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.

開発者によるバグ限局を考慮した 自動バグ修正への影響に関する研究

山手　響介† 首藤　巧† 浅田　翔†
佐藤　亮介† 亀井　靖高† 鵜林　尚靖†

† 九州大学
E-mail: †{yamate,shuto,asada}@posl.ait.kyushu-u.ac.jp, ††{sato,kamei,ubayashi}@ait.kyushu-u.ac.jp

あらまし　ソフトウェア開発におけるデバッグコストの削減を目的として，自動バグ修正の研究が盛んに行われている．自動バグ修正ではバグの箇所を自動で推測するバグ限局と呼ばれるステップがあるが，バグの箇所を正確に特定できるとは限らない．バグ限局でバグの箇所が正しく推測できなければ自動バグ修正の精度が低下し，修正時間が増加すると考えられる．そこで開発者の直感を考慮してバグ限局を行えば，修正精度の向上および修正時間の短縮が期待できる．本研究ではバグ限局を手作業でも行うことができる自動バグ修正ツールを開発し，簡単なプログラムで初期実験を行い結果を分析した．簡単なプログラムにおいて，バグの箇所が分かっている場合は自動でバグ限局を行うよりも開発者によるバグ限局を行った方が，修正の精度が向上し理論的には修正時間が短くなることが明らかになった．
キーワード　自動バグ修正, デバッグ, バグ限局

The Impact of Fault Localization Considering Developers' Intuition on Automatic Bug Repair

Kyosuke YAMATE†, Takumi SHUTO†, Sho ASADA†,
Ryosuke SATO†, Yasutaka KAMEI†, and Naoyasu UBAYASHI†

† Kyushu University
E-mail: †{yamate,shuto,asada}@posl.ait.kyushu-u.ac.jp, ††{sato,kamei,ubayashi}@ait.kyushu-u.ac.jp

Abstract　Research on automatic program repair has been actively conducted in order to reduce debugging costs in software development. In automatic program repair, there is a step called fault localization that automatically guesses the location of the bug, but it is not always possible to accurately identify the location of the bug. If the location of the bug cannot be correctly guessed at the fault localization, the accuracy of the automatic program repair will decrease and the repair time will increase. Therefore, if fault localization is performed in consideration of the intuition of the developer, improvement in correction accuracy and reduction in correction time can be expected. In this research, we developed an automatic program repair tool that can manually identify the location of the bug, and conducted initial experiments with simple programs and analyzed the results. In a simple program, if the location of a bug is known, fault localization by the developer can improve the accuracy of repair and theoretically shorten the repair time, rather than automatic fault localization.
Key words　Automatic program repair, Debugging, Fault localization

1. はじめに

ソフトウェア開発を行う際の大切な行程の一つとしてデバッグ作業があげられる．デバッグ作業は基本的に開発者の手作業で行われる為非常にコストがかかる．デバッグ作業がソフトウェアの開発コストの 50% 以上を占めているという結果が示されている [1], [2]．この問題を解決するために自動でバグの修正を行うツールの開発，改良

が盛んに行われている．

現在提案されている自動バグ修正の手法は，バグを含むソースコードとそのテストスイートを基にバグの箇所を推測し，修正パッチを生成することでバグの修正を行う．この手法ではテストが不十分である場合や，対象のプロジェクトが大きく複雑になった場合に，バグの箇所を推測できない可能性が高くなり，自動バグ修正の精度や修正時間に悪影響を及ぼす．

そこでソフトウェアの開発者によるバグの箇所の推測結果を利用することで，自動バグ修正の精度の向上および修正時間の短縮が期待できると考えた．現在，著者らは Java を対象とした自動バグ修正ツール jProphet を開発している．本研究では jProphet にバグの箇所の推測を手作業でも行うことができる機能を組み込み，簡単なプログラムで初期実験を行い結果を分析した．

本稿では，2 節で自動バグ修正の背景と関連研究を紹介する．3 節では自動バグ修正の課題と本研究の目的について述べる．4 節で開発中の自動バグ修正ツール jProphet を紹介する．5 節で本研究で行った初期実験について述べ，6 節で初期実験の結果および考察を述べる．7 節で妥当性の脅威について述べ，最後に 8 節でまとめと今後の課題について述べる．

