欠陥限局および自動プログラム修正のための自動テスト生成と実行経路を用いたテスト選択

入山 優[1,a)]　肥後 芳樹[1,b)]　楠本 真二[1,c)]

概要：デバッグ作業の負担を軽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の欠陥箇所を特定する技術である．Spectrum-Based Fault Localization（以下 SBFL）は欠陥限局の手法の 1 つで，各テストケースの成否と実行経路をもとに欠陥箇所を特定する．SBFL の精度はテストケースの品質に影響され，テストケースを追加することで SBFL の精度は向上する．しかし，SBFL は自動プログラム修正にも利用されるため，無作為にテストケースを追加するとバグ修正に時間がかかったり時間切れにより修正できなかったりする可能性がある．そのため SBFL に役立つテストケースのみを利用するべきである．本研究では，既存のテストケースと自動生成されたテストケースから実行経路が重複しないようにテストケースを選択することで，テストの実行時間の増加を抑えつつ，欠陥限局の精度を向上させる手法を提案する．実バグのデータセット Defects4J を対象に実験を行い，提案手法により選択されたテストケースを用いることで，56.8%の欠陥において精度が向上することを確認した．テストの実行時間の増加は中央値で最大でも 7.0 秒の増加に抑えられ，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて 4.9 秒短縮できた．また提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて，より多くのバグにおいてパッチを生成できることを確認した．

1. はじめに

ソフトウェア開発において，デバッグは負担の大きい作業である．ソフトウェア開発に必要コストのうち，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]．そのためデバッグの負担を軽減するための研究が盛んに行われている．

デバッグの作業の負担を軽減する技術の 1 つとして欠陥限局という技術がある．欠陥限局とはプログラム中の欠陥箇所を特定する技術である．欠陥限局の手法は様々で，これまでに多くの手法が提案されている [3], [4], [5]．その 1 つに Spectrum-Based Fault Localization（SBFL）という手法がある．SBFL は各テストケースの成否と実行経路をもとに欠陥箇所を特定する．失敗したテストケースで実行された文は欠陥箇所の可能性が高く，成功したテストケースで実行された文は欠陥箇所の可能性が低いというアイデアに基づいている．SBFL はテスト実行により得られる情報を用いて欠陥限局を行っているため，SBFL の精度は入力として与えられるテストケースの影響を受ける．そのため SBFL の精度を向上させることを目的としてテストケースの最適化に関する研究が行われている [6], [7], [8].

また SBFL の精度は自動プログラム修正の精度にも影響する [9]．自動プログラム修正とはバグを含むプログラムとテストケースを受け取ると自動的にバグを修正する技術である．自動プログラム修正では欠陥限局の結果をもとにプログラムの修正箇所を決定するため，高い精度で欠陥を特定することは重要である．

既存のテストケースに自動生成されたテストケースを追加することで SBFL の精度は向上するが，テストの実行時間は増加してしまう．自動プログラム修正では，プログラムが生成されるたびに欠陥が修正されたかをテスト実行により検証する．テストの実行時間が必要以上に増加してしまうと，自動プログラム修正においてバグを修正するまでに時間がかかったり，時間切れのために修正できなかったりする．そのため SBFL の精度を向上させるテストのみを利用すべきである [10].

図 1 はテストケースの選択により SBFL の精度が向上する例である．図 1 に示されるプログラムは 6 行目が欠陥箇所である．この行は `return b;` となっているが，正しく

1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
a) m-iriyam@ist.osaka-u.ac.jp
b) higo@ist.osaka-u.ac.jp
c) kusumoto@ist.osaka-u.ac.jp

© 2022 Information Processing Society of Japan

		テストケース					疑惑値			
		t1	t2	t3	a1	a2	{t1,t2,t3}	{t1,t2}	{t1,t2,a1}	{t1,t2,a2}
		✓	✗	✓	✓	✓				
1	`public class Util {`									
2	`  float max(float a, float b){`									
3	`    if (Float.isNaN(a))`	●	●	●	●	●	0.58	0.71	0.58	0.58
4	`      return b;`	●		●		●				
5	`    else if(Float.isNaN(b))`		●		●		1.00	1.00	0.71	1.00
6	`      return b; //return a;`		●				1.00	1.00	1.00	1.00
7	`    else`									
8	`      return Math.max(a, b);`				●					
9	`  }`									
	`  ...`									
17	`  float min(float a, float b)`									
18	`    ...`					●				
19	`  }`									
20	`}`									

