企業のソフトウェア開発に対する自動プログラム修正技術適用の試み

内藤 圭吾[1,a)]　谷門 照斗[1]　柗本 真佑[1]　肥後 芳樹[1]　楠本 真二[1]　切貫 弘之[2]　倉林 利行[2]
丹野 治門[2]

概要：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，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 2 つの工程に分けられる．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，バグの同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 1 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．その修正方法は，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，自動プログラム修正技術を評価するために作成された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，既存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，1 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以下の 4 つの障壁を明らかにした．

- 修正可能なバグの数
- パラメータの調整
- 複数行の修正
- テストケースの数

以降 2 章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 章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 章では調査対象と，調査方法について述べる．5 章では，実験，調査の結果および考察について述べる．6 章では実験，調査の結果に基づいた今後の課題につ

[1] 大阪大学大学院情報科学研究科 吹田市
[2] 日本電信電話株式会社 港区
[a)] k-naitou@ist.osaka-u.ac.jp

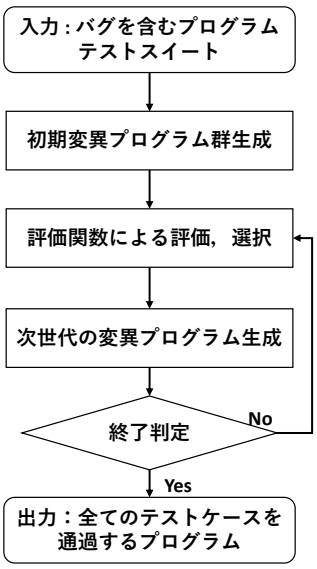

図 1　GenProg の修正の流れ

いて述べる．7 章では妥当性の脅威について述べ，最後に 8 章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 再利用に基づく手法

まず，再利用に基づく手法の 1 つである GenProg の修正方法を説明する．GenProg は修正対象のプログラム中に存在している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，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．プログラムの変更方法は以下の 3 通りがある．

挿入 バグ同定行の次の行に修正対象のプログラム中に存在している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 バグ同定行を修正対象のプログラム中に存在しているソースコードと置換する．

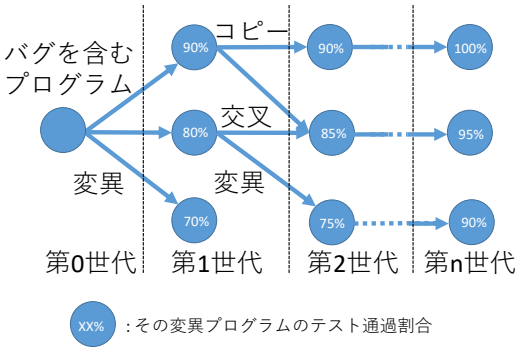

図 2　遺伝的アルゴリズムによるの修正の流れ

GenProg では，プログラムを繰り返し変更する際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を用いた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 2 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，多くの計算コストを要する．

そこで， 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，1 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 プログラム意味論に基づく手法

プログラム意味論に基づく手法である Nopol 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 [10]. Nopol と GenProg の修正手法以外の大きな違いとして，GenProg は修正するバグに特別な制約はないが，Nopol は if 文に関係するバグの修正しかできない点が挙げられる．Xuan らは Nopol を 22 個のバグに適用し，そのうち 17 個のバグの修正に成功させることでその有用性を示した．また，その 17 個の正のうち，13 個の修正は開発者が対象のバグに施した修正と機能的に等価であり，正しい修正であったと報告している．

2.3 修正パターンに基づく手法

次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 はあらかじめ 10 個の修正パターンを作成しておき，それをバグ同定された箇所に適用することで修正を行う．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 [11]．また，実験対象や実験内容に対する批判も存在している [12].

