所が潜む文の候補とする．そのため，失敗テストが実行する文の数が少ないほど，欠陥箇所の候補となる文の数は少なくなる．よって，失敗テストが実行するプログラム文の数は SBFL の正確さと非常に結びつきが強いと考え，本項目を調査した．

調査結果を表 **8** に記す．実欠陥/人工欠陥のいずれにおいても `rEEFail` = 1 の時，`rEEFail` = 0 と比べ失敗テストが実行する文の数は少ない．`rEEFail` = 1 の時, `rEEFail` = 0 より失敗テストが実行する文の数が少ない理由への考察を述べる．異常な処理が発生すると，プログラムはこれに対処するため例外を throw する．throw された例外は，クラス間の呼び出し関係における最上位レイヤのクラスに伝搬する前に，適切な下位レイヤのクラスで処理される．このことから，例外の発生を検証する例外期待テストは，例外が throw される下位レイヤのクラスの挙動を検証する傾向にあると筆者らは考えた．下位レイヤのクラスが呼び出すプログラム文の総数は上位レイヤのクラスが呼び出すプログラム文の総数と比べ少ないため，非例外期待テストと比べて例外期待テストの実行経路は短くなる．実際に，実欠陥において例外期待テストと非例外期待テストが実行するプログラム文の数を計測したところ，それぞれ平均で 63 行，160 行であった．したがって，例外期待テストは非例外期待テストより実行するプログラム文の数が少なくなり，`rEEFail` = 1 において，失敗テストが実行したプログラム文の数は少ない傾向にあると考察する．

先にも述べたように，SBFL は失敗テストで実行した文を欠陥箇所が潜む文の候補とする．失敗テストの実行経路が短くなると，欠陥箇所の候補が狭まり欠陥箇所をより正しく推定できる．よって，RQ1 の結果である，`rEEFail` = 1 の時，`rEEFail` = 0 と比べ SBFL の結果が正確であるのは，`rEEFail` = 1 の時に失敗テストが実行するプログラム文の数が少ない傾向にあるためだと筆者らは考える．

欠陥箇所を実行するテストに占める失敗テストの割合

SBFL は，失敗テストで実行された文は欠陥箇所である可能性が高く，成功テストで実行された文は欠陥箇所である可能性が低いというアイデアに基づく．そのため，欠陥箇所を実行するテストに占める失敗テストの割合は SBFL の正確さと強く関連する．したがって，本項目を調査する．

表 **9** に欠陥箇所を実行するテストに占める失敗テストの割合を記す．各プロジェクトで最も高い数値を太文字としている．表 9 からわかるように，実欠陥では `rEEFail` = 1 の方が `rEEFail` = 0 と比べ欠陥箇所を実行するテストに占める失敗テストの割合は高い．しかし，人工欠陥については `rEEFail` = 0 の方が `rEEFail` = 1 の場合よりも欠陥箇所を実行するテストに占める失敗テストの割合は高いという逆の結果が得られた．そのため，`rEEFail` が欠陥箇所を実行するテストに占める失敗テストの割合に与える規則的な影響は本実験からは発見できなかった．

> **RQ2 への回答**: 失敗テストが全て例外期待テストの場合，失敗テストが実行するプログラム文の数は少ない傾向にある．

5.3 RQ3: カスタム例外期待テスト，標準/サードパーティ例外期待テストが SBFL の正確さに与える影響は異なるか？

本 RQ では RQ1，RQ2 のように，失敗テスト中のカスタム例外期待テスト，標準/サードパーティ例外期待テストの割合に着目し調査を進める．失敗テスト中のカスタム例外期待テストの割合を `rCEFail`，標準/サードパーティ例外期待テストの割合を `rSTEFail` とする．RQ1 と同様に `rCEFail`，`rSTEFail` によって欠陥を分類し調査を進める．

なお，紙面の都合上，RQ1 の箱髭図のように Rank の分布は示さない．本 RQ では，Rank の平均値を用いて議論を進める．以降において，Rank の平均値を ave(Rank) と表現する．

`rCEFail`，`rSTEFail` により欠陥を分類した時の rTop-5 と ave(Rank) を表 **10** に記す．表の左部分が rTop-5 を，右部分が ave(Rank) を表す．rTop5-All と，ave(Rank)-All はプロジェクトによる区分をせず，実験対象全体で計測した場合の rTop-5 と ave(Rank) を表す．