図 1 テストケースの選択により SBFL の精度が向上する例

は `return a;` である．`t1`，`t2`，`t3` は既存のテストケースであり，`a1` と `a2` は自動生成されたテストケースである．既存のテストケースを全て用いた場合，疑惑値が最も高くなる文は 5 行目と 6 行目である．`t1` と `t2` のみを用いた場合も同様に 5 行目と 6 行目の疑惑値が最も高くなる．`t3` を利用しなくても欠陥箇所の疑惑値の順位は変わらないので，`t3` は SBFL の精度には影響しないテストケースである．`t3` のようなテストケースを選択すると不必要に実行時間が増加してしまう．既存のテストケースである `t1` と `t2` に加えて自動生成されたテストケースである `a1` を用いた場合，疑惑値が最も高くなる文は 6 行目のみである．欠陥箇所を 1 つに絞り込むことができているので，既存のテストケースだけを用いるよりも SBFL の精度は向上している．一方で `t1`，`t2`，`a2` を用いた場合，既存のテストケース全てを用いた場合と同じ結果になる．`t3` と `a2` の実行経路は異なっているが，失敗テストの `t2` で実行される `max` メソッドに関する実行経路のみに注目すると `t3` と `a2` の実行経路は同じである．失敗テストで実行されるメソッドに関する実行経路は疑惑値に影響を与えるが，失敗テストで実行されないメソッドに関する実行経路は疑惑値に影響を与えないため，`t1`，`t2`，`t3` を用いた場合と `t1`，`t2`，`a2` を用いた場合の結果が同じになったと考えられる．`a2` のようなテストを選択しても SBFL の精度には影響を与えず，不必要に実行時間が増加してしまう．

本研究では，既存のテストケースと自動生成されたテストケースから実行経路が重複しないテストケースを選択することで，テストの実行時間の増加を抑えつつ，欠陥限局の精度を向上させる．失敗テストの選択では，既に選択されたテストケースと異なる実行経路のテストケースのみを選択する．成功テストの選択では，選択するテストケースの数を少なくするために失敗テストで実行されるメソッドに関する実行経路が重複しないようにテストケースを選択する．実バグのデータセット Defects4J を対象に実験を行い，提案手法により選択されたテストケースを用いることで，56.8%の欠陥において精度が向上することを確認した．テストの実行時間の増加は中央値で最大でも 7.0 秒の増加に抑えられ，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて 4.9 秒短縮できた．また提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて，より多くのバグにおいてパッチを生成できることを確認した．

2. Spectrum-Based Fault Localization

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．欠陥限局の手法の 1 つに，Spectrum-Based Fault Localization（SBFL）という手法がある．この手法では，欠陥を含むプログラムとテストケースを受け取ると各テストケースの成否と実行経路を収集し，その情報をもとに疑惑値を算出する．疑惑値とは欠陥箇所の可能性の高さを表す数値である．疑惑値の計算手法には様々な種類がある．Abreu らは，7 つの疑惑値の計算手法を比較し，Ochiai[11] が優れた手法であると結論付けている [12]．そのため，本研究では疑惑値の算出に Ochiai を利用する．Ochiai の計算式は以下の式で表される．

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

式中の各変数は以下の意味を表す．

s: 疑惑値計算対象のプログラム文
$susp(s)$: s の疑惑値
$fail(s)$: 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