2.4 その他の手法

修正に Stack Overflow を利用する手法に QACrashFix がある [13]．QACrashFix では，エラーが発生した際に出力されるエラートレースを解析し，Stack Overflow で同様のエラーを検索する．その後，その回答からコードスニペットを生成し，その修正コードスニペットをを含むプログラムに適用することで変異プログラムを生成する．QACrashFix は 24 個のバグの内，8 個のバグについて正しい修正パッチを生成することで有効性を示した．また，QACrashFix が修正に成功したバグは既存の自動プログラム修正ツールでは修正できないものであった．

パッチのランク付けに過去の修正履歴を用いる手法として HistoricalFix がある [14]．HistoricalFix の修正方法は，GenProg と大きくは違わないが，変異プログラムの評価順序が GenProg とは異なる．HistoricalFix ではまず対象プログラムの過去のバグ修正履歴の収集を行う．その後 GenProg や PAR で使用された変更操作を用いて変異プログラムを生成し，生成した変異プログラムの変更内容が対象プログラムの過去の修正履歴と類似しているものから順に変異プログラムの評価を行っていく．HistoricalFix は 90 個のバグを対象に修正を行い，23 個のバグについて正しい変異プログラムを出力することでその有効性を示した．

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，GenProg と Nopol は実装したツールが公開されているため，その 2 つのツールに関しては実験を行ったが，PAR はツールが公開されていないため，調査のみを行った．また，QACrashFix と HistoricalFix は，ツールやその使用方法が公開されておらず，加えてその修正方法により修正可能性を論じることもできないため，紹介にとどめる．PAR と同じく修正パターンに基づく手法で，より新しいツールとして，HistoricalFix というツールが公開されている．しかし，本研究環境では正常に実験が行えず，また HistoricalFix は修正パターンが不明であるため，今回は対象から外した．

2.5 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 [15]．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疑惑値を出力する．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疑惑値を 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 行の疑惑値 $suspicious(i)$ 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，各行の疑惑値を計測し，全ての行の疑惑値を出力する．

Ochiai のようなテストを実行し，その実行経路に基づいて各行の疑惑値を計測する自動バグ同定手法を spectrum-based といい，Ochiai の他に Tarantula[16] や，jaccard[4] といった自動バグ同定手法も存在する．しかし，複数の論文で他の spectrum-based の自動バグ同定手法と比較して，Ochiai の自動バグ同定の正確性とその一貫性は優れていると指摘されている [17][18]．今回用いた 自動プログラム修正ツールにおいても，自動バグ同定ステップでは Ochiai が利用されている．

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 1 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して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 既存の自動プログラム修正は企業のソフトウェア開発に対しても有効であるか

RQ2 自動プログラム修正の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 2 つのシステムのバグ，合計 408 個 (327 個 + 81 個) を対象に行う．対象のシステムは共に Java で記述され，その規模はそれぞれ，約 19 万行，約 2.5 万行である．

自動プログラム修正ツールとして，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 2 つを実装したツール Astor[8] と，Nopol を用いた．

