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.

企業におけるソフトウェア開発に対する自動プログラム修正技術

内藤 圭吾† 谷門 照斗† 柗本 真佑† 肥後 芳樹† 楠本 真二†

切貫 弘之†† 倉林 利行†† 丹野 治門††

† 大阪大学大学院情報科学研究科 吹田市
†† 日本電信電話株式会社 港区

あらまし 近年多くの自動プログラム修正の研究が行われている．その修正対象はほとんどの場合オープンソースソフトウェアや自動プログラム修正技術の評価用に用意されたデータセットであり，企業におけるソフトウェア開発においてそのツールおよびアルゴリズムが有効であるかどうかは示されていない．そこで本稿では，実際に企業内で開発，利用されているシステムに対して既存の自動プログラム修正ツールを適用することで，企業のソフトウェア開発で発生したバグに対する自動プログラム修正ツールの有効性を調査した．また，本稿ではその調査結果と企業のソフトウェア開発に対する自動プログラム修正ツール適用の障壁についても議論する．

キーワード デバッグ，プログラム自動修正，コード再利用

1. は じ め に

ソフトウェア開発には様々な工程があり，その中でもデバッグ作業に必要な時間はプログラミング工程の 50%以上を占めるといわれている [1]. そこで，開発工程を短縮するためにデバッグ支援の研究が盛んに行われている．デバッグはバグ同定とバグ修正の二つに分けられる．これまでに，デバッグ支援のためにバグ同定の自動化 [2] [3] [4] や，バグの理解支援の研究 [5] が行われてきた．そして近年，バグの特定及び修正を自動で行う自動プログラム修正技術の研究が盛んに行われている．

自動プログラム修正技術の一つに，ソースコードの再利用に基づく手法である GenProg がある [6]．GenProg は，バグを含むプログラムと，失敗テストを含むテストスイートを入力とし，全てのテストケースを通過するプログラムを出力する．その動作はバグ同定された箇所に対して行の挿入や削除，置換を遺伝的アルゴリズムに基づいて繰り返し行うことによってプログラムの修正を行う，というものである．Le Goues らは GenProg を 8 個のオープンソースソフトウェアのバグ，合計 105 個に対して適用し，そのうち 55 個のバグについて修正に成功することでその有効性を示した [6]．その後も，GenProg を評価する研究や，その他の自動プログラム修正技術の研究が行われてきた．その評価にはいずれも GenProg の評価同様オープンソースソフトウェアや，既存のデータセットが用いられている [7] [8]．つまり，現在企業におけるソフトウェア開発に対して自動プログラム修正技術を適用し評価した研究は未だにない．そのため，現在の自動プログラム修正技術が企業におけるソフトウェア開発において有効であるのか，また有効でない場合どういった問題があり，解決すべきなのかが判明していない．

以上より，本研究では企業で開発運用されているシステムに対して自動プログラム修正技術を適用し，その有効性及び実用化に向けての障壁を調査した．調査の結果，一つのバグについて開発者の修正と同等の修正を自動プログラム修正によって行うことができた．また実用化に向けて三つの障壁を明らかにした．以降 2. では関連研究のアルゴリズムやその有効性について述べる．3. では本研究の研究背景および調査項目について述べる．4. では調査対象と，調査方法について述べる．5. では，実験，調査の結果および考察について述べる．6. では妥当性の脅威について述べ，最後に 7. で本研究のまとめと今後の課題について述べる．

2. 関 連 研 究

近年多くの自動プログラム修正の研究が行われており，それらは以下のように分類できる．

- 再利用に基づく手法
- プログラム意味論に基づく手法
- 修正パターンに基づく手法

これらの手法はいずれも，バグを含むプログラムと，失敗テストを含むテストスイートを入力とし，修正が完了したプログラムを出力する．バグ同定及び，終了判定にはテストスイートを用いる．以下で，これらの手法の代表的なものが，どういった方法で修正を行うのかを紹介する．

2.1 GenProg

まず，再利用に基づく手法の一つである GenProg の修正方法を説明する．GenProg は修正対象プログラム中のソースコードを用いて，バグ同定された箇所を変更したプログラム (以下,