© 2022 Information Processing Society of Japan

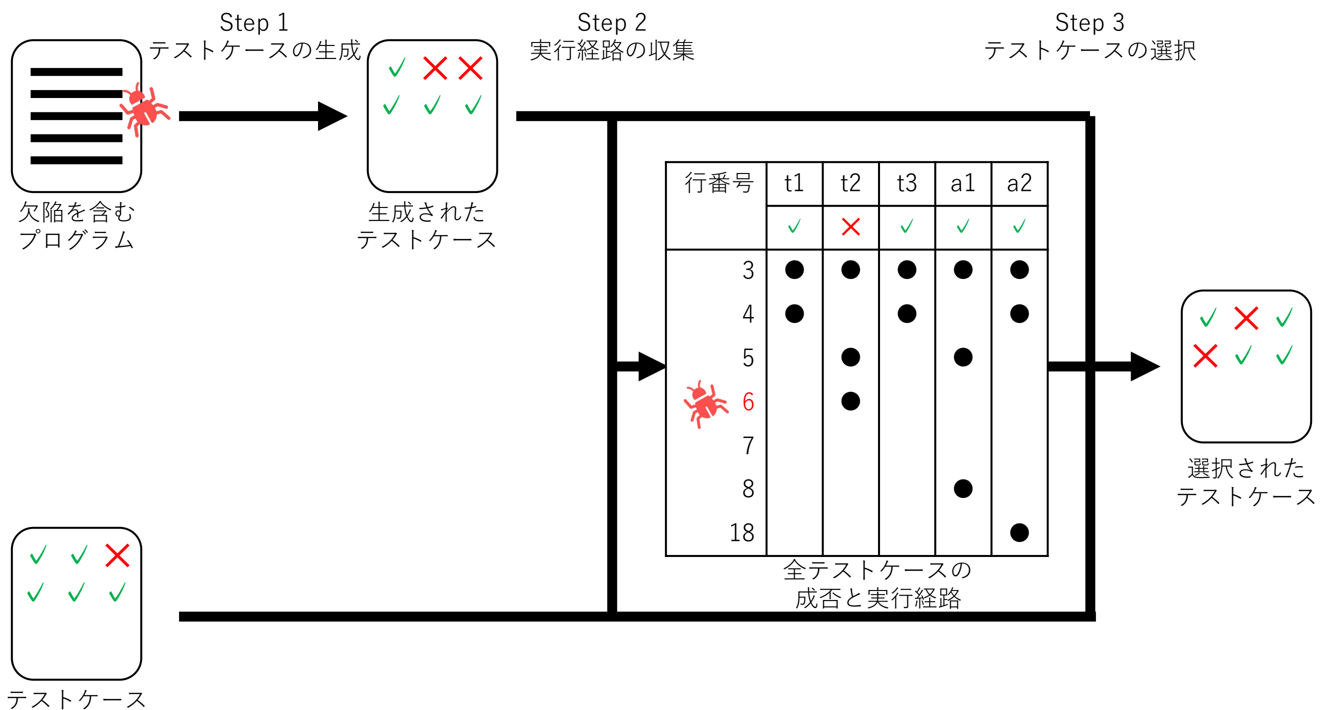

図 2　提案手法の概要

3. 提案手法

本研究では，テストの実行時間の増加を抑えつつ，SBFL の精度を向上させることを目的として，既存のテストケースと自動生成されたテストケースから実行経路が重複しないようにテストケースを選択する手法を提案する．既に選択されたテストケースと同じ実行経路のテストケースを追加しても，新しい情報を得ることができない．それに対して既に選択されたテストケースと異なる実行経路のテストケースを追加することで，各文の疑惑値を詳細に計算することができる．そのため実行経路が重複しないようにテストケースを選択することで，テストスイートの肥大化を防ぎつつ，SBFL の精度を向上させることができると考えられる．提案手法の概要を**図 2** に示す．提案手法は以下の 3 つのステップで構成されている．

Step 1 テストケースの生成
Step 2 実行経路の収集
Step 3 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 Step 1: テストケースの生成

テストケースの生成に必要なコストを削減するために，テストケース自動生成ツールを用いる．テストケース自動生成ツールによるテストケースの生成は以下の 2 つの方法が考えられる．

(1) テストケースの入力値のみを自動生成し，期待値を手動で入力する
(2) テストケースの入力値と期待値の両方を自動生成する

本研究では，テストケースの生成に必要なコストを削減するために 2 つ目の手法を用いた．この手法では正しい期待値を生成するために，テストケース自動生成ツールにバグを含まないプログラムを入力する必要がある．そのためバグが発生する直前のリビジョンのプログラムをテストケース自動生成ツールの入力として与えた．

また全クラスのテストケースを生成するのではなく，既存の失敗テストで実行されるクラスのテストケースのみを生成するようにした．これらのクラスはバグを含む可能性があるためである．

3.2 Step 2: 実行経路の収集

欠陥を含むプログラムに対して，既存のテストケースと自動生成されたテストケースを実行し，全テストケースの成否と実行経路を収集する．

3.3 Step 3: テストケースの選択

既存のテストケースと自動生成されたテストケースから実行経路が重複しないように以下の手順に従ってテストケースを選択する．

(1) 既存の失敗テストと自動生成された失敗テストに対して，以下の処理を行う．
  (a) 1 つ目の失敗テストであれば選択
  (b) 2 つ目以降の失敗テストについて，既に選択された全ての失敗テストと実行経路が異なる場合は選択
(2) 既存の成功テストと自動生成された成功テストに対し

© 2022 Information Processing Society of Japan

		テストケース							疑惑値		
		t1	t2	t3	a1	a2	a3	a4	{t1,t2,t3}	{t1,t2,a1,a3}	{t1,t2,t3,a1,a2,a3,a4}
		✓	✗	✓	✓	✓	✗	✗			
1	`public class Util {`										
2	`  float max(float a, float b){`										
3	`    if (Float.isNaN(a))`	●	●	●	●	●	●	●	0.58	0.71	0.65
4	`      return b;`	●		●		●					
5	`    else if(Float.isNaN(b))`		●		●		●	●	1.00	0.82	0.87
6	`      return b; //return a;`		●				●	●	1.00	1.00	1.00
7	`    else`										
8	`      return Math.max(a, b);`				●						
9	`  }`										
10	`  float max(float[] array){`										
11	`    ...`						●			0.71	0.58
12	`    float max = array[0];`						●			0.71	0.58
13	`    for(int j = 1; j < array.length; j++)`						●			0.71	0.58
14	`      max = max(array[j], max);`						●			0.71	0.58
15	`    return max;`						●			0.71	0.58
16	`  }`										
17	`  float min(float a, float b)`										
18	`    ...`					●					
19	`  }`										
20	`}`										
	選択するか否か（選択する=T，選択しない=F）	T	T	F	T	F	T	F			