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 + 81 個)
- 単体テストで発見されたバグ
  (上記の内 132 個 + 38 個)
- バグ修正情報を取得できるバグ
  (上記の内 78 個 + 2 個)
- 修正が Java ファイルのみのバグ
  (上記の内 22 個 + 2 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，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 というテスト実行，評価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 および Nopol 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，テストの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．自動プログラム修正ツールの出力は全てのテストケースを通過する変異プログラムであり，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変異プログラムが，全てのテストケースを通過するがバグは直っていない変異プログラムなのか，バグが正しく修正された変異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，修正したリビジョンが確認できないもの，複数のバグが 1 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，機能の実装とバグの修正が同時に行われているものなどが存在していたため，そのバグの修正内容が取得でき，バグと修正内容が一意に紐づけられるどうか手作業で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

```
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

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，調査および議論を行った．

実験 企業で開発されたシステムのバグに対して自動プログラム修正ツールを適用し，その修正可能性を実験により明らかにする．

調査 公開されていないツールについては，そのアルゴリズムの観点から各バグの修正可能性を調査する．

議論 企業とのミーティングを通して自動プログラム修正ツール実用化に向けての障壁について議論する．

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．追加テストケースは筆者らが追加したものと企業に追加を依頼したものがある．筆者らが追加したテストケースに関しては，企業に確認を取り追加したテストケースに誤りがないことを確かめた．

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，24 個のバグのうち，13 個のバグについて実験を行っている．残りの 11 個バグは，バグを発現させるテストケースが作成できていない等の問題があるため，実験を行えていない．

今回の実験を通して 1 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 のコードは変数名や，バグとは関係のない処理などが実際のコードとは異なる．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，オープンソースソフトウェアと同様に，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，その修正は有効であることが分かった．残りの 12 個は，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．実験結果を表 1 に示す．それぞれ状態の意味は以下のとおりである．

Succeeded: ツールによってバグの修正に成功した．

Failed (NoModPoint): プログラムの各行に疑惑値を計測したが，どの行を改変することで修正可能か不明であった．このエラーはツールや環境の問題であり，さらなる調査が必要である．

Failed (Exception): ツール実行時に ExecutionException[*1]で実行が停止してしまった．このエラーはツールや環境の問題であり，さらなる調査が必要である．

Failed (BugLocalization): 自動バグ同定によって同定された行に，人間の修正した行が含まれず，なおかつ修正に失敗した．

Failed (NoSolution): 探索空間をすべて調査したが，テストがすべて通る修正が発見されなかった．

また，実験結果について企業のレビューは以下のようであった．

FL

- FL 結果をユーザーにわかりやすく通知する機構を作成すれば，FL だけでも現場で活用できる．
- 動作の近い成功テストと失敗テスト間でも，テスト

表 1 実験結果

Bug ID	ASTOR	NOPOL
BUG-1	Succeeded	Failed(NoSolution)
BUG-2	Failed(NoModPoint)	Failed(Exception)
BUG-3	Failed(BugLocalization)	Failed(Exception)
BUG-4	Failed(BugLocalization)	Failed(BugLocalization)
BUG-5	Failed(BugLocalization)	Failed(BugLocalization)
BUG-6	Failed(BugLocalization)	Failed(BugLocalization)
BUG-7	Failed(BugLocalization)	Failed(BugLocalization)
BUG-8	Failed(BugLocalization)	Failed(BugLocalization)
BUG-9	Failed(BugLocalization)	Failed(BugLocalization)
BUG-10	Failed(BugLocalization)	Failed(BugLocalization)
BUG-11	Failed(NoSolution)	Failed(NoSolution)
BUG-12	Failed(NoSolution)	Failed(NoSolution)
BUG-13	Failed(NoSolution)	Failed(NoSolution)

[*1] java.util.concurrent.ExecutionException

