7K-08

フォルトローカライゼーションの精度向上に関する研究

謝 佳智 †　　名倉 正剛 ‡　　高田 眞吾 †
† 慶應義塾大学　　‡ 南山大学

1 はじめに

フォルトローカライゼーション技術は，プログラム中に含まれる不具合の原因箇所を推定する技術である．本研究ではそのうちの Spectrum-Based Fault Localization を対象に，推定精度を向上させる手法を提案する．

2 SBFL と原因箇所特定に関する課題

2.1 Spectrum-Based Fault Localization (SBFL)

SBFL は，テストケースによる実行経路情報を利用するフォルトローカライゼーション技術である [1]．テストケース実行によるテスト結果と実行経路情報から，各ステートメントに対して疑惑値を算出し，その疑惑値により不具合の原因箇所を推定する．疑惑値の算出方法にはさまざまなものが提案されているが，一例として原因箇所特定性能が優れているとされる Ochiai の計算式では次のように定義される [2].

$$Suspiciousness(s) = \frac{Fail(s)}{\sqrt{TotalFail \times (Fail(s) + Pass(s))}}$$

ここで，$TotalFail$ は失敗テストケースの総数を表し，あるステートメント s を実行するテストケースのうち，$Fail(s)$ が失敗した数を，$Pass(s)$ が成功した数を表す．

表 1 に，Ochiai を利用した場合の疑惑値算出と，それに基づく原因箇所特定の例を示す（表 1 (a) 部分）.

表 1: Ochiai による疑惑値算出と原因箇所特定の例

#	プログラムコード	(a) test1 (x= 1)	(a) test2 (x=-1)	(a) susp	(b) test1 (x= 1)	(b) test2 (x=-1)	(b) test3 (x=-1) (x= 1)	(b) susp
1	`int closeToZero(int x) {`							
2	`  int a = 0;`	✓	✓	0.7	✓	✓	✓	0.8
3	`  if (x < 0) {`	✓	✓	0.7	✓	✓	✓	0.8
4	`  a = x + 1;`		✓	0.0		✓	✓	0.5
5	`  } else if (x > 0) {`	✓		1.0	✓		✓	1.0
6	`  a = x + 1;`	✓		1.0	✓		✓	1.0
	`  //↑本来は a = x - 1`							
7	`  }`							
8	`  return a;`	✓	✓	0.7	✓	✓	✓	0.8
9	`}`							
	期待する返り値	0	0		0	0	0, 0	
	得られた返り値	2	0		2	0	0, 2	
	テスト結果	NG	OK		NG	OK	NG	

この例では，2 列目に示した Java プログラムコードの 6 行目に不具合箇所が存在している．3，4 列目は，それぞれ個別のテストケースとそれに対する実行経路を表しており，各行が実行されたかどうかを ✓ で示している．これら 2 つのテストの結果から Ochiai の計算式により算出した疑惑値を 5 列目に示している．この例では，6 行目の疑惑値が最も高く，不具合原因の可能性が高いことを示している．

2.2 原因箇所特定に関する課題

SBFL により原因箇所特定を実施する際には，一般的な単体テストフレームワークを利用し，各テストケースに対する実行成否をアサーションで判別する．ソースコード 1 に JUnit* でのテストケースの例を示す．

ソースコード 1: テストケースの例

```
void test1() { // テストケース#1
    assertEquals(closeToZero(1), 0);
}
void test2() { // テストケース#2
    assertEquals(closeToZero(-1), 0);
}
```

この例では，表 1 の 3，4 列目に示した 2 つのテストケースを，それぞれ `test1()`, `test2()` メソッドとして定義している．しかしテストケースをどのように構成するかはテストケース作成者に依存するため，この例のように各テストケースに 1 つのアサーションしか存在しないという保証はない．このことが，SBFL による原因箇所特定に影響を与える．

例えばソースコード 2 のように，`test1()` メソッドと `test2()` メソッドの アサーションを両方実行する `test3()` というテストケースが存在することを考える．この場合 `assertEquals(closeToZero(-1), 0)` の実行は成功し，`assertEquals(closeToZero(1), 0)` の実行は失敗する．その結果として，`test3()` メソッドが示すテストケースの実行は失敗する．表 1 (a) と同様に実行経路を求め，このメソッドが追加された場合の疑惑値を算出すると，`test3()` メソッドは `test1()` メソッドと `test2()` メソッドが実行する実行経路すべてを実行する（表 1 (b)）．したがってそれらの実行経路に含むすべての行が，失敗テストケースにより実行されたことになる．この結果，本来不具合に関係しない行（4 行目）も失敗テストケースの実行に関係する行として扱われ，疑惑値が高く算出される†.