図 3　テストケースの選択例

て，以下の処理を行う．

(a) 既に選択された全てのテストと実行経路を比較し，失敗テストで実行されるメソッドに関する実行経路が異なる場合は選択

既に選択された全ての失敗テストと異なる実行経路の失敗テストは新しい情報を持っているため選択する．失敗テストで実行されないメソッドに関する実行経路は疑惑値に影響しない．それに対して失敗テストで実行されるメソッドに関する実行経路は疑惑値に影響する．選択するテストケースの数を少なくするために，成功テストの選択では疑惑値に影響のある実行経路のみを比較する．

図 3 はテストケース選択の例である．`t1`，`t2`，`t3` は既存のテストケースであり，`a1`，`a2`，`a3`，`a4` は自動生成されたテストケースである．`a4` は失敗テストであり，`t2` と同じ実行経路なので `a4` は選択されない．`t3` は成功テストであり，`t1` と同じ実行経路なので `t3` は選択されない．`a2` は成功テストであり，同じ実行経路のテストケースは存在しない．しかし，失敗テストで実行される 2 行目から 16 行目に注目すると，`a2` は `t1` と同じ実行経路なので `a2` は選択されない．そのため図 3 の例において，提案手法により選択されるテストケースは `t1`，`t2`，`a1`，`a3` である．既存のテストケースだけを用いた場合，疑惑値が最大となる箇所は 5 行目と 6 行目だが，提案手法により選択されたテストケースを用いた場合，疑惑値が最大となる箇所は 6 行目のみである．提案手法により選択されたテストケースを用いることで欠陥箇所を 1 つに絞り込むことができているので，既存のテストケースだけを用いるよりも SBFL の精度は向上している．提案手法により選択されたテストケースを用いた場合と既存のテストケースと自動生成されたテストケースを全て用いた場合で疑惑値に差があるが，いずれの場合でも疑惑値が最大となる文は欠陥箇所の 6 行目である．

4. Research Question

提案手法を評価するために，以下の 2 つの Research Question（以下 RQ）を設定した．

RQ1 提案手法が SBFL にどのような影響があるのか

RQ2 提案手法が自動プログラム修正にどのような影響があるのか

RQ1 では，SBFL の精度やテストケースの数，テストの実行時間への影響を調査する．提案手法により選択されたテストケースを用いることでテストの実行時間の増加を抑えつつ，SBFL の精度を向上させることができるかを確認する．

RQ2 では，提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いるよりも多くのバグにおいてパッチを生成できるかどうかを確認する．

4.1 実験 1

実験 1 では，RQ1 の SBFL への影響について調査する．

5. 評価実験

提案手法の評価実験として，2 つの実験を行った．以下では各実験について説明する．

5.1 実験対象

本実験では，実バグのデータセットである Defects4J[13] に含まれるプロジェクトのうち**表 1** に示すプロジェクトを実験対象とした．Defets4J には表 1 以外のプロジェクトも

© 2022 Information Processing Society of Japan

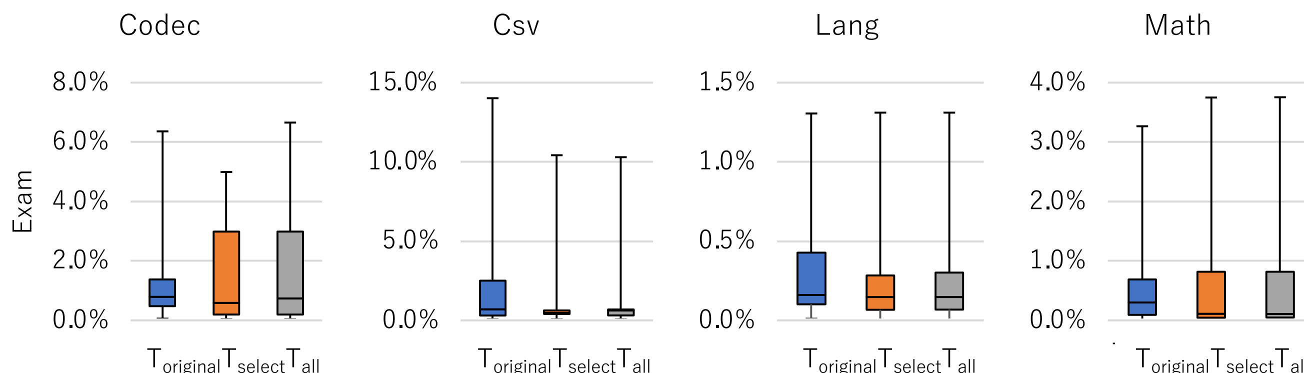

図 4　EXAM

含まれているが，著者らの環境では Java のバージョンの違いなどによって正しく動作しなかった．そのため，本研究では表 1 のプロジェクトを実験対象とした．著者らの環境では正しく動作しなかったバグは除外した．失敗テストで実行されない欠陥箇所は疑惑値が算出されない．そのような欠陥箇所は除外した．