```
for(int i; i < num; i++) {
-     if(i >= num) {
+     if(str[i] != null && i >= num) {
        list.add(str[i]);
      }
}
```

図 4　理論上 PAR によって修正できるバグの一例

で使用するデータの作成方法が異なる場合，バグ原因箇所以外にも失敗テストのみで実行される行が出現してしまう．その場合，関係のない行の疑惑値が高い値を示してしまう．この問題を防ぐために，テスト作成を専任のメンバーが行う，テスト作成にもコーディング規約を設けるなどの対策が必要となる．

APR

- ランダムに修正を行うため，「仕様書に適合していない」，「余計な条件分岐や変数の追加が行われている」等の問題があり，現時点では修正結果を開発現場で利用することは困難である．
- 再利用に基づくため，多くのコードクローンが生成され，保守性の面で課題が存在する．
- 修正方針のヒントを与える，という形ではデバッグ支援に活用できる．

5.2　調査結果

5.2.1　PAR の修正可能性

Kim らが示していた 10 個のパターン [11] を用いて今回のシステムを修正した場合，理論上 2 個のバグを修正できることが明らかになった．ただし，システムごとにバグの修正方法に傾向はみられたため，修正対象のソフトウェアに合うよう適切にパターンを生成することで，この結果は向上すると思われる．理論上修正可能であったバグの一例を図 4 に示す．図 3 と同様の理由により，バグの修正内容に関係のない変数名等は実際のコードとは異なる．また，もう一つのバグも図 4 と同様に条件式の修正であった．

5.3　自動プログラム修正実用化に向けた障壁

実験及び企業とのミーディングを通して，次の 4 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
- 複数行の修正
- 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 節で述べた通り，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，対象バグの数が 408 個から 24 個にまで減少した．これは，全体のバグの数に対してわずか約 6% 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 自動プログラム修正ツールが出力した変異プログラムを評価するためのフィルタリングであるため，その 90 個のバグが修正可能であるかは未確認である．しかし，90 個のバグ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提供されたバグの 28% 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 1 つである．

ただし，バグ修正情報が取得できるバグであるかどうかのフィルタリングで多くのバグがフィルターされてしまったのは，あくまで今回実験の結果を評価するためのフィルタリングのため，実用化に対しての問題ではない．実用化に向けての問題としては，その前後のフィルタリング，“単体テストで発見されたバグ”，“修正が Java ファイルのみのバグ”の 2 つにあると考えられる．前者は，結合テストが自動化されていないという，自動プログラム修正ツール以外にも問題が存在するが，後者はそうではない．システムはソースだけで構成されている場合は極稀で，ほとんどの場合は設定ファイルやプロパティファイルも含まれている．そうであるにも関わらず，自動プログラム修正の修正対象がソースコードのみであるという点は，自動プログラム修正ツールに閉じた問題である．実際，今回の実験でも，“修正が Java ファイルのみのバグ”のフィルタリングでバグの数が 80 個から 24 個，実に 70% ものバグがフィルターされてしまっている．

以上より，今後の 自動プログラム修正の課題は，その修正対象をソースコードのみから，それ以外の設定ファイルやプロパティファイルにまで拡張することであると考えられる．

Github 上のプロジェクトの内スター数上位 100 件のプロジェクトと，今回企業から提供されたシステムについて，それぞれ全バグ修正コミットに対する修正が Java のみのバグ修正コミットの割合を調査したところ，OSS で約 60%，企業のシステムで約 50% と，約 10%の違いがみられた．したがって，この障壁については企業のシステムにおいて顕著に現れていると考えられる．

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，1 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 2 つ目の障壁である．

複数行の修正

今回の実験において，修正できたバグはわずかに 1 件に留まってしまった．この原因はいくつかあるが，その中でも大きな原因として，複数行の修正を必要とするバグの修正が難しいことが挙げられる．今回実験対象としたシステムのバグにおいて，修正行が 1 行だけであったものは，わずかに 2 件であり，その他のバグは修正に複数行の変更を必要としていた．従って，自動プログラム修正ツールを実用化するためには，複数行の修正が可能であることが必要不可欠である．しかし，今回実験に用いた自動プログラム修正ツール Nopol は単一行の修正しか行うことはできない．そのため Nopol ではその 2 件以外のバグはアルゴリズム上開発者と同様の修正は不可能であった．また，GenProg では遺伝的アルゴリズムを用いることで複数行の修正を可能としているが，GenProg にも複数行修正を行う上での問題がある．GenProg で遺伝的アルゴリズムを用いる場合，その評価関数として，変異プログラムのテストケースの通過率を用いている．しかし，Java のテストケースを作る際に，1 行 1 行に対応したテストケースを作ることはない．それ故に，複数行の修正が必要なバグにおいて，バグの存在するすべての行を修正して初めて 1 つのバグ発現テストケースが通過するようになる．そのため，複数行の修正の修正が必要なバグの修正過程において，遺伝的アルゴリズムの評価関数がうまく働かない．従って，各行の修正により評価が上昇していく新たな評価関数を用意する，各行に対応するテストケースを作成する，複数行を一度に修正するアルゴリズムを導入する等の対策が必要である．

以上の理由より，既存の自動プログラム修正ツールは複数行の修正を必要とするバグの修正が難しい．これが 3 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 1 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 章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疑惑値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疑惑値に大きく寄与する例を述べる．図 5 に 2 つのテストケースによる実行経路を示す．