表 8: `rEEFail` と失敗テストが実行するプログラム文の数

実欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	252	73	62	477	328	89
0 < rEEFail < 1	290	—	32	134	373	—
rEEFail = 1	112	—	39	—	16	—

人工欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	327	74	87	400	292	109
0 < rEEFail < 1	685	—	97	14	1582	159
rEEFail = 1	150	—	8	17	—	3

表 9: `rEEFail` と欠陥箇所を実行するテストに占める失敗テストの割合

実欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	0.48	**0.59**	0.52	0.56	0.73	**0.33**
0 < rEEFail < 1	0.57	—	0.35	**1.00**	**1.00**	—
rEEFail = 1	**0.75**	—	**0.61**	—	**1.00**	—

人工欠陥

	Math	Chart	Lang	Jsoup	JacksonCore	Codec
rEEFail = 0	0.85	**0.89**	0.86	0.74	**0.88**	0.81
0 < rEEFail < 1	**0.90**	—	**0.93**	**1.00**	0.87	0.92
rEEFail = 1	0.76	—	0.76	0.03	—	**1.00**

まず，表上部の実欠陥に着目する．rCEFail について見ると，列 rTop5-All から，$\mathtt{rCEFail} = 0$，$0 < \mathtt{rCEFail} < 1$，$\mathtt{rCEFail} = 1$ の全てで rTop-5 はほぼ同じとわかる．また，ave(Rank) については，列 ave(Rank)-All を見ると，$\mathtt{rCEFail} = 1$ が最も悪い．次に，rSTEFail について見ると，列 rTop5-All から $\mathtt{rSTEFail} = 1$ の時，$\mathtt{rSTEFail} = 0$，$0 < \mathtt{rSTEFail} < 1$ と比べ良い rTop-5 を示すとわかる．$\mathtt{rSTEFail} = 1$ の時，列 rTop5-All の rTop-5 は 0.70 であり，これは欠陥箇所を 70%の割合で 5 位以内に検出できたことを意味する．ave(Rank) についても，$\mathtt{rSTEFail} = 1$ の時，$\mathtt{rSTEFail} = 0$，$0 < \mathtt{rSTEFail} < 1$ と比べ非常に良い値を示した．また，列 rTop5-All と ave(Rank)-All から，$\mathtt{rSTEFail} = 1$ の時，$\mathtt{rCEFail} = 1$ の時よりも rTop-5 は 0.37, ave(Rank) は 75 良いことを確認した．次に，表下部の人工欠陥に着目する．列 rTop5-All と ave(Rank)-All から，$\mathtt{rSTEFail} = 1$ の時，$\mathtt{rCEFail} = 1$ の時よりも rTop-5 は 0.21，ave(Rank) は 4.98 良いことを確認した．実欠陥，人工欠陥の両方において，$\mathtt{rSTEFail} = 1$ の時，$\mathtt{rCEFail} = 1$ の結果と比べて rTop-5 と ave(Rank) が共に良いことを確認した．これらの結果から，$\mathtt{rSTEFail} = 1$ の時，rTop-5 と ave(Rank) は $\mathtt{rCEFail} = 1$ よりも良いと結論づける．

$\mathtt{rSTEFail} = 1$ の時, $\mathtt{rCEFail} = 1$ より rTop-5, ave(Rank) が良い理由は，$\mathtt{rSTEFail} = 1$ の方が失敗テストの実行経路が短いからである．5.2 節で述べたように，失敗テストの実行経路が短いほど SBFL は正確に欠陥箇所を推定できる．$\mathtt{rSTEFail} = 1$ と $\mathtt{rCEFail} = 1$ における，失敗テストの実行経路の長さはそれぞれ 47.40 と 168.33 であった．では，カスタム例外期待テストの実行経路はなぜ標準/サードパーティ例外期待テストの実行経路より長くなるか考察する．開発者がカスタム例外を作成する理由の 1 つは，ビジネスロジックやワークフローに関する例外に対処するためである．このようなカスタム例外を発生させるには，ビジネスロジックやワークフローを実装したプログラムを呼び出し，カスタム例外が発生する状況を再現する必要がある．それに対し，標準/サードパーティ例外はプログラムの開発過程で頻出する一般的な例外であり，単にメソッドを不適切に呼び出す/null オブジェクトを参照するといった状況でも発生しうる．そのため，筆者らは標準/サードパーティ例外よりカスタム例外の方が，例外の発生する状況の再現が複雑と考えた．例外期待テストは，例外が発生する状況を再現し，意図した通りに例外がスローされるか検証する．よって，例外の発生する状況の再現が複雑なほど，例外が発生する状況を再現するために多くのプログラム文を実行する必要があり，実行経路は長くなる．そのため，カスタム例外の発生する状況を再現する必要のあるカスタム例外期待テストは，標準/サードパーティ例外期待テストより実行経路が長くなると筆者らは考察する．