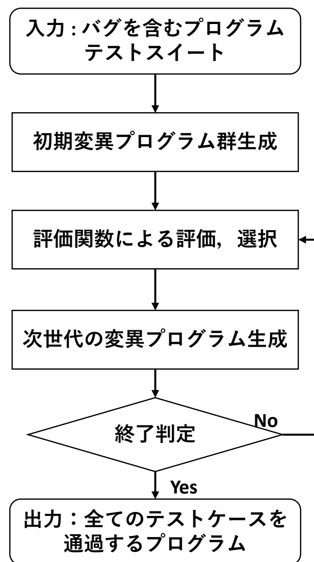

図 1　GenProg の修正の流れ

変異プログラム) を繰り返し生成，評価を行うことでバグの修正を行う．GenProg の修正の流れを図 1 に示す．プログラムの変更の方法は以下の 3 通りがある．

挿入 バグ同定行の次の行に新たにソースコードを挿入する．

削除 バグ同定行を削除する．

置換 バグ同定行を削除し，その行に新たにソースコードを挿入する．

GenProg では，より効率的に修正を行うためにプログラムの変更に遺伝的アルゴリズムを用いている．遺伝的アルゴリズムによる修正の流れを図 2 に示す．以下，遺伝的アルゴリズムによる修正について述べる．

まず，バグを含むプログラムから複数の変異プログラムを生成する．その後，それぞれの変異プログラムに対して全てのテストケースを実行し，成功したテストケースの数をその変異プログラムの評価として与える．そして，評価の良いものだけから次の世代を生み出す．次の世代の生成は以下の 3 通りの方法がある．

コピー 何も変更を加えずに次の世代に同じ変異プログラムを残す．

変異 変更を追加して新たな変異プログラムを生成する．

交叉 二つの変異プログラムの変更内容を組み合わせて新たな変異プログラムを生成する．

世代の生成を，全てのテストケースを通過する変異プログラムが生成されるか，世代数が上限に達するまで繰り返す．

Le Goues らは GenProg を 8 個のオープンソースソフトウェアのバグ，合計 105 個に対して適用し，そのうち 55 個のバグについて修正を成功させることによってその有効性を示した [6]．しかし，GenProg は変異プログラムの評価時に全てのテストケースを実行するため，計算コストがかかる．そこで，

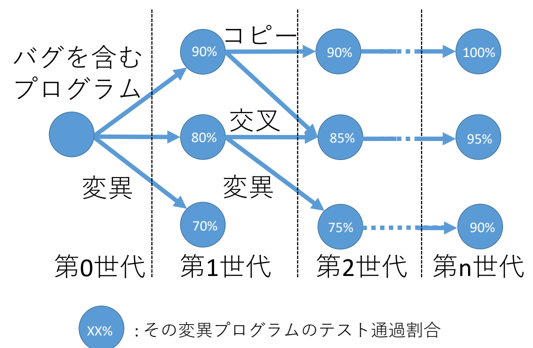

図 2　遺伝的アルゴリズムによるの修正の流れ

Qi らは失敗するテストケースが見つかった場合，その段階でテストケースの実行を打ち切り，次の変異プログラムを生成することで修正を行う手法，RSRepair を提案した [9]．Qi らは GenProg と同様の 8 個のオープンソースソフトウェアのバグを用いて RSRepair を評価し，GenProg と比較してより多くのバグを修正できたと報告している．しかし，一つでも失敗するテストケースが見つかった段階でテストケースの実行を打ち切るため，GenProg とは異なり RSRepair は単一行の修正しか行うことができない．

2.2　その他の手法

次にプログラム意味論に基づく手法である SemFix は，テストスイートからバグ同定された箇所が満たすべき制約を導出し，その制約を満たすプログラム文を生成する [7]. Nguyen らはこの手法を 5 つのソフトウェアのバグに対して適用し， GenProg よりも多くのバグを修正できたことを報告している．再利用に基づく手法は既存のプログラム中に存在しないソースコードによる修正を行うことができないことに対し，プログラム中にソースコードが存在しなくても修正が可能であることがプログラム意味論に基づく手法の利点である．この手法では，バグ同定された箇所が満たすべき論理式を，SMT ソルバを用いて解く．しかし，SMT 問題は NP-完全であるため，論理式が複雑な場合は現実的な時間で解くことができないことが欠点の一つである．また，複数行の変更を必要とするバグの修正を行うこともできないという欠点も存在する．