5.1.1　実験設定

提案手法ではテストケース自動生成ツールとして Evosuite[14] を利用した．Evosuite へ与えるシード値は 1 とし，テストケースの生成の制限時間は 20 分とした．この時間内に生成された全てのテストケースを Step 1 で生成されたテストケースとして利用する．Evosuite により生成されたテストケースを用いて以下の 3 つのテストスイートを用意する．

$T_{original}$: 既存の全てのテストケース
T_{select}: T_{all} から提案手法により選択されたテストケース
T_{all}: $T_{original}$+ 生成された全てのテストケース

これら 3 つのテストスイートを用いて SBFL を適用し，SBFL の精度，テストケースの数，テストの実行時間を比較する．精度の評価指標として EXAM を用いる．EXAM は以下の式で定義される．

$$EXAM = \frac{\text{欠陥箇所の疑惑値の順位}}{\text{総ステートメント数}} \times 100\% \quad (2)$$

欠陥箇所の疑惑値の順位は，SBFL により算出された疑惑値の値が高い順に文を順位づけしたときの欠陥箇所の順位である．欠陥箇所と疑惑値の値が同じ文が複数存在する場合，欠陥箇所の疑惑値の順位はそれらの中で最も低い順位とした．この評価指標は値が小さいほど欠陥限局の精度が高いことを表す．

表 1　実験対象のプロジェクト

プロジェクト名	バグの数	欠陥箇所の行数
commons-codec（Codec）	15	37
commons-csv（Csv）	8	17
commons-lang（Lang）	36	88
commons-math（Math）	91	265

5.1.2　EXAM の比較

$T_{original}$，T_{select}，T_{all} それぞれ用いた場合の EXAM に対する箱ひげ図を**図 4** に示す．縦軸は EXAM を表している．$T_{original}$ と T_{select} の中央値を比較すると，全てのプロジェクトにおいて T_{select} の EXAM は $T_{original}$ の EXAM より小さいことが確認できる．また T_{select} と T_{all} の中央値を比較すると，Codec と Csv において T_{select} の EXAM は T_{all} の EXAM より小さくなった．

$T_{original}$，T_{select} それぞれ用いた場合の EXAM を比較し，精度が向上した欠陥箇所，精度が変化しなかった欠陥，精度が低下した欠陥の個数を示した結果が**表 2** である．全てのプロジェクトにおいて精度が低下した欠陥箇所の数より精度が向上した欠陥箇所の数は多く，過半数以上の欠陥箇所において精度が向上した．全体では，56.8%の欠陥箇所において精度が向上した．

5.1.3　テストケースの数の比較

$T_{original}$，T_{select}，T_{all} それぞれのテストケースの数に対する箱ひげ図を**図 5** に示す．T_{select} のテストケースの数は $T_{original}$ のテストケースの数より多くなったが，T_{all} のテストケースの数より少なくなった．Lang において T_{all} の中央値が 253.5 であり，T_{select} の中央値は 30 であった．Math では T_{all} の中央値が 321 であり，T_{select} の中央値は 56 であった．この 2 つのプロジェクトでは提案手法によりテストケースの数を大幅に削減できたことが確認できる．

5.1.4　テストの実行時間の比較

$T_{original}$，T_{select}，T_{all} それぞれ用いた場合のテストの実行時間に対する箱ひげ図を**図 6** に示す．T_{select} と T_{all} を比較すると，提案手法によりテストの実行時間の増加を抑制できたことが確認できる．Codec や Csv については Lang

表 2　EXAM

プロジェクト名	精度向上	変化なし	精度低下
Codec	19（51.4%）	4（10.8%）	14（37.8%）
Csv	9（52.9%）	2（11.8%）	6（34.1%）
Lang	55（62.5%）	3（3.4%）	30（34.1%）
Math	148（55.8%）	21（7.9%）	96（36.2%）
合計	231（56.8%）	30（7.4%）	146（35.9%）