2. 背　　景

2.1 自動バグ修正

自動バグ修正の代表的な手法に，入力としてバグを含むプログラムとテストスイートを受け取ることでプログラムを修正する技術がある．本稿ではこの手法を対象に研究を行う．この自動バグ修正手法の修正過程は，大きく分けるとバグ限局，コード改変の 2 つに分けることができる．バグ限局ではプログラムの中からバグの原因となっている箇所を推測し，コード改変ではバグ限局によって推測された箇所を優先的に改変することでプログラムを修正する．

2.2 バグ限局

自動バグ修正におけるバグ限局は，バグを含むプログラムと，失敗するテスト，成功するテストの両方を含むテストスイートを入力として受け取り，バグの箇所を推測する技術が主流である．その中でも実行経路情報に基づいてバグ限局を行う Spectrum-Based Fault Localization [3](以降，SBFL) は近年盛んに研究されている手法の 1 つである．SBFL は失敗するテストで実行される文はバグの原因である可能性が高く，成功するテストで実行される文はバグの原因である可能性が低いという考えに基づいてバグの箇所を推測する．SBFL は各テストを実行した際に得られる実行経路情報と成否情報を基にバグを含むプログラムの各文に対して疑惑値を付ける．疑惑値とは，その文がバグの原因である可能性の高さを表す数値である．疑惑値の算出方法はこれまでに多くの手法が提案されている [3]．手法の一例を表 1 に示し，簡単なプログラムに対して Jaccard を用いて疑惑値を算出した例を図 1 に示す．図 1 の例では，8 行目の文がバグの原因である．成功するテストで 8 行目を通ったものは 1 つもなく，失敗するテストで 8 行目を通ったものは 1 つである．直感的に，8 行目の文は他の行の文に比べると失敗したテストの数の割合が多いため，バグの原因である可能性が高いことがわかる．実際に Jaccard を用いて計算しても，8 行目の疑惑値が最も高くなる．

表 1　SBFL の手法の例

名前	疑惑値の計算式
Jaccard	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
Ochiai	$\frac{N_{CF}}{\sqrt{(N_{CF}+N_{UF})\times(N_{CF}+N_{CS})}}$
Tarantula	$\frac{N_{CF}/(N_{CF}+N_{UF})}{N_{CF}/(N_{CF}+N_{UF})+N_{CF}/(N_{CS}+N_{US})}$

N_{CF} ： その行を実行した失敗したテストの数
N_{CS} ： その行を実行して成功したテストの数
N_{UF} ： その行を実行せず失敗したテストの数
N_{US} ： その行を実行せず成功したテストの数

	サンプルプログラム	テスト				疑惑値
		A	B	C	D	
1	String call(int i) {					
2	if(i % 15 == 0)	●	●	●	●	0.25
3	return "FizzBuzz";	●				0.00
4	if(i % 5 == 0)		●	●	●	0.30
5	return "Buzz";		●			0.00
6	if(i % 3 == 0)			●	●	0.50
7	return "Fizz";			●		0.00
8	return String.valueOf(0); //bug				●	1.00
9	}					
	テストの成否(成功=T，失敗=F)	T	T	T	F	

図 1　Jaccard を用いた疑惑値の算出例

2.3 関 連 研 究

SBFL における疑惑値算出の手法． Jones ら [4] は，バグの可能性がある箇所に色を付けることでデバッグ作業を支援する手法 Tarantula を提案した．C 言語で書かれたバグを含むプログラムに対してこの手法を用いることで，開発者がプログラムのバグを見つけるのに効果的であることが示されている．Ochiai や Jaccard は元々他の分野で用いられていた手法であり，Abreu らはこれら 2 つの手法をバグ限局に応用し開発者の観点から 9 種類の手法を比較して Ochiai が優れていると結論づけている [5], [6]．

SBFL の結果が自動バグ修正に与える影響． Assiri ら [7] は，SBFL によるバグ限局でバグの箇所が推測できなかっ

た場合，自動バグ修正の性能が制限されることを示している．また，10 種類の手法を比較して正しい修正パッチの生成率が最も高いものは Jaccard であると結論づけている．