最後に修正パターンに基づく手法である PAR の修正方法を説明する．PAR ではあらかじめ 10 個の修正パターンを作成しておき，それを用いて修正を行う．Kim らは PAR を 5 つのオープンソースソフトウェアのバグに対して適用し，GenProg よりも多くのバグを可読性の高い形で修正に成功したことを報告している．しかし，用意されたパターンに当てはまらないバグは修正できないことや，修正パターンの作成に技術的に難しいものが存在することが問題点として挙げられている [10]．また，実験対象や実験内容に対する批判も存在している [11]．

本研究では，自動プログラム修正ツールを企業で開発運用されているシステムのバグに対して適用する. その際に複数行の修正が必要不可欠であるため，再利用に基づく手法を対象として調査を行う．

2.3 自動バグ同定手法

今回実験に使用したツールは自動バグ同定に Ochiai というフレームワークを用いているため，以下で Ochiai の自動バグ同定の方法を説明する [12]．Ochiai はバグを含むプログラムと，失敗するテストケースを含むテストスイートを入力として与え，出力には各行のバグが存在している怪しさを出力する．失敗テストケースと成功テストケースの実行経路を用いて怪しさを 0〜1 の数値で計測する．i 行を実行する失敗テストケースの数を $fail(i)$，i 行を実行する成功テストケースの数を $pass(i)$，失敗するテストケースの総数を $totalfail$ とおくと，i 行の怪しさ $suspicious(i)$ は以下の様になる．

$$suspicious(i) = \frac{fail(i)}{\sqrt{totalfail * (fail(i) + pass(i))}}$$

Ochiai はこのメトリクスで，各行の怪しさを計測し，全ての行の怪しさを出力する．

3. 調査の目的

本章では自動プログラム修正技術の課題を示し，課題解決に向けた調査項目について述べる．

3.1 自動プログラム修正技術の課題

関連研究で述べた通り，再利用に基づく手法の一つである GenProg を含む様々な自動プログラム修正技術はオープンソースソフトウェアを用いることによって，手法の有効性を評価している．つまり，先行研究では本来の目的である企業のシステムに対する修正の可否や，その有効性が示されていない．オープンソースソフトウェアと企業で開発されたシステムでは，テストの方法や開発フローなどの点で違いがあるため，オープンソースソフトウェアで有効であるからといって，必ずしも企業で利用可能であるとはいえないと著者らは考えた．

3.2 調査項目

本研究の目的は，企業におけるソフトウェア開発に対する自動プログラム修正技術が有効であるかどうか，および自動プログラム修正技術の実用化に向けた障壁を明らかにすることである．そのため以下の調査項目を設定する．

RQ1 自動プログラム修正は実開発環境でも有効であるか

RQ2 実用化にはどのような障壁が存在するか

4. 調　　査

本章では調査の方法について述べる．

4.1 調査対象

本調査は企業から提供されたシステムのバグ，合計 327 個を対象に行う．システムの詳細は以下の通りである．

開発言語 Java

総行数 約 19 万行

自動プログラム修正のツールとして，自動バグ同定のフレームワーク Ochiai および，GenProg を Java に使用できるように改変した jGenProg の二つを実装したツール Astor を用いた [8]．

実験，評価を行うために次のフィルタリングを行った．括弧の中は，そのフィルタリングを行った後，残ったバグの数である．

- 提供されたバグ (327 個)
- 単体テストで発見されたバグ (上記の内 132 個)
- バグ修正情報を取得できるバグ (上記の内 79 個)
- 修正が Java ファイルのみのバグ (上記の内 23 個)

