修士学位論文

題目

バグの特徴を用いた自動プログラム修正ツールの比較

指導教員
楠本　真二　教授

報告者
横山　晴樹

平成 29 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 28 年度 修士学位論文

バグの特徴を用いた自動プログラム修正ツールの比較

横山　晴樹

内容梗概

デバッグは多大な労力を要する作業であり，作業コストの削減が課題である．近年，デバッグの支援に向けて自動プログラム修正に関する多数の手法が提案されており，ツールの公開も行われている．自動プログラム修正は欠陥を含むプログラムとテスト集合から，修正したプログラムを出力するデバッグ支援手法であり，近年盛んに研究されている．既存の自動プログラム修正に対する評価方法は，修正できた欠陥の個数や，修正時間，修正品質などの評価が主であった．一方，修正結果は欠陥が持つ特徴によって様々であるにもかかわらず，どのような特徴を持つ欠陥を修正できたのかという観点での評価はあまり行われていない．

本研究では，欠陥の特徴を欠陥レポートから抽出し，優先度と再オープンの有無の観点での評価を行った．優先度の高い欠陥は開発者が早急な修正を必要とするものであり，再オープンされた欠陥は開発者が修正に失敗した欠陥である．そのため，自動プログラム修正がこれらの欠陥を修正できれば，開発者に対して大きな貢献となる．調査では，オープンソースの Java プロジェクトにおける 170 個の欠陥に対する自動プログラム修正ツール jGenProg，jKali，および Nopol の修正性能を調査した．調査では，jGenProg と Nopol が高い優先度を持つ欠陥を高い割合で修正できることを明らかにした．また，修正時間や修正行数などの観点から，自動プログラム修正の修正性能を多角的に評価し，修正行数の増加が修正時間の増加に繋がる可能性を示した．

主な用語

デバッグ，自動プログラム修正，欠陥追跡システム，ソフトウェアリポジトリマイニング

目 次

1 はじめに **1**

2 準備 **3**

2.1 自動プログラム修正 3

2.2 欠陥追跡システム 7

3 研究目的 **10**

3.1 研究動機 10

3.2 リサーチクエスチョン 11

4 調査 **12**

4.1 調査手法 12

4.2 調査対象 13

4.3 調査結果 14

5 考察 **16**

5.1 修正時間および修正行数の分析 16

5.2 Trivial の欠陥に対する目視調査 17

6 データセットを拡張した調査 **19**

6.1 欠陥の収集 19

6.2 拡張後のデータセットにおける調査結果 20

7 関連研究 **22**

8 調査の妥当性に対する議論 **26**

8.1 内的妥当性に対する議論 26

8.2 外的妥当性に対する議論 26

9 おわりに **27**

謝辞 **28**

参考文献 **29**

図 目 次

1 デバッグ自動化技術の概要 4
2 自動プログラム修正の処理概要 5
3 GenProg の処理概要 6
4 JIRA における欠陥レポートの簡略化したライフサイクル 8
5 調査手法の概要 12
6 1 つの欠陥レポートに対して複数の欠陥が対応する例 13
7 優先度別および再オープンの有無別の修正成功率 15
8 優先度別および再オープンの有無別の平均修正時間 16
9 優先度別および再オープンの有無別の平均追加/削除行数 17
10 MATH-393 におけるテストコードの変更部分 18
11 拡張後のデータセットにおける優先度別および再オープンの有無別の修正成功率 . . 21

表 目 次

1 Math プロジェクトに対する修正個数 10
2 優先度別および再オープンの有無別の修正個数 10
3 Defects4J におけるプロジェクト 14
4 Defects4J における優先度別および再オープンの有無別の欠陥数 14
5 優先度別および再オープンの有無別の修正個数 14
6 データセット拡張に用いる欠陥 20
7 データセット拡張後の欠陥数 20
8 優先度別および再オープンの有無別の修正個数 21

1 はじめに

ソフトウェア開発において，デバッグは開発者が多大な労力を要する作業である．デバッグにかかる費用は世界中に年間推定 3000 億ドルにも及び，開発者はコーディング時間の約半分をデバッグに費やすと言われている [1]．そのため，デバッグ費用の削減が課題である．近年，デバッグの支援に向けて自動プログラム修正に関する多数の手法が提案されており，ツールの公開も行われている [2, 3, 4, 5, 6]．

自動プログラム修正は，欠陥を含むプログラムとテスト集合を入力として与え，修正に成功すると，与えた全てのテストを通過するプログラムを出力する手法の総称である．自動プログラム修正の修正性能は，主に欠陥を含むプログラム集合に対する修正個数によって評価される．また，修正時間 [2]，人間にとって読みやすいかを評価したり [7]，差分が小さいパッチになっているかを評価したりすることもある [8]．一方，修正結果は欠陥が持つ特徴によって様々であるにもかかわらず，どのような特徴を持つ欠陥を修正できたのかという観点での評価はあまり行われていない．

そこで，本研究では欠陥の特徴を用いて，自動プログラム修正ツールの評価を行う．欠陥の特徴としては，欠陥追跡システム中の欠陥レポートから得られる以下の 2 つを用いる．

- 優先度
- 再オープンの有無

優先度の高い欠陥は開発者が早急な修正を必要とするものである．再オープンされた欠陥は開発者が修正に失敗した欠陥である．そのため，自動プログラム修正がこれらの欠陥を修正できれば，開発者に対して大きな貢献となる．

我々は，Java プロジェクトのデータセット Defects4J [9] に含まれる 170 個の欠陥と，各欠陥に対する自動プログラム修正ツールの修正記録や，各欠陥における優先度と再オープンの有無を調査し，欠陥の特徴と自動プログラム修正の修正性能との関係を分析した．本研究では以下の 3 つの自動プログラム修正ツールを用いる．

- jGenProg [5]
- jKali [5]
- Nopol [6]

本研究における主要な貢献は次のとおりである．

- jGenProg と Nopol が優先度の高い欠陥を多く修正することを明らかにした．
- Nopol が再オープンされた欠陥をあまり修正できないことを明らかにした．
- 修正時間や修正行数といった観点から，自動プログラム修正の修正性能を多角的に評価した．

本論文では，2 章で準備として自動プログラム修正と欠陥追跡システムを紹介する．3 章では研究目的として，研究動機とリサーチクエスチョンを述べる．4 章では調査手法と調査対象を示し，調査結果を述べ，リサーチクエスチョンに回答する．5 章では調査結果に関する考察を述べる．6 章ではデータセットを拡張して追加で行った調査について述べる．7 章では本論文に関連する研究を紹介する．8 章では調査における内的，外的妥当性に対する議論について述べる．最後に，9 章では本研究のまとめと今後の研究課題について述べる．