RQ3 への回答: 失敗テストが全て標準/サードパーティ例外期待テストの時，失敗テストが全てカスタム例外期待テストの場合と比べて，SBFL の結果はより正確な傾向にある．

6. 関連研究

Francisco らは例外処理がどの程度テストされるかを Java プロジェクトを対象に調査した [11]．その結果，約 60.91%のプロジェクトで少なくとも 1 つのテストメソッドが例外処理をテストすること，テストメソッドの総数に占める例外期待テストの割合は 10%以下であること，開発者は標準/サードパーティ例外よりもカスタム例外を対象に多くテストする傾向にあることを確認した．彼らは，例外期待テストの作成により注力するべきと報告する．本研究で得られた研究結果は，例外期待テストを作成する動機の 1 つになると考える．

7. 妥当性の脅威

評価指標としてこれまでの研究で広く利用されている Rank や，Top-N を元に作成した rTop-N を利用した．ま

表 10: rCEFail/ rSTEFail と rTop-5，ave(Rank)

実欠陥	rTop-5							ave(Rank)						
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	rTop5-All	Math	Chart	Lang	Jsoup	JacksonCore	Codec	ave(Rank)-All
$\mathtt{rCEFail} = 0$	**0.33**	**0.38**	**0.40**	**0.27**	0.13	**0.23**	0.31	63.97	**16.12**	**19.24**	**78.93**	109.8	**25.88**	55.37
$0 < \mathtt{rCEFail} < 1$	0.00	—	—	—	**0.50**	—	**0.33**	**45.50**	—	—	—	**13.25**	—	**24.00**
$\mathtt{rCEFail} = 1$	**0.33**	—	—	—	—	—	**0.33**	86.50	—	—	—	—	—	86.50
$\mathtt{rSTEFail} = 0$	0.30	**0.38**	0.35	**0.29**	0.13	**0.23**	0.29	70.58	**16.12**	19.00	79.79	104.5	**25.88**	59.51
$0 < \mathtt{rSTEFail} < 1$	0.00	—	0.50	0.00	—	—	0.25	45.50	—	**11.75**	**67.00**	—	—	34.00
$\mathtt{rSTEFail} = 1$	**0.67**	—	**0.67**	—	**1.00**	—	**0.70**	**6.08**	—	25.83	—	**1.00**	—	**11.50**

人工欠陥	rTop-5							ave(Rank)						
	Math	Chart	Lang	Jsoup	JacksonCore	Codec	rTop5-All	Math	Chart	Lang	Jsoup	JacksonCore	Codec	ave(Rank)-All
$\mathtt{rCEFail} = 0$	0.23	**0.52**	**0.39**	**0.51**	**0.26**	0.53	0.49	38.88	**9.23**	**16.75**	**33.08**	32.24	29.38	23.42
$0 < \mathtt{rCEFail} < 1$	0.37	—	—	—	0.13	0.39	0.36	42.92	—	—	—	**16.17**	62.96	51.99
$\mathtt{rCEFail} = 1$	**0.54**	—	—	—	—	**1.00**	**0.59**	**11.28**	—	—	—	—	**1.50**	**10.33**
$\mathtt{rSTEFail} = 0$	0.27	**0.52**	0.49	**0.51**	**0.26**	0.48	0.52	38.62	**9.23**	27.36	33.35	32.65	41.07	29.76
$0 < \mathtt{rSTEFail} < 1$	0.39	—	0.34	0.00	0.13	**0.85**	0.34	**9.25**	—	11.74	**7.00**	**15.75**	**3.23**	11.67
$\mathtt{rSTEFail} = 1$	**0.50**	—	**1.00**	0.00	—	—	**0.80**	11.50	—	**2.43**	13.50	—	—	**5.35**

た，実験対象として Defects4J と，ミューテーションツールを用いて作成した人工欠陥を利用した．異なる評価指標や実験対象を利用した場合には本研究と異なる結果が得られる可能性がある．