以下では，それぞれのフィルタリングの必要性とその方法について述べる．

まず，“単体テストで発見されたバグ” について述べる．ソフトウェアの開発中にバグを発見する方法としては，単体テストや結合テストなどが挙げられる．今回対象言語として選んだ Java において，単体テストは JUnit というテストの自動化を行うためのフレームワークが実装されているが，結合テストは自動化がされていない．GenProg は前述のアルゴリズムの通り，変異プログラムを生成するたびに全てのテストケースを実行し，テストケースが成功しているか判定を行うため，テストは実行と成功か失敗かの判定が自動化されている必要がある．そこで，今回は対象バグとして単体テストで発見されているバグを選択した．今後，結合テストの自動化が実装された場合，結合テストで発見されたバグについても対象として実験を行うことができると考えられる．対象システムにおいてバグはバグ票で管理されており，そこにどのようにしてそのバグが発見されたかも記載してあったため，その記述を参考にフィルタリングを行った．

次に，“バグ修正情報が取得できるバグ” について述べる．GenProg の出力は全てのテストケースを通過する修正プログラムである．ここで気をつけなければならないことは，全てのテストケースを通過することと，バグが修正されたことは等価ではないということである．テストスイートが不十分であった場合，バグが修正されていないがテストケースは全て通過する，というプログラムが出力されることもある．つまり，自動プログラム修正ツールが出力した修正プログラムが全てのテストケースを通過するだけの修正プログラムなのか，バグが正しく修正された修正プログラムなのかを評価する必要があるということである．そこで，企業で利用されているバージョン管理システムから，そのバグの修正内容が取得できるかどうかでフィルタリングを行った．バージョン管理システムには以下の情報が管理されている．

- 修正日時
- 修正した開発者名
- 修正したファイル名
- 修正内容

バグ票記載のバグには，バージョン管理システムで修正したリビジョンが確認できないもの，複数のバグが一つのリビジョンで修正され修正内容とバグが紐付けられないものが存在していたため，手作業でそのバグの修正内容が取得できるかどうか確認することでフィルタリングを行った．

最後に “修正が Java ファイルのみのバグ” について述べる．今回利用したツールには修正対象が Java ファイルのみという

```
for(int i; i < num; i++){
      if(checkNum(num)) {
            list = new ArrayList<String>();
+           break; //開発者修正
+           return list; // GenProg 修正
      }
      list.add(num);
}
return list;
```

図 3 修正結果

制約がある．しかし，今回実験対象としたシステムには，ソースコードである Java ファイル以外にも，設定ファイルである .property ファイルや，入出力ファイルである .xml ファイルなどが含まれていた．そこで，修正内容が Java ファイルのみであるかどうかのフィルタリングが必要である．前述した“バグ修正情報が取得できるバグ”のフィルタリングと同様に，バージョン管理システムから，その修正ファイルが Java ファイルのみかどうか手作業でフィルタリングを行った．

4.2 調査方法

本研究では各 RQ に回答するために以下の実験と調査を行った．

実験 企業で開発されたシステムのバグに対して GenProg を適用し，その修正可能性を調査する．

調査 企業とのミーティングを通して実用化に向けての障壁を調査する．

今回，題材としたシステムでは，バグの修正内容についてはバージョン管理ツールで保存されているが，バグを発現させるテストケースの存在しないバグも存在した．そこで，本研究ではバグを含むリビジョンのテストスイートにバグを発現させるテストケースを新たに追加することで，テストスイートの補完を行い，実験を実施した．

5. 結果

この章では実験，調査の結果と結果に対する考察について述べる．

5.1 実験結果

現在，23 件のバグのうち，4 件について実験を行っている．

実験を通して一つのバグについて修正を行うことができた．修正内容を図 3 に示す．ただし，企業内で運用されているシステムであるため，図 3 のコードは実際のコードとは変数名や，バグとは関係のない処理など一部異なる．図 3 を見ると，GenProg の修正は開発者の修正と同等の動作を行うことが分かる．また，その修正コードの可読性も十分なものといえる．よって，企業のソフトウェア開発においても既存の自動プログラム修正技術で修正可能なバグが存在し，有効であることが分かった．

(a)成功テストケース	(b)失敗テストケース
11: int b = 5;	11: int b = 5;
12: if (a > b){	12: if (a > b){
13: a = b;	13: a = b;
14: }else{	14: }else{
15: a = b - a;	15: a = b - a;
16: }	16: }