© 2022 Information Processing Society of Japan

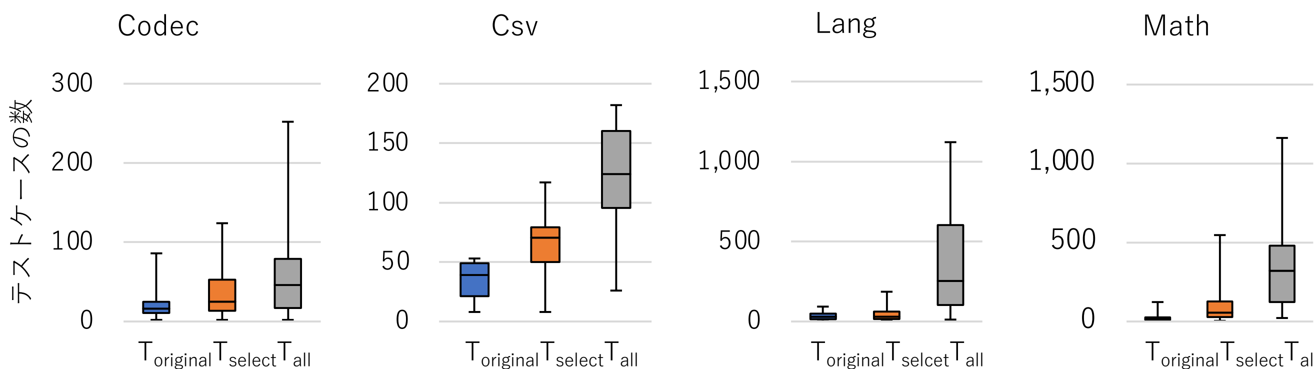

図 5 テストケースの数

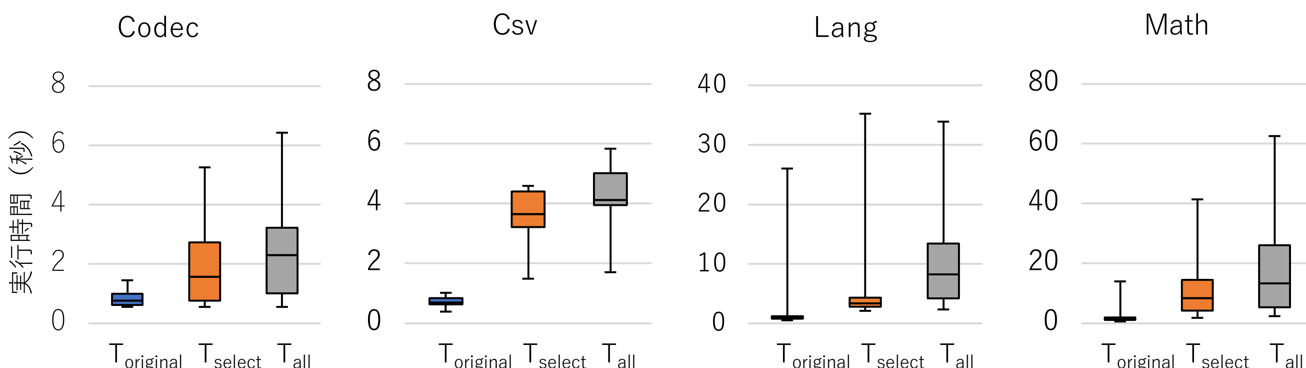

図 6 テストの実行時間

や Math ほど実行時間の増加を抑制することができなかった．T_{all} について Codec や Csv におけるテストケースの数は Lang や Math におけるテストケースの数より少なく，テストの実行時間も短い．そのため Codec や Csv では Lang や Math と比べると提案手法の有効性は小さかったと考えられる．

> **RQ1 への回答：**提案手法によりテストの実行時間の増加を抑えつつ，SBFL の精度を向上させることができた．提案手法により 56.8%の欠陥において精度が向上し，テストの実行時間の増加は中央値で最大でも 7.0 秒の増加に抑えられた．

5.2 実験 2

実験 2 では，RQ2 の自動プログラム修正への影響について調査する．

5.2.1 実験設定

実験 1 で作成した T_{select} と T_{all} を用いて，自動プログラム修正を行い，パッチ生成可能なバグの数を比較する．本実験では，自動プログラム修正ツールとして kGenProg[15] を利用した．kGenProg は遺伝的アルゴリズムを用いてバグを自動修正する GenProg[16] を Java 言語向けに再実装したツールである．制限時間は 30 分とし，この間にパッチを生成できたバグの数を比較する．kGenProg へ与えるパラメータを**表 3** に示す．その他の遺伝的アルゴリズムの動作に必要なパラメータは kGenProg のデフォルト値を用いた．

5.2.2 パッチ生成可能なバグの数の比較

T_{select} と T_{all} を用いて kGenProg を実行したときのパッチ生成可能なバグの数を**表 4** に示す．各プロジェクトのバグの数をプロジェクト名の列にある括弧の中に記述している．Codec や Lang では T_{select} と T_{all} においてパッチ生成可能なバグの数は等しくなった．Lang や Math では T_{select} を用いることで T_{all} を用いるよりも多くのバグにおいてパッチを生成することができた．しかし T_{all} を用いた