```
11:  int b = 5;
12:  if (a > b){
13:       a = b;
14:  }else{
15:       a = b - a;
16:  }
```

(a)成功テストケース

```
11:  int b = 5;
12:  if (a > b){
13:       a = b;
14:  }else{
15:       a = b - a;
16:  }
```

(b)失敗テストケース

図 5 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．

テストケースがこの 2 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．各行の疑惑値を表 2 にまとめる．図 5 の 2 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 2 を見ると，テストケース 2 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 行目の疑惑値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 行目も高い疑惑値になってしまっている．表 2 を見ると，成功テストケースを 9 個，99 個に増やすと，11，12 行目の疑惑値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疑惑値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，

表 2 テストケースの数の違いによる疑惑値の値の変化

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

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 2 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．

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し，一部のバグについては有用である．

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，複数行の修正の困難さという 4 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. 今後の課題

今回実験，調査を行う上で自動バグ同定及び自動プログラム修正の対象がソースコードのみである点が修正可能なバグの数を少なくしていた原因の 1 つであった．そこで今後，対象をソースコードだけから設定ファイル等にまで拡張した自動バグ同定および自動プログラム修正ツールの実装が課題となる．

7. 妥当性の脅威

7.1 内的妥当性

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．

7.2 外的妥当性

今回は特定の企業の 2 つのプロジェクトに対して既存ツールを適用したため，対象が他の企業のプロジェクトになった場合や，同一企業のプロジェクトにおいても他のプロジェクトを対象とした場合結果が異なる可能性がある．今回の結果はあくまで対象としたプロジェクトに特有のものである可能性があるため，今後対象プロジェクトを増やし結果の一般化を図る必要がある．

8. 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 1 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 4 つの障壁を明らかにした．今後は，引き続き 自動プログラム修正ツールの評価を行うとともに，対象をソースコードのみから設定ファイル等まで拡張した自動バグ同定および自動プログラム修正のツール開発を行っていく．

参考文献

[1] Baker, J.: Experts battle L192bn loss to computer bugs, , available from ⟨http://www.cambridgenews.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html⟩ (accessed 2015/06/09).

[2] Saha, D., Nanda, M. G., Dhoolia, P., Nandivada, V. K., Sinha, V. and Chandra, S.: 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] Jones, J. A. and Harrold, M. J.: 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] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E.: 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] Zuddas, D., Jin, W., Pastore, F., Mariani, L. and Orso, A.: 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] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W.: 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] Nguyen, H. D. T., Qi, D., Roychoudhury, A. and Chandra, S.: Semfix: Program repair via semantic analysis, *Proceedings of the 2013 International Conference on Software Engineering*, pp. 772–781 (2013).

[8] Martinez, M. and Monperrus, M.: ASTOR: a program repair library for Java, *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441–444 (2016).

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

[10] Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D. and Monperrus, M.: Nopol: Automatic repair of conditional statement bugs in java programs, *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55 (2017).

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

[12] Monperrus, M.: 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).

[13] Gao, Q., Zhang, H., Wang, J., Xiong, Y., Zhang, L. and Mei, H.: Fixing recurring crash bugs via analyzing q&a sites (T), *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, IEEE, pp. 307–318 (2015).

[14] Le, X. B. D., Lo, D. and Goues, C. L.: History Driven Program Repair, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1, pp. 213–224 (online), DOI: 10.1109/SANER.2016.76 (2016).

[15] Abreu, R., Zoeteweij, P. and Van Gemund, A. J.: 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).

[16] Jones, J. A., Harrold, M. J. and Stasko, J.: Visualization of test information to assist fault localization, *Proceedings of the 24th international conference on Software engineering*, ACM, pp. 467–477 (2002).

[17] Abreu, R., Zoeteweij, P. and Van Gemund, A. J.: 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).

[18] Abreu, R., Zoeteweij, P. and Van Gemund, A. J.: On the accuracy of spectrum-based fault localization, *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, IEEE, pp. 89–98 (2007).