*https://junit.org/junit5/

†この例では紙面の都合上単純なケースしか記載していないが，このようなテストケースが数多く存在することで，実際の不具合箇所ではない場所の疑惑値が一番高く算出されることもある．

A Study on Improving Accuracy of Fault Localization
Yoshitomo Sha†, Masataka Nagura‡, Shingo Takada†
†Keio University, ‡Nanzan University

ソースコード 2: 複数アサーションを含むテストケース

```
void test3() { // テストケース#3
    assertEquals(closeToZero(-1), 0);
    assertEquals(closeToZero(1), 0);
}
```

3 提案手法

本研究では，2.2 節で述べた課題を解決するため，複数アサーションを含むテストケースを分割した上で疑惑値を算出し，フォルトローカライゼーションを行う手法を提案する．基本的なアイデアとしては 2.2 節の `test3()` のようなテストケースが存在するときに，疑惑値の算出前にアサーションごとにテストケースを分割しておく．そして分割したテストケースを利用し単体テストを実行した結果に基づき，疑惑値を算出する．

2.2 節で述べた `test3()` の例では，アサーションごとに `test1()` と `test2()` の 2 つのテストケースに分割することで表 1（a）で記述した例と同様の状態（結果的に `test1()` と `test2()` に相当するテストケースが 2 つずつ存在する状態）になり，原因箇所を適切に実施できる．しかし複数のアサーションを含み実行に失敗するテストケースが，この例のように成功するアサーションと失敗するアサーションを均等に含むとは限らない．実行が失敗する 1 つのアサーションと，成功する多数のアサーションを含むテストケースが存在することもあり，前述のアイデアに従ってテストケースを分割すると，成功するアサーションを含むテストケースが多数生じることになる．疑惑値によるフォルトローカライゼーションにおいて，テストケースの不均等な冗長性は，精度に悪影響を与える可能性があることが報告されている [3]．テストケースを分割しない状態よりも精度が低下することを避けるため，各テストケースによる疑惑値に対する影響の重みを変えないように，次の方針でテストケースを分割する．

1. 失敗，成功するアサーションが混在するテストケースについては，失敗または成功するアサーションだけをまとめた 2 つのテストケースを生成する．
2. 失敗，または成功するアサーションのみを含むテストケースについては，複製することで同一の 2 つのテストケースを生成する．

これによりテストケースの総数は 2 倍に増えるが，もともと失敗，または成功するアサーションのみを含んでいたテストケースについては，それらがテストケース総数に占める割合を変えないようにする．そして生成したテストケース群を利用し，既存手法と同様に疑惑値を算出しフォルトローカライゼーションを実施する．

4 評価実験

有効性を評価するため，フォルトローカライゼーションツール GZoltar* に対し提案手法を実装した．そして公開バグデータセット Defects4J† を利用し評価実験を行った．Defects4J は，6 つのオープンソース Java プロジェクトから収集したバグを含む．ここではそのうちの Chart プロジェクトに関する 26 個のバグのうち，1 行のコードのみに起因する 7 個のバグを対象にフォルトローカライゼーションを実施した場合に，バグ箇所のステートメントが疑惑値によるランキングの上位何位になっていたかを評価した．提案手法の適用有無でどのように変化したかを表 2 に示す．このように，4 個のバグでランキングが高くなった．ランキング順にバグの原因箇所かどうかを確認する場合，それらについては効率的に原因箇所を確認できるようになる．

表 2: 提案手法の適用有無による比較

バグ ID	対象プログラムのステートメント数	バグ箇所のランキング結果	
		提案手法適用無	提案手法適用有
Chart1	7057	35	34
Chart8	194	3	3
Chart9	915	66	14
Chart12	4337	14	13
Chart13	886	49	49
Chart20	517	5	2
Chart24	25	3	3

5 結論

本研究では，テストケースをアサーション実行成否により分割することで，フォルトローカライゼーションの推定精度を向上する方法を提案した．そして実験の結果，精度を向上できる可能性があることを確認した．

謝辞 本研究の成果の一部は，科研費基盤研究 (C)17K00110，2019 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] W.E. Wong, R. Gao, et al. A survey on software fault localization. *IEEE Trans. on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

[2] R. Abreu, P. Zoeteweij, et al. An evaluation of similarity coefficients for software fault localization. *2006 12th Pacific Rim Int'l Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, 2006.

[3] D. Hao, L. Zhang, et al. Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study. *21st IEEE Int'l Conf. on Software Maintenance (ICSM'05)*, pp. 683–686, 2005.

*http://www.gzoltar.com/
†http://program-repair.org/defects4j-dissection/