表 3 kGenProg へ与えるパラメータ

パラメータ	値
個体あたりのテストの制限時間	4 秒
最大世代数	10,000
一世代あたりの個体生存数	5
交叉によって生成される一世代あたりに個体数	0
seed 値	1

表 4 パッチ生成可能なバグの数

プロジェクト名	両方で生成可能	T_{select} のみ生成可能	T_{all} のみ生成可能
Codec（15）	0	0	0
Csv（8）	1	0	0
Lang（36）	3	1	0
Math（91）	2	4	2

© 2022 Information Processing Society of Japan

Complex.java
```
1  public class Complex implements Serializable {
2    ...
3    public boolean equals(Object other) {
4      ...
5      if (rhs.isNaN()) {
6        ret = this.isNaN();
7      } else {
8        ret = (Double.doubleToRawLongBits(real)
             == Double.doubleToRawLongBits(rhs.getReal()))
             && (Double.doubleToRawLongBits(imaginary)
             == Double.doubleToRawLongBits(rhs.getImaginary()));
         //ret = (real == rhs.real)
             && (imaginary == rhs.imaginary);
9      }
10   }
11   public Complex multiply(Complex rhs) {
12     ...
13     return createComplex(real * rhs.real
         - imaginary * rhs.imaginary,
14       real * rhs.imaginary + imaginary * rhs.real);
15   }
16   ...
17 }
```

選択されなかった既存のテストケース
```
1 public void testAtan() {
2   Complex z = new Complex(3, 4);
3   Complex expected = new Complex(1.44831, 0.158997);
4   TestUtils.assertEquals(expected, z.atan(), 1.0e-5);
5 }
```

図 7　T_{select} を用いることで SBFL の精度が低下したバグの例

場合のみパッチ生成可能なバグも存在した．

> **RQ2 への回答：**提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いるよりも，多くのバグにおいてパッチを生成できた．

6. 考察

T_{select} を用いることで $T_{original}$ を用いるよりも SBFL の精度が低下したバグについて考察する．また T_{all} を用いた場合のみパッチ生成可能なバグについても考察する．

6.1 提案手法により SBFL の精度が低下したバグ

実験 1 において T_{select} を用いることで $T_{original}$ を用いるよりも SBFL の精度が低下したバグが存在した．本節ではその原因について考察する．

T_{select} を用いることで SBFL の精度が低下したバグの例（Math の欠陥 ID96）を**図 7** に示す．$T_{original}$ を用いた場合の欠陥箇所の疑惑値の順位は 5 位であったが，T_{select} を用いた場合，13-14 行目の順位が上位にきたことで欠陥箇所の疑惑値の順位は 6 位になった．失敗テストは `equals` と `multiply` を実行し，提案手法により選択されなかった成功テスト `testAtan` は `equals` を実行せず，`multiply` を実行した．`multiply` を実行する成功テストの数が少なくなったことで，13-14 行目の疑惑値が大きくなり，SBFL の精度は低下したと考えられる．テスト選択により実行する失敗テストと成功テストの数が既存のテストスイートと大きく変化する行がある．このような行の疑惑値は大きく変化するため，SBFL の精度が向上する場合もあるが，精度が低下する場合もある．

UniformRealDistribution.java
```
1 public class UniformRealDistribution
2   extends AbstractRealDistribution {
3   ...
4   /** {@inheritDoc} */
5   public boolean isSupportUpperBoundInclusive() {
6     return false;  //return true;
7   }
8   ...
9 }
```

選択されたテストケース
```
1 @Test(expected=NumberIsTooLargeException.class)
2 public void testPreconditions2() {
3   new UniformRealDistribution(1, 0);
4 }
```

選択されなかったテストケース
```
1 @Test(expected=NumberIsTooLargeException.class)
2 public void testPreconditions1() {
3   new UniformRealDistribution(0, 0);
4 }
```

図 8　T_{all} を用いた場合のみパッチ生成可能なバグの例

6.2 T_{all} を用いた場合のみパッチ生成可能なバグ

実験 2 において T_{select} を用いた場合ではパッチを生成できなかったが，T_{all} を用いた場合ではパッチを生成できたバグが存在した．本節ではその原因について考察する．

T_{all} を用いた場合のみパッチ生成可能なバグの例（Math の欠陥 ID22）を**図 8** に示す．提案手法により `testPreconditions1` は `testPreconditions2` と実行経路が重複しているとみなされ，`testPreconditions1` は選択されなかった．自動プログラム修正では生成された個体をテスト実行により評価し，次の世代に残すべき個体を選択する．提案手法によって SBFL の精度向上に役立たないテストケースとして選択されなかったテストケースが自動プログラム修正における個体の選択に役立つ可能性がある．T_{all} を用いることで T_{select} を用いるよりも良い個体が選択され，パッチを生成できたと考えられる．