2 準備

2.1 自動プログラム修正

近年，デバッグ支援の研究として，デバッグの自動化に向けた研究が活発に行われている．デバッグの主要な作業には以下の 2 つが挙げられる．

- 欠陥の所在の特定
- ソースコードの修正

以上の作業を自動化する技術として以下の 2 つが挙げられる．

- 自動欠陥限局
- 自動プログラム修正

以上のデバッグ自動化技術の概要を図 1 に示す．自動欠陥限局は欠陥を含むプログラムとテスト集合から，欠陥の所在を特定する技術である．また，自動プログラム修正は，欠陥を含むプログラムとテスト集合，欠陥の所在から，修正したプログラムを生成する．欠陥を含むプログラムとは，通過しないテストケースが少なくとも 1 つ存在するプログラムである．入力に用いるテスト集合には，欠陥を含むプログラムが通過しないテストケースと，通過するテストケースがそれぞれ少なくとも 1 つ含まれていなければならない．自動プログラム修正に関する研究は近年特に活発であり [2, 3, 4]，国際会議へ投稿された論文の数は 2012 年以降のもので 50 本を超える [10]．

次に，自動プログラム修正の処理概要を図 2 に示す．まず，自動プログラム修正は欠陥を含むプログラムにおける欠陥の所在に対し，ソースコードの変更を行う．変更方法には，同一プロジェクト内からのプログラム文の挿入や，プログラム文の削除を行う手法 [2, 5, 11] や，欠陥の所在においてプログラムの分岐を行い，分岐条件をテスト集合から生成する手法 [4, 6] などがある．次に，テスト集合を実行し，全てのテストを通過する場合は修正したプログラムを出力する．通過しないテストが存在する場合は，再びソースコードの変更を行う．

本調査で用いる自動プログラム修正ツールは以下の 3 つである．

- jGenProg [5]
- jKali [5]
- Nopol [6]

また，3 つのツールのうち jGenProg と jKali は GenProg [2] および Kali [11] を基にした，Java プログラムの欠陥修正を目的としたツールである．Nopol も同様に Java プログラムを対象としたツールである．以下では，GenProg, Kali および Nopol の概要を述べる．

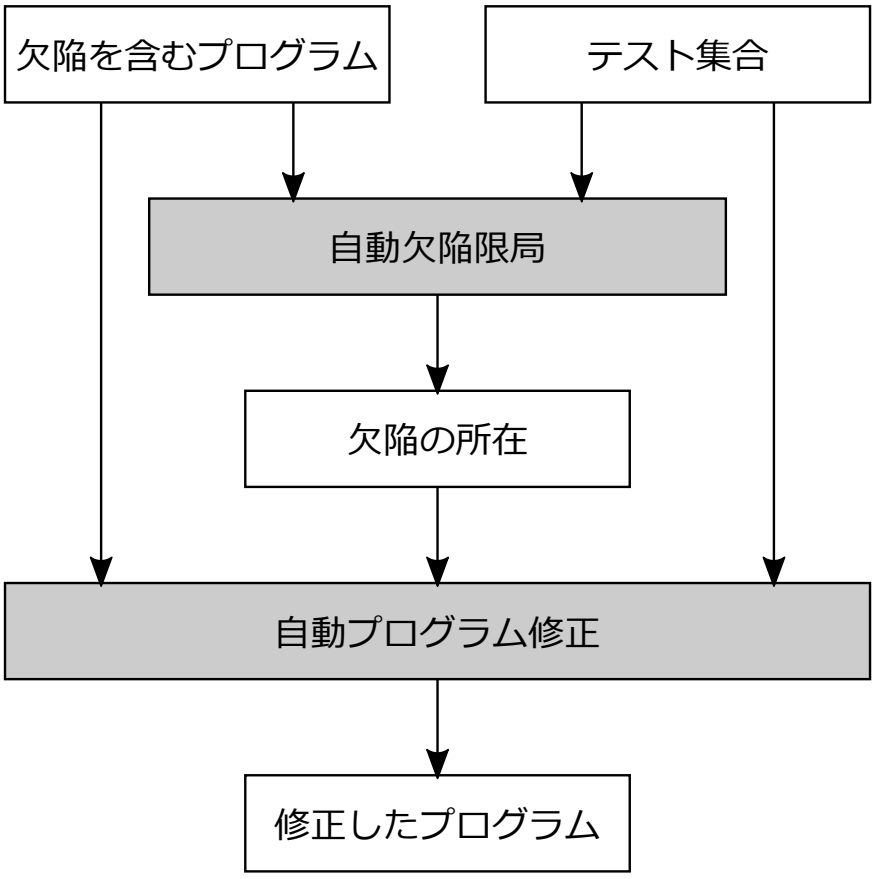

図 1: デバッグ自動化技術の概要

GenProg

GenProg は遺伝的プログラミング [12] に基づく自動プログラム修正である．遺伝的プログラミングとは，プログラムの構造などの木構造で表されるデータを対象にした，解の探索を行うアルゴリズムである．遺伝的プログラミングを用いた自動プログラム修正では，欠陥を含むプログラムを基にして，ランダム性のある操作によって様々なプログラムを生成し，テスト集合を全て通過するプログラムを生成することを目指す．

GenProg の修正手法は図 3 のようになる．初期変異プログラムの生成では，後述の変異によってプログラムに変更を加えたものを一定数生成する．次に，トーナメント選択では，生成されたプログラムから修正完了に近いプログラムを一定数選択し，残りのプログラムを廃棄する．交叉では，トーナメント選択で絞り込まれたプログラム集合から，2 つずつプログラムを選び，プログラム中のランダムな位置で 2 つのプログラムを入れ替える．また，変異では，欠陥の所在となる位置にプログラム文を挿入したり，削除したり，置換したりする．置換は，挿入と削除の組み合わせで表される．最後