図 4 テストによるプログラムの実行経路．網掛け部分が実行されたコード．

残りの 3 件は，バグ同定の失敗，自動プログラム修正の失敗により，修正を行うことができなかった．バグ同定の失敗の原因については後に議論する．

5.2 調査結果

実験及び企業とのミーディングを通して，次の 3 点における障壁が確認された．

- 修正可能なバグの数
- パラメータの調整
- テストケースの数

以下では，それぞれの障壁に関する詳細を述べていく．

修正可能なバグの数

4.1 で述べた通り，今回用いたツールには適用するうえでの制約がいくつかある．そこでフィルタリングを行うことで対象バグの数が 23 個まで減少した．これは，全体のバグの数に対してわずか 7% である．バグ修正情報を取得できるかどうかで，53 個のバグがフィルタリングされたが，これは実験，調査を行う上で Astor が出力した修正内容を評価するためのフィルタリングであるため，その 53 個が修正可能であるかは未確認である．しかし，その 53 個全てが修正可能なバグであった場合でも，修正可能なバグの数は 23% にとどまる．この修正可能なバグの数の少なさが，実用化に向けての障壁の一つである．

パラメータの調整

今回用いた自動プログラム修正のツール Astor には，様々なパラメータが存在する．以下のパラメータはその一部である．

- ランダムに行を取得するシード値
- 遺伝的アルゴリズムにおける，一世代あたりの変異プログラムの数．
- 遺伝的アルゴリズムにおける，最大世代数．

こういったパラメータが合計 70 個存在し，最適な値が容易に求められないものも多く存在する．本来自動プログラム修正技術の目的は開発工程の短縮であるにも関わらず，パラメータの調整に時間がかかってしまっては本末転倒である．したがって，このパラメータの調整の難度が二つ目の障壁である．

テストケースの数

企業においてテストケース作成の基準の一つにカバレッジがある．カバレッジとはプログラム全体のうち，テストケースによって実行された命令や分岐の割合のことである．テストケースによって実行された命令の割合を命令カバレッジ，テスト

ケースによって実行された分岐の組み合わせの割合を分岐カバレッジという．カバレッジを指標としてテストケースを作成した場合，一定数のテストケースが作成されるが，自動プログラム修正を行うためにはテストケースの数が不十分な場合がある．2. で自動バグ同定のフレームワークである Ochiai のメトリクスを紹介したが，自動バグ同定のメトリクスでは失敗か成功かに関わらずテストケースの数が怪しさを決定する重要なファクターになっている．以下で，テストケースの数が怪しさに大きく寄与する例を述べる．図 4 に二つのテストケースによる実行経路を示す．テストケースがこの二つだけの場合と，(a) の成功テストケースと同様の実行経路の成功テストケースが 9 個，99 個の場合についての各行怪しさを表 1 にまとめる．図 4 の二つのテストケースだけで，命令カバレッジと分岐カバレッジはともに 100% になっている．しかし，表 1 を見ると，テストケース二つだけでは，失敗テストケースのみで実行されている 13 行目に加えて，11 行目と 12 行目の怪しさも高い値を示している．11 行目，12 行目は成功テストケース，失敗テストケースともに実行されているため，直感的にはそこまで怪しくはないが，テストケースが少ないため，11，12 行目も高い怪しさになってしまっている．表 1 を見ると，成功テストケースを 9 個，99 個に増やすと，11，12 行目の怪しさが減っていくことが分かる．以上の例から分かるように，テストケースの数が怪しさを決定する大きなファクターとなっている．そのため，テストケースの数が不十分であった場合，自動バグ同定が正しく行われない．今回実験を行ったバグについても，自動バグ同定がうまくいっていないために自動プログラム修正が行えなかったバグも存在する．