7. 妥当性の脅威

7.1 テストケースの生成

本研究では，テストケースの自動生成に Evosuite を利用した．他のテストケース自動生成ツールを用いた場合，実験結果が異なる可能性がある．また Evosuite のテストケースの生成には乱択の操作が含まれる．seed 値やテストの生成時間を変えた場合．実験結果が変化する可能性がある．

7.2 疑惑値の計算

本研究では，疑惑値の計算式に Ochiai を用いた．他の計算式を用いた場合，実験結果が変化する可能性がある．

7.3 自動プログラム修正ツール

本研究では，自動プログラム修正ツールとして kGenProg を用いた．他の自動プログラム修正ツールを用いた場合，実験結果が変化する可能性がある．また kGenProg へ与え

© 2022 Information Processing Society of Japan

るパラメータの値を変えた場合，実験結果が変化する可能性がある．

7.4 実験対象

本研究では，Defects4J に含まれるバグを実験対象とした．他のバグを対象にして実験を行なった場合，実験結果が変化する可能性がある．

8. おわりに

本研究では，テストの実行時間の増加を抑えつつ SBFL の精度を向上させるために，既存のテストケースと自動生成されたテストケースから実行経路が重複しないようにテストケースを選択する手法を提案した．実バグのデータセット Defects4J を対象に実験を行い，提案手法により選択されたテストケースを用いることで，56.8%の欠陥において精度が向上することを確認した．テストの実行時間の増加は中央値で最大でも 7.0 秒の増加に抑えられ，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて 4.9 秒短縮できた．また提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて，多くのバグにおいてパッチを生成できることを確認した．

今後の課題としては，他の自動プログラム修正ツールを用いた場合でも提案手法により選択されたテストケースを用いることで，既存のテストケースと自動生成されたテストケースを全て用いた場合と比べて，パッチを生成できるかを調査することが挙げられる．

謝辞 本研究は JSPS 科研費（JP20H04166, JP21K18302, JP21K11820, JP21H04877, JP22H03567, JP22K11985）の助成を得て行われた．

参考文献

[1] 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).

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

[3] Jin, W. and Orso, A.: F3: Fault Localization for Field Failures, *the proceedings of the International Symposium on Software Testing and Analysis*, pp. 213–223 (2013).

[4] Korel, B.: PELAS-program error-locating assistant system, *the proceedings of IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253–1260 (1988).

[5] Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S. P.: SOBER: Statistical Model-Based Bug Localization, *the proceedings of the European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 286–295 (2005).

[6] Dandan, G., Tiantian, W., Xiaohong, S. and Peijun, M.: A test-suite reduction approach to improving fault-localization effectiveness, *Computer Languages, Systems & Structures*, Vol. 39, No. 3, pp. 95–108 (2013).

[7] Wu, Y., Liu, Y., Wang, W., Li, Z., Chen, X. and Doyle, P.: Theoretical Analysis and Empirical Study on the Impact of Coincidental Correct Test Cases in Multiple Fault Localization, *IEEE Transactions on Reliability*, Vol. 71, No. 2, pp. 830–849 (2022).

[8] Perez, A., Abreu, R. and van Deursen, A.: A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches, *the proceedings of IEEE/ACM International Conference on Software Engineering*, pp. 654–664 (2017).

[9] Assiri, F. Y. and Bieman, J. M.: Fault localization for automated program repair: effectiveness, performance, repair correctness, *Software Quality Journal*, Vol. 25, No. 1, pp. 171–199 (2017).

[10] Kuma, T., Higo, Y., Matsumoto, S. and Kusumoto, S.: Improving the Accuracy of Spectrum-Based Fault Localization for Automated Program Repair, *the proceedings of International Conference on Program Comprehension*, pp. 376–380 (2020).

[11] Abreu, R., Zoeteweij, P. and Van Gemund, A. J.: An Evaluation of Similarity Coefficients for Software Fault Localization, *the proceedings of Pacific Rim International Symposium on Dependable Computing*, pp. 39–46 (2006).

[12] Abreu, R., Zoeteweij, P., Golsteijn, R. and van Gemund, A. J. C.: A Practical Evaluation of Spectrum-Based Fault Localization, *J. Syst. Softw.*, Vol. 82, No. 11, pp. 1780–1792 (2009).

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

[14] Fraser, G. and Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, *the proceedings of ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pp. 416–419 (2011).

[15] 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, *the proceedings of Asia-Pacific Software Engineering Conference*, pp. 697–698 (2018).

[16] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically finding patches using genetic programming, *the proceedings of IEEE International Conference on Software Engineering*, pp. 364–374 (2009).

© 2022 Information Processing Society of Japan