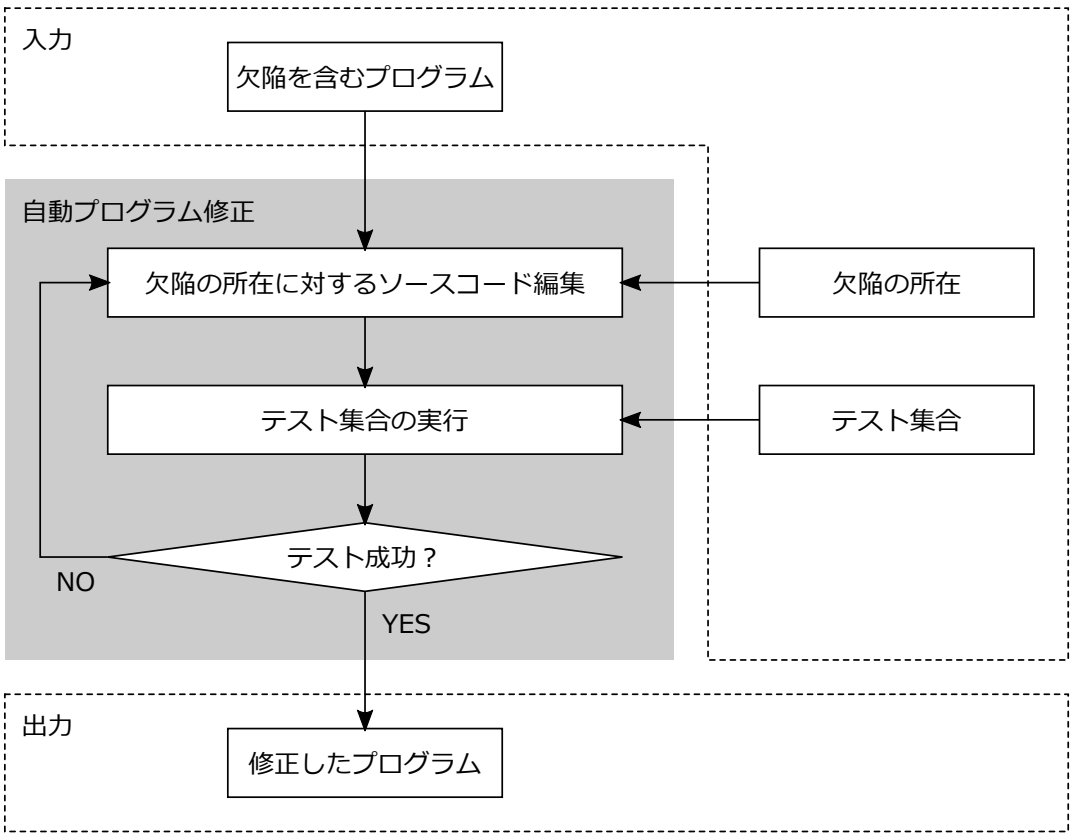

図 2: 自動プログラム修正の処理概要

に，生成したプログラムに対しテスト集合を実行し，全てのテストを通過するプログラムを出力する．全てのテストを通過するプログラムが存在しない場合は，トーナメント選択に戻る．

GenProg は C 言語のプログラムを対象としたツールが作成され，105 個の欠陥のうち 55 個を修正することに成功した．また，最近では Java のプログラムを対象としたツールも作成されている [5]．

GenProg はソースコードが公開されており，様々な亜種が存在する．例えば，遺伝的プログラミングの変異部分のみに機能を制限して高速化した RSRepair [13] や，意味的に等しいテストの実行を抑制し，テスト実行回数を削減した AE [14] などがある．

Kali

Qi らの調査 [11] は，GenProg, RSRepair, AE といった手法がプログラムの機能を一部削除するような修正を行う場合があることを明らかにした．テスト集合の構成によっては，プログラムに必要な機能を削除することでテスト集合を通過できてしまう場合があるためである．しかし，機能削除による修正は，多くの場合開発者が意図しない修正となっている [11] ため，機能削除によって修正に成

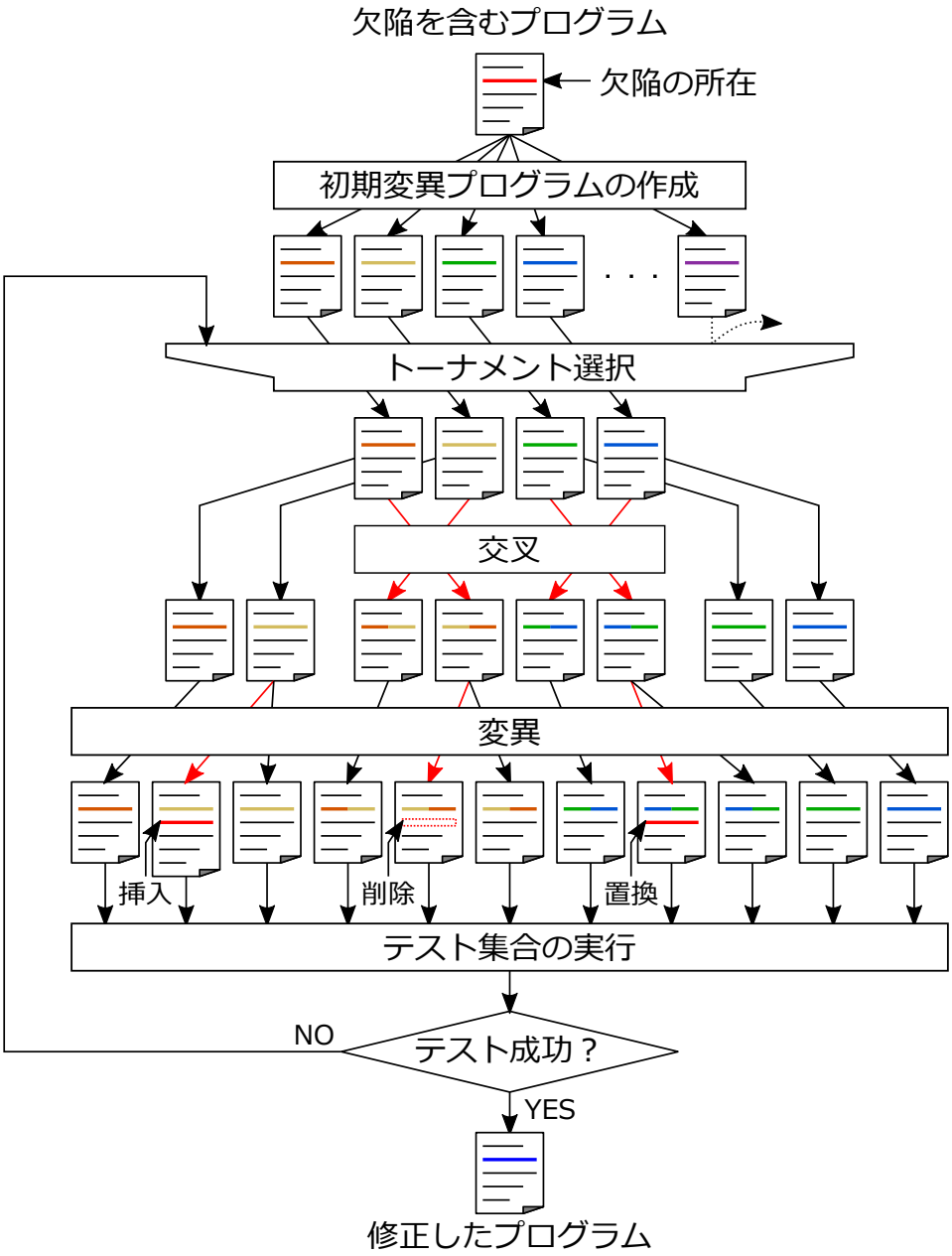

図 3: GenProg の処理概要

功しても望ましい結果とは言えない．

Kali は上記の調査を通じて作成されたツールであり，機能削除による修正を行うツールである．Kali が修正に成功する場合，機能削除によって全てのテスト集合を通過できてしまうことになる．Kali は以下の 3 つの方法で機能削除を行う．

- プログラム文およびブロックの削除
- if 文の条件式を恒真および恒偽に変更
- 関数中での return 文の挿入

また，Kali は GenProg と同様，Java のプログラムを対象としたツールが公開されている [5].