8. おわりに

本研究では，例外期待テストと非例外期待テストが SBFL に与える影響を調査した．例外期待テストとはプログラムの例外処理を検査するテストである．実欠陥と人工欠陥に対し実験を行い，失敗テストが全て例外期待テストの場合，失敗テストに例外期待テストが全く含まれない場合と比べ SBFL が正確に欠陥箇所を推定できることを明らかにした．また，失敗テストに占めるカスタム例外期待テスト，標準/サードパーティ例外期待テストの割合によって SBFL の正確さに与える影響が異なるか調査し，失敗テストが全て標準/サードパーティ例外の場合に特に SBFL の結果が正確であることを確認した．本研究の結果から，失敗テストに占める例外期待テストの割合によって開発者は SBFL の結果が正しい傾向にあるか判断できるようになり，デバッグ作業のさらなる効率化が期待される．

SBFL は自動欠陥修正 (Automated Program Repair, APR) でも活用される技術である [12][18][19]．これまでの研究により，欠陥限局は APR の有効性に影響を与えることが知られている [20]．そのため，今後の研究として，例外期待テストが APR の有効性に与える影響の調査が挙げられる．また，本研究では失敗テストの個数や，欠陥が潜むプログラム文の種類別の分析ができておらず，今後の課題として挙げられる．

謝辞 本研究は JSPS 科研費（JP20H04166, JP21K18302, JP21H04877, JP21K11829, JP22H03567, JP22K11985）の助成を得て行われた．

参考文献

[1] Hailpern, B. and Santhanam, P.: Software debugging, testing, and verification, *IBM Systems J.*, Vol. 41, No. 1, pp. 4–12 (2002).

[2] Britton, T., Jeng, L., Carver, G. and Cheak, P.: Quantify the time and cost saved using reversible debuggers, Technical report, Cambridge Judge Business School (2012).

[3] Jin, W. and Orso, A.: BugRedux: Reproducing field failures for in-house debugging, *Proc. Int'l Conf. on Software Engineering*, pp. 474–484 (2012).

[4] Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S. P.: SOBER: Statistical Model-Based Bug Localization, *Proc. European Software Engineering Conf. Held Jointly with Int'l Symposium on Foundations of Software Engineering*, pp. 286–295 (2005).

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

[6] Ali, S., Andrews, J. H., Dhandapani, T. and Wang, W.: Evaluating the Accuracy of Fault Localization Techniques, *Proc. Int'l Conf. on Automated Software Engineering*, pp. 76–87 (2009).

[7] Parnas, D. and Würges, H.: Response to Undesired Events in Software Systems., *Proc. Int'l Conference on Software Engineering*, pp. 437–446 (1976).

[8] Garcia, A. F., Rubira, C. M., Romanovsky, A. and Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software, *J. of Systems and Software*, Vol. 59, No. 2, pp. 197–222 (2001).

[9] Bernardo, R. D., Sales Jr., R., Castor, F., Coelho, R., Cacho, N. and Soares, S.: Agile Testing of Exceptional Behavior, *Proc. Brazilian Symposium on Software Engineering*, pp. 204–213 (2011).

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

[11] Dalton, F., Ribeiro, M., Pinto, G., Fernandes, L., Gheyi, R. and Fonseca, B.: Is Exceptional Behavior Testing an Exception? An Empirical Assessment Using Java Automated Tests, *Proc. Int'l Conf. on Evaluation and Assessment in Software Engineering*, pp. 170–179 (2020).

[12] 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, *Proc. Asia-Pacific Software Engineering Conf.*, pp. 697–698 (2018).

[13] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *Proc. Int'l Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[14] Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D. and Zhang, L.: Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach, *Proc. Int'l Symposium on Software Testing and Analysis*, pp. 75–87 (2020).

[15] Li, X., Li, W., Zhang, Y. and Zhang, L.: DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization, *Proc. Int'l Symposium on Software Testing and Analysis*, pp. 169–180 (2019).

[16] 入山 優：SBFL の精度向上を目的とした実行経路に基づくテストのグループ化，修士論文，大阪大学 (2023).

[17] Kochhar, P. S., Xia, X., Lo, D. and Li, S.: Practitioners' Expectations on Automated Fault Localization, *Proc. Int'l Symposium on Software Testing and Analysis*, pp. 165–176 (2016).

[18] Martinez, M. and Martin, M.: Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor, *Proc. Int'l Symposium on Search Based Software Engineering*, pp. 65–86 (2017).

[19] Yuan, Y. and Banzhaf, W.: ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming, *Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067 (2020).

[20] Qi, Y., Mao, X., Lei, Y. and Wang, C.: Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques, *Proc. Int'l Symposium on Software Testing and Analysis*, pp. 191–201 (2013).