Qi ら [8] は，開発者の観点で優れているとされている SBFL の手法は自動バグ修正において良好なパフォーマンスを発揮しないことを示している．また，自動バグ修正の観点から 14 種類の手法を比較して Jaccard が優れていると結論づけている．本研究では自動バグ修正におけるバグ限局で SBFL を用いるため，疑惑値算出の手法として Jaccard を用いる．

目視評価を組み合わせたモジュールのランク付け．

笠井ら [9] は，不具合を含む可能性の高いモジュールから順に受入れ検査を実施することを目的とし，判別モデルによる結果と目視評価とを組み合わせたモジュールのランク付け手法を提案しており，その手法の有効性を示している．受入れ検査とは，ソフトウェアの開発を外部委託する際に，委託者が完成したソフトウェアに問題がないか確認することである．受け入れ検査時に不具合を多く含むモジュールから順にテストを実施できれば品質向上や不具合の修正工数削減につながる．自動バグ修正においても同様に，目視によるバグ限局を考慮することで，修正精度の向上が期待できる．

3. 動　　機

3.1 自動バグ修正の課題

自動バグ修正におけるバグ限局で全てのバグを特定できるとは限らない．バグの箇所が特定できなかった場合，修正の失敗や修正時間の増加につながると考えられる．

3.2 本研究の目的

開発者がバグの箇所をある程度把握している場合，その情報をバグ限局の結果に反映することで，短時間かつ正確にバグを修正できると考えられる．本研究では手作業でバグ限局を行う機能を組み込みんだ自動バグ修正ツールを開発し，初期実験を行う．初期実験ではバグの箇所が判明している前提でバグ限局を行い，開発者の直感が当たっている場合の自動バグ修正に与える影響を調査する．

4. jProphet

本節では，現在著者らで開発している自動バグ修正ツール jProphet を紹介する．jProphet は Long らが提案した自動バグ修正技術である Prophet [10] を参考に開発されており，対象の言語は Java である．jProphet は入力として，バグを含むプログラムと失敗するテストを含むテストスイートを受け取り，修正パッチを出力することでプログラムの修正を行う．jProphet の修正過程を図 2 に示す．

表 2　jProphet で用いるテンプレート

名前	操作内容
Condition Refinement	if 文の条件式の書き換え
Condition Introduction	対象の文を if 文で挟み込む
Control Flow Introduction	コントロールフロー制御文を追加
Method Replacement	関数の置換および引数の置換
Variable Replacement	対象の文の中の変数などの置換
Copy and Replace	文の複製および変数の置換

4.1 バグ限局

jProphet ではバグ限局を SBFL と手作業の 2 つの手法に切り替えて実行することができる．

SBFL. SBFL でのバグ限局は入力としてバグを含むプログラムとそのテストスイートを受け取り，それぞれのテストの成否と実行経路情報を用いて，各行の疑惑値を算出する．疑惑値算出の計算式は 2.2 節の表 1 で示したものから選択することができる．

手作業. 手作業でのバグ限局では，ファイルパスと行番号，付与したい疑惑値を入力することで任意の箇所を任意の疑惑値にすることができる．

4.2 修正パッチ候補生成

対象ソースコードから抽象構文木 (AST) を取得し，各ノードに対しテンプレートを適用して修正パッチの候補を生成する．jProphet で用いるテンプレートを表 2 に示す．現状ではバグ限局の結果にかかわらず，対象ソースコードの全ての文に対して修正パッチ候補を生成する．

4.3 修正パッチ候補の並び替え

バグ限局で算出された疑惑値の高い行を修正する修正パッチ候補が上位に来るように並び替える．~~現状では~~同率順位の修正パッチ候補は生成された順に並ぶようになっている．

4.4 テスト実行による修正パッチ候補の検証

並び替えられた修正パッチ候補を上位から順に修正対象のプログラムに適用し，入力として受け取った~~失敗するテストを含む~~テストスイートを実行する．全てのテストが成功するようなパッチが見つかった時点で修正パッチ候補の検証を終了し，そのパッチを修正パッチとして出力する．