Nopol

Nopol は if 文の追加および if 文の条件式の変更により修正を行うツールである．
Nopol の修正手法は以下の 2 段階から構成される．

- 満たすべき制約の収集
- プログラム合成

満たすべき制約の収集では，プログラムの制御が欠陥の所在に達した時に，スコープ内の変数が満たさなければならない制約をテストの実行により収集する．プログラム合成では，収集した変数の制約を満たすように if 文を追加したり if 文の条件式を変更したりする．制約を満たすような式を生成する際，Nopol は制約を SMT 問題に変換し，SMT ソルバによって問題を解く．SMT 問題は NP-完全な決定問題であり，多項式時間での解法が存在せず，網羅的な探索が必要である [15]．SMT ソルバとは，SMT 問題を高速に解くためのツールであり，Nopol では Z3 [16] や，CVC4 [17] が用いられている．

SMT 問題に帰着させる自動プログラム修正は，テスト集合全体から制約を抽出し SMT 問題に帰着させる SemFix [18] が最も素朴な手法である．また，SemFix の亜種として，差分が小さく開発者にとって読みやすい修正を行うことを目指した DirectFix [8] がある．これらの手法は，ソースコードの規模の増大に伴い，制約の数が増大し，SMT 問題を解くために必要な時間が爆発的に増大するという弱点がある．一方 Nopol では，通過しないテストのみに着目することで制約の数を抑え，高速化に成功している．また，制約を抽出する箇所を絞り込むことでソースコードの規模が増大しても制約の数が増大しないようにした Angelix [4] といった手法もある．

2.2 欠陥追跡システム

欠陥追跡システムとは，ソフトウェア開発中に発生した欠陥を，開発者によって修正されるまで追跡するためのシステムである．欠陥追跡システムは発生した欠陥の一元管理や，修正作業の割り当て，欠陥修正における議論の記録等を可能にする．欠陥は欠陥レポートという形式で欠陥追跡システムに登録される．欠陥レポートには以下のような記録が含まれる．

- 欠陥の名称
- 欠陥を説明する記述

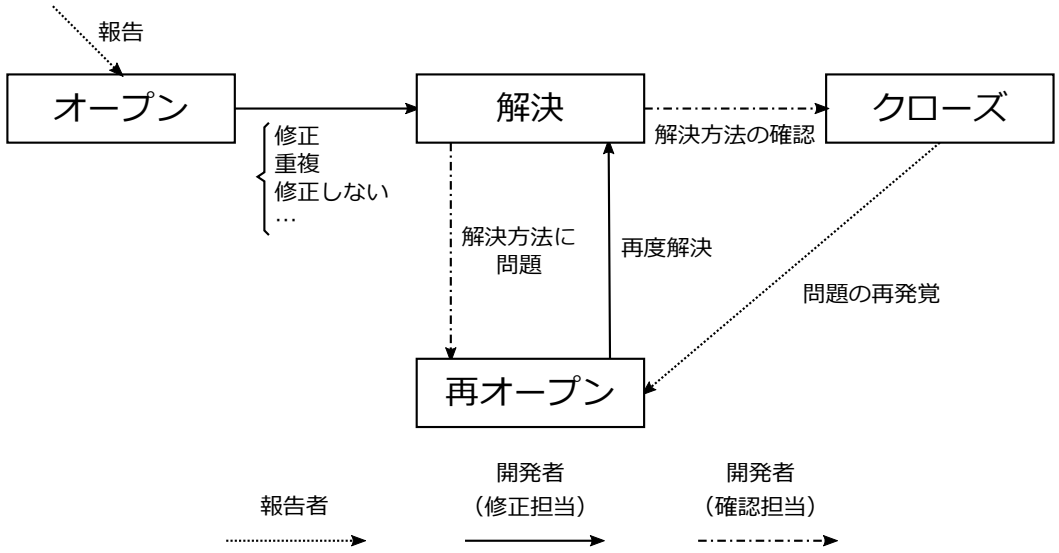

図 4: JIRA における欠陥レポートの簡略化したライフサイクル

- 欠陥の報告者名
- 欠陥修正に割り当てられた開発者名
- 修正を確認した開発者名
- 欠陥のライフサイクル
- 欠陥の優先度

欠陥レポートが高い優先度を持つことは，開発者が早急に欠陥の修正を行う必要があることを表す．また，欠陥レポートはライフサイクルを持ち，欠陥に対するあらゆるイベントが記録される．例えば，イベントには次のものがある．

- 欠陥レポートのステータスの変更
- 欠陥修正への開発者の割り当て
- 欠陥修正に伴うコミットの記録
- 欠陥の優先度の変更

図 4 に，欠陥追跡システムの 1 つである JIRA [19] における，欠陥レポートの簡略化したライフサイクルを示す．各セルは欠陥レポートのステータスを表し，有向辺はステータスの変更を表す．通常，欠陥レポートは欠陥の報告者によって作成され（オープン），修正担当の開発者によって「修正」，

「重複」,「修正しない」といった方法で解決され，確認担当の開発者が解決方法を確認し，問題がなければ閉じられる（クローズ）．しかし，解決方法に問題があったり，閉じられた後に再度問題が発覚したりすると，欠陥レポートは再度開かれる（再オープン）．

欠陥が再オープンされると，開発者の作業量増加や，ソフトウェア利用者の信用の低下に繋がる [20]．また，再オープンされた欠陥レポートは，再オープンのない欠陥レポートに比べ，閉じられるまでの過程が長期化するという報告もされている [21]．

3 研究目的

3.1 研究動機

これまでに，自動プログラム修正における評価には欠陥の修正個数，修正時間，修正品質といった評価指標が用いられてきた．Martinez らの調査 [22] は，欠陥を含んだ Java プロジェクトのデータセット Defects4J [9] に対し，自動プログラム修正ツール jGenProg [5], jKali [5], Nopol [6] を実行させて，各ツールの修正個数や修正時間を示した．

例えば，Defects4J 内の Math [23] プロジェクトに対する修正個数は表 1 のようになる．修正個数の観点で見れば，Nopol は良い修正性能を持っている．

欠陥の中には優先して修正すべき欠陥や，開発者による修正が不完全で，繰り返し修正作業が行われた欠陥などがある．Math プロジェクトは欠陥追跡システム JIRA で管理されており，欠陥毎に優先度の高さや再オープンの有無を調査することができる．優先度の高さは

$$\text{Blocker} > \text{Critical} > \text{Major} > \text{Minor} > \text{Trivial}$$

である．再オープンの有無は，欠陥レポートのステータスが一度でも再オープンとなれば，「再オープンあり」となる．我々は，予備調査として jGenProg, jKali, Nopol が修正できた Math プロジェクトの各欠陥に対し，優先度および再オープンの有無の対応付けを行った．

表 2 は，Math プロジェクトに対する修正個数を優先度および再オープンの有無の観点で分離したものである．優先度に注目すると，jGenProg や jKali は最も高い優先度 Blocker を持つ欠陥の修正に成功している．また，再オープンに注目すると，jGenProg や jKali は再オープンされた欠陥の修正に成功している．このように，優先度や再オープンの有無といった欠陥の特徴を用いることで，既存の

表 1: Math プロジェクトに対する修正個数

自動プログラム修正ツール	jGenProg	jKali	Nopol
修正個数	18	14	21

表 2: 優先度別および再オープンの有無別の修正個数

優先度/再オープンの有無	jGenProg	jKali	Nopol
Blocker	1	1	0
Critical	2	0	2
Major	11	11	15
Minor	4	2	4
Trivial	0	0	0
再オープンあり	1	1	0
再オープンなし	17	13	21

修正個数などの観点とは異なる視点で自動プログラム修正を評価できる可能性がある．しかし，上記の予備調査は少数のデータに対する分析であり，妥当な評価を行うためには，多数のデータを用いた調査が必要である．

本研究の目的は，既存の自動プログラム修正を欠陥の特徴を用いて評価することである．欠陥の特徴を得る手段として，欠陥追跡システムに登録される欠陥レポートを利用する．本研究では特に，以下の 2 つを欠陥の特徴として用いる．