この障壁は少ないテストケースで高いカバレッジを達成する，という戦略をとっている場合のみに限った話ではない．例として，0〜9 の自然数を入力として，入力の値を四捨五入するプログラムを作成したとする．そのプログラムに対して，代表的なテスト技法である同値分割や，境界値分析に基づきテストケースを作成したとする．その場合，正常系のテストケースとしては，0〜4 の範囲に含まれる自然数を入力とするテストケースと，5〜9 の範囲に含まれる自然数を入力とするテストケースを高々二つずつ用意すると十分である．しかしこれは，あくまで人間が正しく機能を実装できているか，バグが存在していないかを確かめるためのテストケースであるので，自動プログラム修正を行うためには不十分である可能性がある．つまり，人間がプログラムの正しさを確かめるためのテストスイートと，自動プログラム修正において，正しくバグ同定を行うためのテストスイートは異なる，ということがこの障壁の本質である．

以上より，各 RQ に対する回答は次の通りである．

表 1　テストケースの数の違いによる怪しさの値の変化

ソース \成功テストケース数	1	9	99
11: int b = 5;	0.7	0.3	0.1
12: if(a > b){	0.7	0.3	0.1
13: a = b;	1	1	1
15: a = b - a;	0	0	0

RQ1 への回答　企業で開発されたシステムのバグにおいても，自動プログラム修正技術によって修正可能なバグは存在する．

RQ2 への回答　自動プログラム修正技術についての障壁として，修正可能なバグの数の少なさ，パラメータ調整の難度という 2 点の障壁が明らかになった．また，開発者側の自動プログラム修正技術導入に向けた障壁として，人がソフトウェアの品質を確保するためのテストケースと自動プログラム修正技術のためのテストケースに乖離があることが明らかになった．

6. 妥当性の脅威

本研究ではバグを発現させるテストケースを追加して，実験を実施した．その新たなテストケースの作成は，バグの内容やバグの修正内容を把握したうえで，恣意的にバグを発現させるテストケースの作成を行ったため，実際に開発者が手元でバグを発見した際のテストケースとは乖離している可能性がある．開発者が手元でバグを発見した際のテストケースを用いて実験を行った場合，今回とは異なる実験結果や，障壁が明らかになる可能性がある．

7. おわりに

本研究では，既存の再利用に基づく自動プログラム修正技術が，企業におけるソフトウェア開発に適用した際に有効であるかどうか，実用化に向けてどのような障壁が存在しているかを調査した．調査の結果，企業で開発運用されているシステムのバグのうち一つのバグについて開発者による修正と同等の修正を行うことに成功した．また実用化に向けて三つの障壁を明らかにした．今後は，引き続き GenProg の評価を行うとともに，その他の自動プログラム修正ツールの評価も行っていく．また，今回明らかになった障壁の解消に向けたツールの実装などを行うことが今後の課題である．

文　献

[1] J. Baker, “Experts battle l192bn loss to computer bugs,” 2012. http://www.cambridgenews.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html.

[2] D. Saha, M.G. Nanda, P. Dhoolia, V.K. Nandivada, V. Sinha, and S. Chandra, “Fault localization for data-centric programs,” Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp.157–167, 2011.

[3] J.A. Jones and M.J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp.273–282, 2005.

[4] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: Problem determination in large, dynamic internet services,” Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on IEEE, pp.595–604, 2002.

[5] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, “Mimic: locating and understanding bugs by analyzing mimicked executions,” Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp.815–826, 2014.

[6] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” Software Engineering

(ICSE), 2012 34th International Conference on IEEE, pp.3–13, 2012.

[7] H.D.T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” Proceedings of the 2013 International Conference on Software Engineering, pp.772–781, 2013.

[8] M. Martinez and M. Monperrus, “Astor: a program repair library for java,” Proceedings of the 25th International Symposium on Software Testing and Analysis, pp.441–444, 2016.

[9] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” Proceedings of the 36th International Conference on Software Engineering, pp.254–265, 2014.

[10] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” Proceedings of the 2013 International Conference on Software Engineering, pp.802–811, 2013.

[11] M. Monperrus, “A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair,” Proceedings of the 36th International Conference on Software Engineering, pp.234–242, 2014.

[12] R. Abreu, P. Zoeteweij, and A.J. Van Gemund, “An evaluation of similarity coefficients for software fault localization,” Dependable Computing, 2006. PRDC’06. 12th Pacific Rim International Symposium on IEEE, pp.39–46, 2006.