5. 初 期 実 験

初期実験として，jProphet を用いて小さなプログラムを修正することで手作業によるバグ限局が有効であるかを調査する．本節では初期実験で用いるデータセットおよび実験手順について説明する．

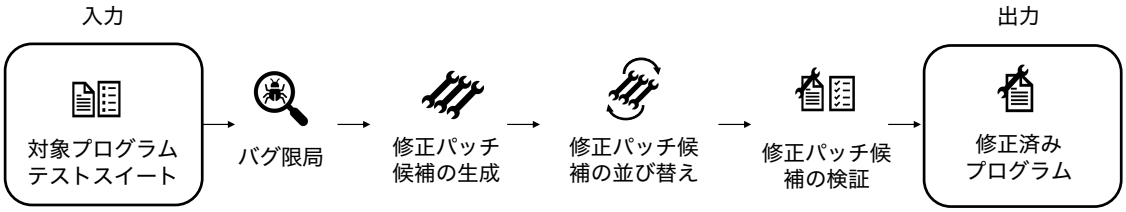

図 2　jProphet の修正過程

5.1　データセット

修正対象のプログラムとして，二分探索のプログラムを 1 行書き換えることによってバグを埋め込んだものを用意する．本研究で用いるバグを埋め込む前の二分探索のプログラムは，入力としてソート済みの整数の配列と探索したい整数一つを受け取り，配列の中から探索したい整数を探し出し，見つかればその配列のインデックス番号を，見つからなかったら-1 を出力する．バグを埋め込んだ二分探索のプログラムをプログラム 1 に示す．テストスイートには，成功するテスト 3 つと失敗するテスト 1 つを用意する．テストスイートの文網羅は 100 %である．文網羅とは，プログラムの全ての文のうちテストで実行された文の割合である．テストの詳細を表 3 に示す．SBFL では疑惑値の算出に用いる計算式の 1 つである Jaccard [3] を用いて疑惑値を算出する．バグの修正には jProphet を用いるが，修正パッチ候補生成における 2 つのテンプレート Condition Refinement および Condition Introduction で生成されるパッチを適用してテスト実行による検証を行うと，実装の問題により正常に動作しなかったため，これら 2 つのテンプレート以外で生成される修正パッチ候補のみを対象として実験を行った．

5.2　実 験 手 順

バグ限局の手法を SBFL，手作業，SBFL と手作業の両方を用いる方法の 3 通りに切り替えて修正を行う．それぞれの手法で生成された修正パッチ候補のランクと出力された修正パッチの内容を比較する．SBFL と手作業の両方を用いるバグ限局では，各行に対してそれぞれの手法で疑惑値を算出し，値の大きい方をその行の疑惑値とする．手作業によるバグ限局は以下の 3 通りの方法で行う．

・バグの箇所一点の疑惑値を最大にする．
・バグの箇所を含む複数行の疑惑値を最大にする．（本実験ではバグの箇所を含む if 文全体を指定）
・ある行 x の疑惑値を最大にし，その前後 y 行の疑惑値を変化させる．ただし前後 y 行の中，もしくは行 x にバグの箇所を含むように x，y をとる．（本実験では y=2 とし，行 x から 1 行離れるごとに最大値 1 から 0.2 ずつ小さい値を付ける）手作業によるバグ限局の例を図 3 に示す．図 3 における指定方法は以下の通りである．

A：バグの箇所一点の疑惑値を最大にする．
B：バグの箇所を含む複数行の疑惑値を最大にする．
C：ある行 x の疑惑値を最大にし，その前後 y 行の疑惑値を変化させる．(x=4 行目の文，y=2)
D：ある行 x の疑惑値を最大にし，その前後 y 行の疑惑値を変化させる．(x=6 行目の文，y=2)

プログラム 1　バグを埋め込んだ BinarySearch ソースコード

```
int search(int[] list, int target) {
    int l = 0;
    int r = list.length;
    int index = -1;
    while (l < r) {
        int m = (l + r) / 2;
        if (list[m] == target) {
            index = l; //index = m;
            break;
        } else if (list[m] > target) {
            r = m;
        } else if (list[m] < target) {
            l = m + 1;
        }
    }
    return index;
}
```