- 優先度
- 再オープンの有無

自動プログラム修正が早急な修正を必要とする優先度の高い欠陥を修正できるならば，開発者への貢献が大きくなる．また，自動プログラム修正が再オープンされた欠陥を修正できるならば，開発者がデバッグに費やす時間を削減することができる．

3.2 リサーチクエスチョン

本研究では，以下の 2 つのリサーチクエスチョンに回答することで，欠陥の特徴を用いた自動プログラム修正の評価を行う．

RQ-1 自動プログラム修正は優先度の高い欠陥を修正できるか．

RQ-2 自動プログラム修正は再オープンされた欠陥を修正できるか．

RQ-1 では，自動プログラム修正が優先度の高い欠陥を優先度の低い欠陥に比べ高い割合で修正できるかを調査する．ここで，優先度の高い欠陥とは，JIRA において優先度 Critical 以上の欠陥である．自動プログラム修正が高い優先度の欠陥を修正できれば開発者への大きな貢献となる．

RQ-2 では，自動プログラム修正が再オープンされた欠陥を再オープンのない欠陥に比べ高い割合で修正できるかを調査する．ここで，再オープンされた欠陥とは，欠陥レポートの状態が一度でも再オープンとなったことのある欠陥である．自動プログラム修正が再オープンされた欠陥を修正できることは，開発者が修正に失敗するような欠陥を修正できることを意味する．

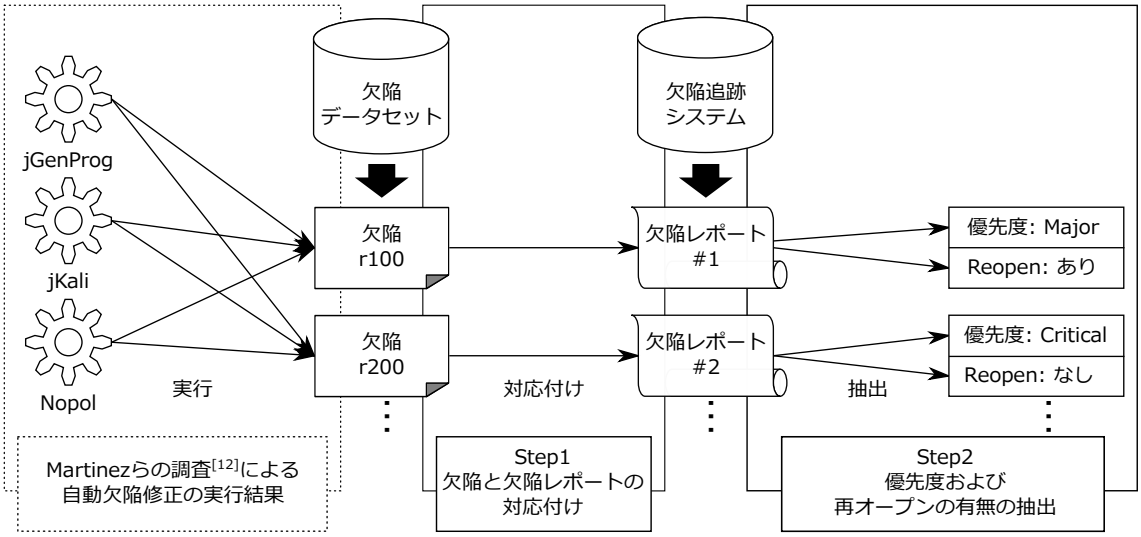

図 5: 調査手法の概要

4 調査

4.1 調査手法

本研究における調査手法の概要を図 5 に示す．調査は以下の 2 ステップで構成される．

ステップ 1 欠陥と欠陥レポートの対応付け

ステップ 2 優先度と再オープンの有無の抽出

ステップ 1 では，自動プログラム修正の修正対象となる各欠陥と，その欠陥を含んだプロジェクトと連携している欠陥追跡システムから，各欠陥と欠陥レポートの ID を 1 対 1 に対応させる．ここで，各欠陥はバージョン管理システムから抽出されており，バージョン管理システムにおけるコミット履歴は欠陥追跡システムの欠陥レポートから参照できると仮定する．各欠陥がバージョン管理システムから抽出されているならば，その欠陥のコミット ID を得ることができる．そして，そのコミット ID を記録した欠陥レポートを検索することで，欠陥と欠陥レポートの対応付けを得ることができる．ただし，図 6 のように，一般に欠陥レポートとコミット履歴は 1 対多の関係にある．それは，欠陥の修正が複数のコミットに渡ったり，再オープンによって修正コミットが複数回になったりするからである．そこで，1 つの欠陥レポートに対して複数の欠陥が対応する場合，本調査では最も古い欠陥を調査に用いる．

図 6 のように欠陥レポートが 1 度再オープンされている場合，最も古い欠陥 r100 に注目すると，この欠陥は r110 のコミットで修正され，閉じられ，r120 の時に再び開かれている．そのため，r100 は実際に再オープンされた欠陥である．一方，r120 の欠陥に注目すると，この欠陥は r130 のコミットで修正され，閉じられ，その後再び開かれていない．そのため，r120 は実際には再オープンされ

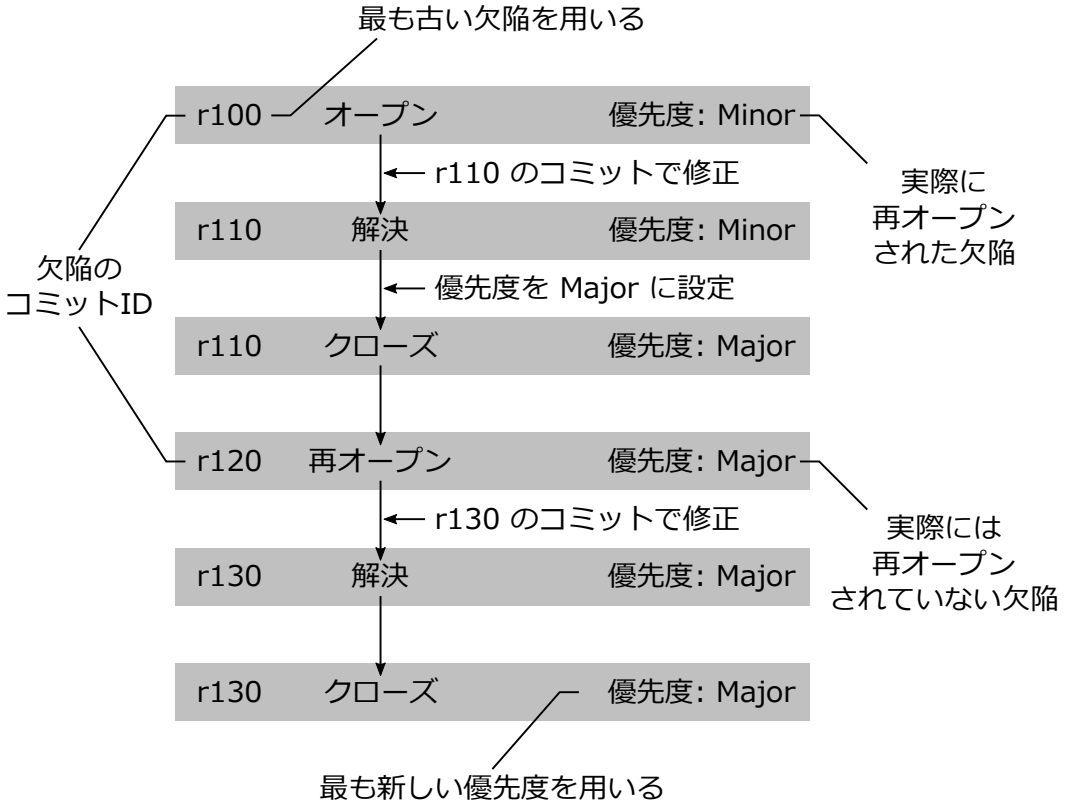

図 6: 1 つの欠陥レポートに対して複数の欠陥が対応する例

ていない欠陥であり，この欠陥を再オープンされた欠陥として用いるのは適切ではない．

ステップ 2 では，各欠陥レポートから，優先度と再オープンの有無を抽出する．優先度は欠陥レポートに記述されている．ただし，図 6 のように開発者は欠陥レポートの優先度を変更することがある．そこで，我々は最新の優先度が開発者の判断する優先度を最も反映すると判断し，本調査では，欠陥レポートに記述されている最も新しい優先度を用いる．また，再オープンの有無も欠陥レポート中のライフサイクル記録から判断することができる．JIRA のように欠陥レポートのステータスとして再オープンが用意されていれば，再オープンの有無の判断は容易である．一方，Github [24] のように，あらかじめ再オープンが用意されていない欠陥追跡システムの場合，本調査では，クローズが複数回存在すれば，「再オープンあり」と判断する．

4.2 調査対象

調査対象は，Java のオープンソースプロジェクトから欠陥を抽出したデータセットの Defects4J である．そのうち，Martinez らの調査 [22] において実行されたプロジェクト（Lang [25], Math [23]）を用いる．そのため，本研究に置ける調査対象は表 3 の通りとなる．ここで，表 3 の優先度および再オープンの有無の列はそれぞれ，優先度および再オープンの有無を調査できるか否かを表している．

また，優先度別および再オープンの有無別の欠陥数は表 4 の通りである．

4.3 調査結果

各ツールに対する修正個数を優先度および再オープンの有無で分類した結果を表 5 に示す．さらに，各ツールと優先度および再オープンの有無との組み合わせにおける修正成功率を図 7 に示す．我々は，特定の優先度に対する修正成功率が全体に対する修正成功率より高いとき，自動プログラム修正がその優先度を多く修正できると判断する．また，再オープンされた欠陥に対する修正成功率が

表 3: Defects4J におけるプロジェクト

プロジェクト	欠陥の数	欠陥追跡システム	優先度	再オープンの有無
Lang	65	Jira	利用可	利用可
Math	106	Jira	利用可	利用可

表 4: Defects4J における優先度別および再オープンの有無別の欠陥数

	Blocker	Critical	Major	Minor	Trivial	計
Lang	2	0	48	13	0	65
Math	1	8	63	32	1	105
計	3	8	101	45	1	170

	再オープンあり	再オープンなし	計
Lang	4	61	65
Math	7	98	105
計	11	159	170

表 5: 優先度別および再オープンの有無別の修正個数

	Blocker	Critical	Major	Minor	Trivial	計
jGenProg	1	2	11	4	0	20
jKali	1	0	11	2	0	16
Nopol	1	2	21	4	0	29
欠陥の総数	3	8	101	44	1	170

	再オープンあり	再オープンなし	計
jGenProg	1	17	18
jKali	1	13	14
Nopol	0	28	28
欠陥の総数	11	159	170

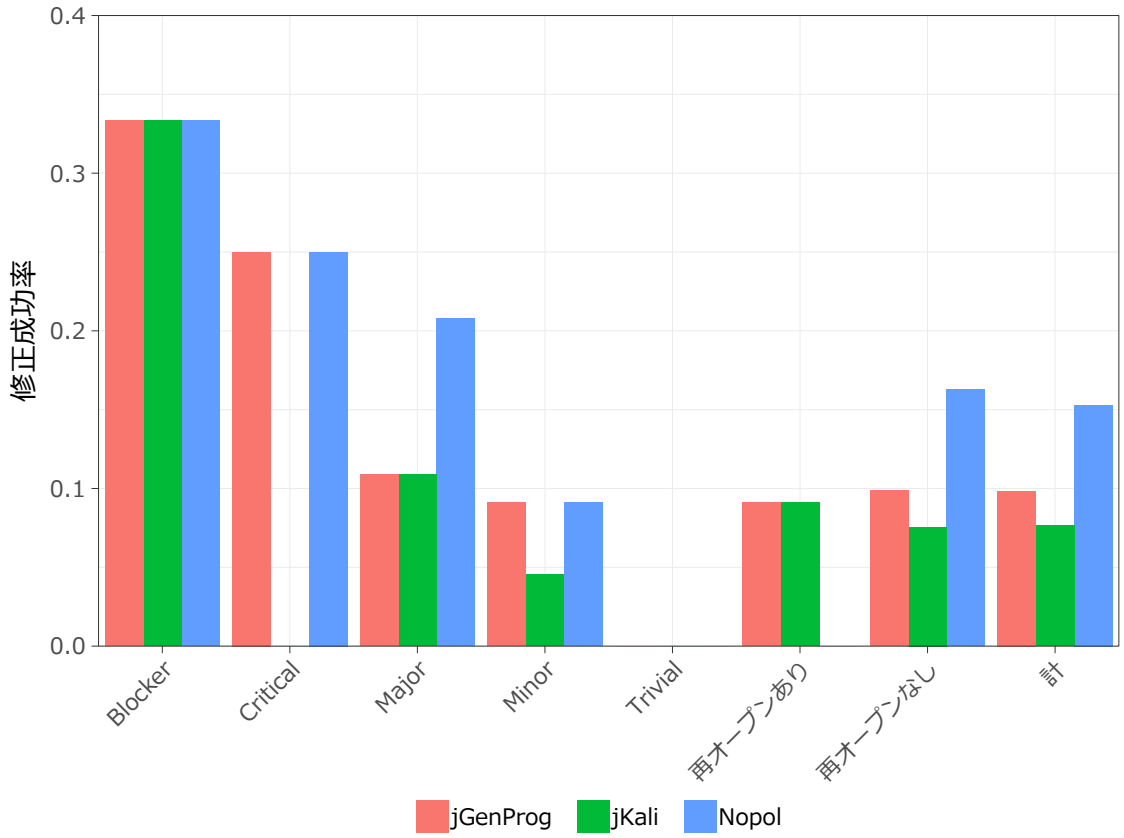

図 7: 優先度別および再オープンの有無別の修正成功率

再オープンされていない欠陥に対する修正成功率より高いとき，自動プログラム修正が再オープンされた欠陥を多く修正できると判断する．

我々が注目する点は，早急な修正を必要とする Blocker や Critical の欠陥に対する修正成功率が，全体に対する修正成功率に比べ高くなっていることである．ただし，jKali は Critical の欠陥を全く修正できていない．そのため，RQ-1 への回答は，jGenProg と Nopol に関しては Yes と言える．一方，jKali に関しては No となる．また，Trivial の欠陥はどのツールでも修正できなかった．