表 3　初期実験で用いるテストスイート

名前	整数の配列	探索する整数	想定される出力	テストの成否
テスト 1	[1, 2, 3, 4, 5]	4	3	成功
テスト 2	[1]	1	0	成功
テスト 3	[1, 2, 3, 4, 5]	6	-1	成功
テスト 4	[1, 2, 3, 4]	2	1	失敗

6.　実験結果と考察

6.1　結　　果

SBFL でのバグ限局によって算出された各行の疑惑値をプログラム 2 に示す．本実験で用意したバグを埋め込んだ二分探索のプログラムに対して SBFL でバグ限局を

サンプルプログラム		指定方法			
		A	B	C	D
1	String call(int i) {				
2	if(i % 15 == 0)	0.0	0.0	0.6	0.0
3	return "FizzBuzz";	0.0	0.0	0.8	0.0
4	if(i % 5 == 1) //bug	1.0	1.0	1.0	0.6
5	return "Buzz";	0.0	1.0	0.8	0.8
6	if(i % 3 == 0)	0.0	0.0	0.6	1.0
7	return "Fizz";	0.0	0.0	0.0	0.8
8	return String.valueOf(i);	0.0	0.0	0.0	0.6
9	}				

図 3　手作業によるバグ限局の例

プログラム 2　SBFL によるバグ限局で算出された各行の疑感値

```
int search(int[] list, int target) {
    int l = 0;                                  0.25
    int r = list.length;                        0.25
    int index = -1;                             0.25
    while (l < r) {                             0.25
        int m = (l + r) / 2;                    0.25
        if (list[m] == target) {                0.25
            index = l; //index = m;             0.30
            break;                              0.30
        } else if (list[m] > target) {          0.30
            r = m;                              0.50
        } else if (list[m] < target) {          0.00
            l = m + 1;                          0.00
        }
    }
    return index;                               0.25
}
```

行うと，バグの箇所でない行の疑感値の方が高くなった．

実際に生成された 2 種類のパッチ A，B をプログラム 3，4 に示す．パッチ A は想定していた正しい修正を行うパッチで，パッチ B は正しく動作するが無駄のある修正を行うパッチである．

3 通りの手法でバグ限局を行って修正を行った結果を表 4 に示す．手作業によるバグ限局と，SBFL+ 手作業によるバグ限局での修正結果は全ての疑感値の指定方法において同じであった．SBFL でバグ限局を行った場合 B のパッチが修正パッチとして出力されたが，手作業および SBFL+ 手作業でバグ限局を行った場合は，7 つの指定方法のうち 3 つで B のパッチ，4 つで A のパッチが修正パッチとして出力された．また，手作業でのバグ限局で B のパッチが出力された 3 つの指定方法のうち 2 つでは，修正パッチ候補として生成された A のパッチのランクが SBFL よりも高かった．

6.2　考　　察

SBFL でバグの箇所以外の疑惑値が最大になる条件とは．

プログラム 3　jProphet が生成したパッチ A

```
        int m = (l + r) / 2;
        if (list[m] == target) {
-           index = l;
+           index = m;
            break;
```

プログラム 4　jProphet が生成したパッチ B

```
        int m = (l + r) / 2;
        if (list[m] == target) {
            index = l;
+           index = m;
            break;
```

表 4　実 験 結 果

	疑感値の指定方法	出力された修正パッチ	正しいパッチの順位
SBFL	-	B	96
手作業	1 箇所を指定	A	29
	複数行を指定	B	86
	2 行前の前後を変化	A	88
	1 行前の前後を変化	A	77
	バグの前後を変化	A	29
	1 行後の前後を変化	B	60
	2 行後の前後を変化	B	96
SBFL + 手作業	1 箇所を指定	A	29
	複数行を指定	B	86
	2 行前の前後を変化	A	88
	1 行前の前後を変化	A	77
	バグの前後を変化	A	29
	1 行後の前後を変化	B	60
	2 行後の前後を変化	B	96