一方，再オープンの有無では修正成功率はあまり変わらなかった．また，Nopol は再オープンされた欠陥を全く修正できなかった．そのため，RQ-2 への回答は，全てのツールに対して No となる．

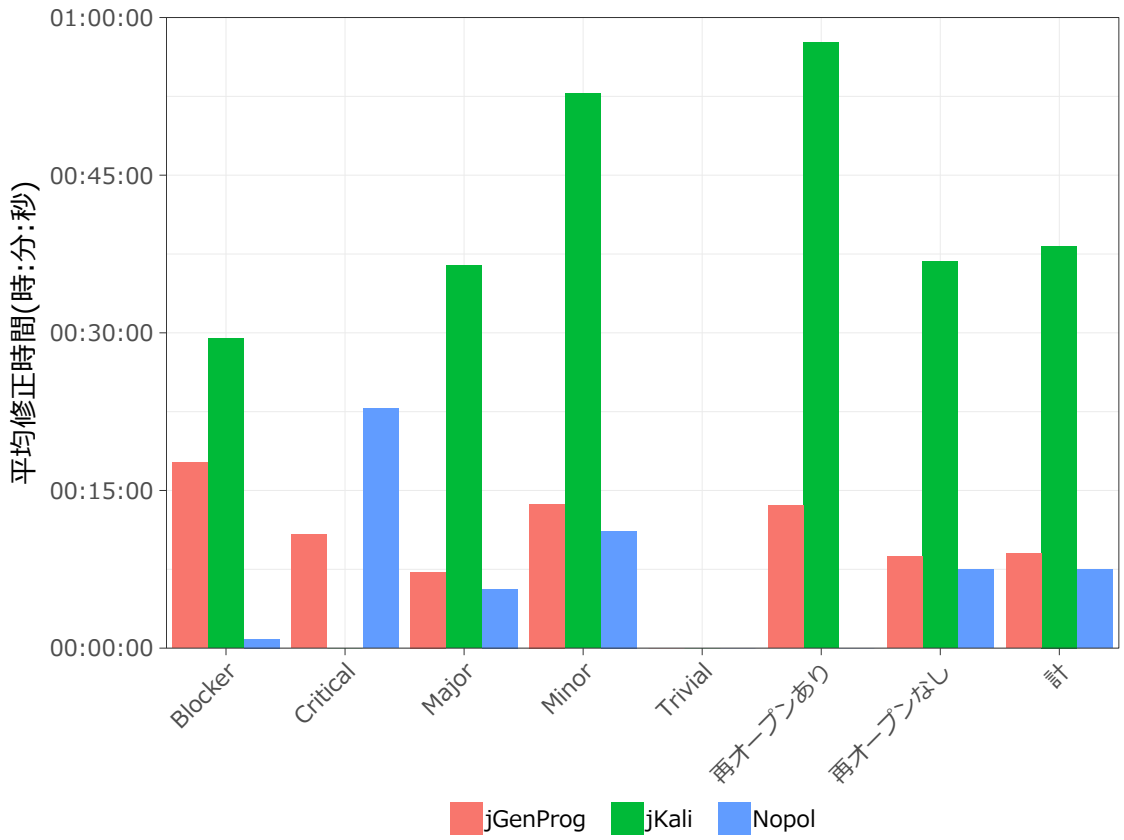

図 8: 優先度別および再オープンの有無別の平均修正時間

5 考察

5.1 修正時間および修正行数の分析

修正成功率の調査結果を分析するため，我々は各ツールが欠陥の修正に成功した際の修正時間を調査した．図 8 は，優先度別および再オープンの有無別の平均修正時間を示している．ここで，平均修正時間とは表 5 のセル毎に算出した修正時間の平均である．例えば，Nopol と再オープンなしの組み合わせでは，29 個の欠陥から平均修正時間を算出している．この結果から，全てのツールにおいて，Minor に対する平均修正時間が全体平均よりも長いことがわかる．再オープンされた欠陥に対しては，jGenProg と jKali の平均修正時間が長くなっている．

また，我々は優先度や再オープンの有無と修正成功率との関係を分析するため，自動プログラム修正の修正行数に注目した．自動プログラム修正において，欠陥の修正に成功した場合，修正部分をソースコードの差分として得ることができる．ソースコードの差分は追加部分と削除部分で構成される．そこで，追加部分の行数と削除部分の行数を別々に集計した．各ツールに対する平均追加/削除行数を図 9 に示す．ここで，平均追加/削除行数とは，表 5 のセル毎に算出した追加行数および削

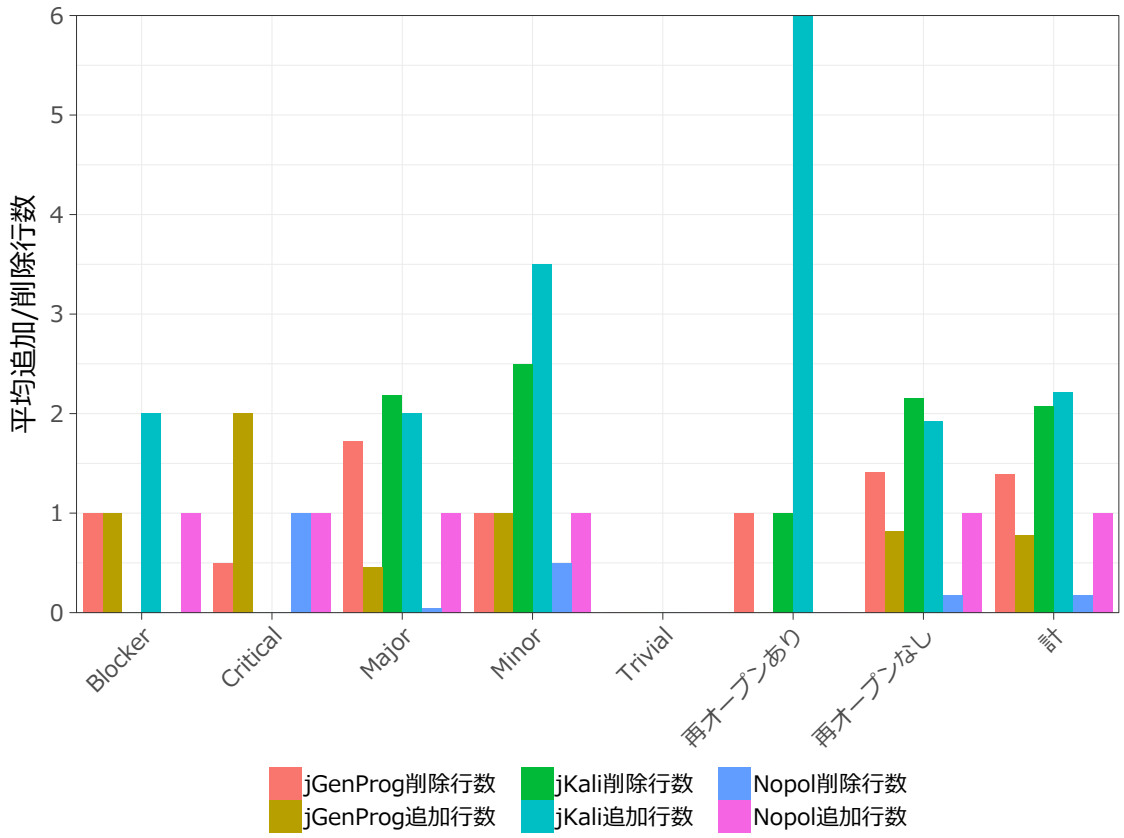

図 9: 優先度別および再オープンの有無別の平均追加/削除行数

除行数の平均である．図 9 から，Minor に対する平均追加/削除行数が全体の平均追加/削除行数よりも多くなっていることがわかる．また，再オープンされた欠陥に対しては，jKali の追加行数が非常に多くなっている．