本研究で用いたバグを埋め込んだ二分探索のプログラムにおいて，表 3 のテスト 1，テスト 2，テスト 4 のように探索したい要素が配列の中にあるようなテストを実行した場合，バグの箇所が必ず実行される．よって探索したい要素が配列の中にあり，かつ成功するようなテストが増えればバグの箇所の疑感値が下がる．二分探索のプログラムに限らず，このように一定の条件を満たすプログラムとテストスイートを入力として受け取ると，SBFL によるバグ限局が失敗する場合がある為，バグの箇所の目処がついていれば手作業によるバグ限局が有効であると考えられる．

正しいが無駄のある修正パッチが出力される条件とは．

本実験で出力された正しいが無駄のある修正パッチは，本

来修正されるべき文の 1 行後の文に対する修正パッチであった．使用されたテンプレートは Copy and Replace である．Copy and Replace は，対象の文の直前に，Variable Replacement を適用した他の文を挿入する操作を行う．プログラム 2 より，バグの箇所の疑惑値とその 1 行後の疑惑値が等しいことがわかる．現状の jProphet において，疑惑値が等しい修正パッチ候補は生成された順番に並べられる．本実験では，バグの箇所を修正するパッチよりも 1 行後の文を修正するパッチの方が先に生成されたため，正しいが無駄のある修正パッチが出力された．二分探索のプログラムに限らず，バグの箇所の疑惑値が他の箇所以下である時は，想定しているパッチが出力されない可能性が高くなると考えられる．

7. 妥当性への脅威

内的妥当性．本研究では，開発者がプログラム内のバグの位置を把握している前提で実験を行った．実際にそのような状況がどれくらいあるのかは明らかではない．開発者によるバグ限局でバグの箇所が推測できなかった場合の修正結果も含めて今後調査する必要がある．

外的妥当性．本研究では二分探索のプログラムを実験対象にした．他の探索プログラムや一般の大きな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．

8. まとめと今後の課題

本研究では，手作業でのバグ限局を行う機能を組み込みんだ自動バグ修正ツールを開発し，小さなプログラムで初期実験を行うことで，手作業でのバグ限局が自動バグ修正の結果へ及ぼす影響について調査した．SBFL でのバグ限局で修正を行うと正しいパッチが出力されないが，手作業によるバグ限局で修正を行うと正しいパッチが出力される場合があることが確認できた．また，バグ限局に SBFL を用いた場合より手作業を用いた場合の方が正しい修正パッチのランクが高かった．

今後の課題として，いくつか追加で実験をする必要があると考えている．本稿では，小さなプログラム 1 つを対象とし，バグの箇所が明らかである状態で初期実験を実施した．一方，一般的な大きなプログラムでの実験や，バグの箇所が明らかである状況がどれくらいあるかの調査ができていないため，今後の課題となっている．また，jProphet には確率モデルを実装する予定である．確率モデルとは，OSS から収集した修正履歴を基に修正パッチ候補にスコアを付ける機能である．確率モデルによるスコア付与はバグ限局とは別で行われ，修正パッチ候補の並び替えの際にそれぞれの結果を組み合わせてランク付けを行う．確率モデルを実装することで，修正パッチ候補の並び順が変わると考えられるので追加で実験を行う必要がある．

謝　　辞

本研究の一部は JSPS 科研費 18H04097・18H03222，および，JSPS・国際共同研究事業の助成を受けたものです．

文　　献

[1] James S. Collofello and Scott N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, Vol. 9, No. 3, pp. 191 – 195, 1989.

[2] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67, Jan 2019.

[3] 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.

[4] James Jones, Mary Harrold, and John Stasko. Visualization of test information to assist fault localization. pp. 467– 477, 02 2002.

[5] Rui Abreu, Peter Zoeteweij, and Arjan Gemund. On the accuracy of spectrum-based fault localization. pp. 89–98, 10 2007.

[6] 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. ~~SI: TAIC PART 2007 and MUTATION 2007.~~

[7] Fatmah Assiri and James Bieman. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal*, Vol. 25, , 03 2016.

[8] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. pp. 191–201, 07 2013.

[9] 笠井則充, 森崎修司, 松本健一. 目視評価と判別モデルを組み合わせた fault-prone モジュールのランク付け手法. 情報処理学会論文誌, Vol. 53, No. 9, pp. 2279–2290, sep 2012.

[10] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pp. 298–312, 2016.