修正時間の観点では，全てのツールが Minor の欠陥や再オープンされた欠陥の修正に多くの時間を掛けていた．修正行数の観点では，全てのツールの Minor の欠陥に対する修正行数が多くなっており，再オープンされた欠陥に対しては，jKali の追加行数が非常に多くなっていた．この結果から，修正行数の増加が修正時間の増加に繋がる可能性があると言える．

5.2 Trivial の欠陥に対する目視調査

我々は全てのツールが Trivial の欠陥を修正できないことを明らかにした．Trivial の定義は，機能やデータに影響しない欠陥（例えばスペルミス）である [21]．Trivial の定義に従えば，Trivial の欠陥では通らないテストが発生しないはずである．

しかし，本調査で用いた Trivial の欠陥である MATH-393 [26] に含まれるコミットを目視確認すると，ソースコードの変更と共に，テストコードの変更が行われていた．テストコードの変更部分は

```
--- .../MultiStartUnivariateRealOptimizerTest.java (r799874)
+++ .../MultiStartUnivariateRealOptimizerTest.java (r979032)
@@ -83,8 +83,8 @@
   assertEquals(-0.27195612846834, minimizer.optimize(f, GoalType.MINIMIZE, -0.3, -0.2), 1.0e-13);
-  assertEquals(-0.27194301946870, minimizer.getResult(), 1.0e-13);
-  assertEquals(-0.04433426940878, minimizer.getFunctionValue(), 1.0e-13);
+  assertEquals(-0.27195612846834, minimizer.getResult(), 1.0e-13);
+  assertEquals(-0.04433426954946, minimizer.getFunctionValue(), 1.0e-13);
   double[] optima = minimizer.getOptima();
   double[] optimaValues = minimizer.getOptimaValues();
```

図 10: MATH-393 におけるテストコードの変更部分

図 10 の通りである．テストにおける期待値がごく僅かに変更されているが，1.0×10^{-13} の精度が要求されるため，`getResult()` および `FunctionValue()` メソッドの機能面での修正が必要である．そのため，開発者が Trivial と判断したこの欠陥レポートは，実際には Trivial の定義に反した欠陥レポートであることがわかった．

以上のように，優先度の定義と開発者による優先度の判断が食い違う場合が他の欠陥レポートにも存在する可能性があり，今後の研究で優先度の正確さを明らかにする必要がある．

6　データセットを拡張した調査

6.1　欠陥の収集

表 4 の Blocker や Critical，Trivial，再オープンありの列に注目すると，サンプル数が非常に少ないことがわかる．そこで，本調査ではデータセットの拡張を目的として Blocker や Critial，再オープンありの欠陥を収集した．Trivial の欠陥は定義上テストケースを通らないような欠陥ではなく，自動プログラム修正の修正対象外であるため本調査では収集しない．

欠陥の収集対象は優先度および再オープンの有無を利用可能な Lang [25] および Math [23] プロジェクトである．また，プロジェクト開始から 2017 年 1 月 1 日現在までの JIRA で管理されている全ての欠陥レポートを用いる．得られた欠陥レポートは，以下の基準で絞り込みを行う．

- 欠陥修正が行われているか
- ステータスがクローズであるか
- 優先度が Blocker, Critical である，または再オープンされているか
- テストとソースコードの変更が行われているか
- テストでは未定義のシンボルを用いていないか
- Defects4J の欠陥と重複していないか

次に，絞り込み後に残った欠陥レポートに含まれるコミットから，欠陥を抽出する．Lang および Math プロジェクトでは，ソースコードの修正とテストの追加を 1 つのコミットで行うことが奨励されており [27, 28]，コミットそのものを欠陥として用いることができる例は少ない．そのため，テストの追加とソースコードの修正を分離し，テストの追加のみを適用することで，欠陥を抽出することができる．

本調査では，Defects4J [9] データセットの作成方法に準拠し，1 つのデータには 1 つの欠陥のみが含まれることを保証する．すなわち，1 つの欠陥に対し，複数の欠陥レポートにわたって存在する複数の欠陥が存在することはない．以上を実現させるため，テストの追加を行う前に，通過していないテストがあるかを調査し，これから追加するテストによって明らかになる欠陥と既存の欠陥を分離するため，通過していないテストを無効化する．

以上の欠陥データセットの構築方法で得られた欠陥は表 6 のとおりである．データセット拡張に用いることのできる欠陥は，Critical の欠陥が延べ 5 個，再オープンされた欠陥が延べ 7 個となった．一方，Blocker の欠陥を利用することはできなかった．

また，データセット拡張後の欠陥数は表 7 のとおりである．データセットの拡張により，Lang における Critical の欠陥が利用可能となった．

6.2 拡張後のデータセットにおける調査結果

表 6 の 10 個の欠陥に対し，jGenProg, jKali, Nopol をそれぞれ実行した．各ツールに与える設定は，Martinez らのは Defects4J に対する調査 [22] に従っている．ツールを実行した結果，全てのツールが欠陥を 1 つも修正することができなかった．そのため，修正個数の観点では，表 5 と同様の結果となった．次に，以上の結果から修正成功率の算出を行った．優先度別および再オープンの有無別の修正成功率は図 11 のようになる．図 11 から，優先度の観点では図 7 の場合に比べ，Critical の修正成功率が低下していることがわかる．一方，再オープンの有無に関してはあまり修正成功率が変化していない．

本調査では，さらに優先度および再オープンの有無の各カテゴリ間における修正成功率の変化を統計的手法を用いて比較した．用いた統計的手法は Fisher の正確検定 [29, 30] である．検定方法は有意水準 5%の片側検定で，帰無仮説は「カテゴリ間での修正成功率に差はない」とした．

表 6: データセット拡張に用いる欠陥

プロジェクト	欠陥レポート ID	優先度	再オープンの有無
lang	428	Minor	あり
	440	Major	あり
	1003	Critical	あり
	1147	Critical	なし
math	153	Critical	なし
	327	Critical	なし
	484	Major	あり
	1068	Minor	あり
	1070	Critical	あり
	1165	Minor	あり

表 7: データセット拡張後の欠陥数

	Blocker	Critical	Major	Minor	Trivial	計
Lang	2(+0)	2(+2)	49(+1)	16(+1)	0(+0)	69(+4)
Math	1(+0)	11(+3)	64(+1)	34(+2)	1(+0)	111(+6)
計	3(+0)	13(+5)	113(+2)	50(+3)	1(+0)	180(+10)

	再オープンあり	再オープンなし	計
Lang	7(+3)	62(+1)	69(+4)
Math	11(+4)	100(+2)	111(+6)
計	18(+7)	162(+3)	180(+10)

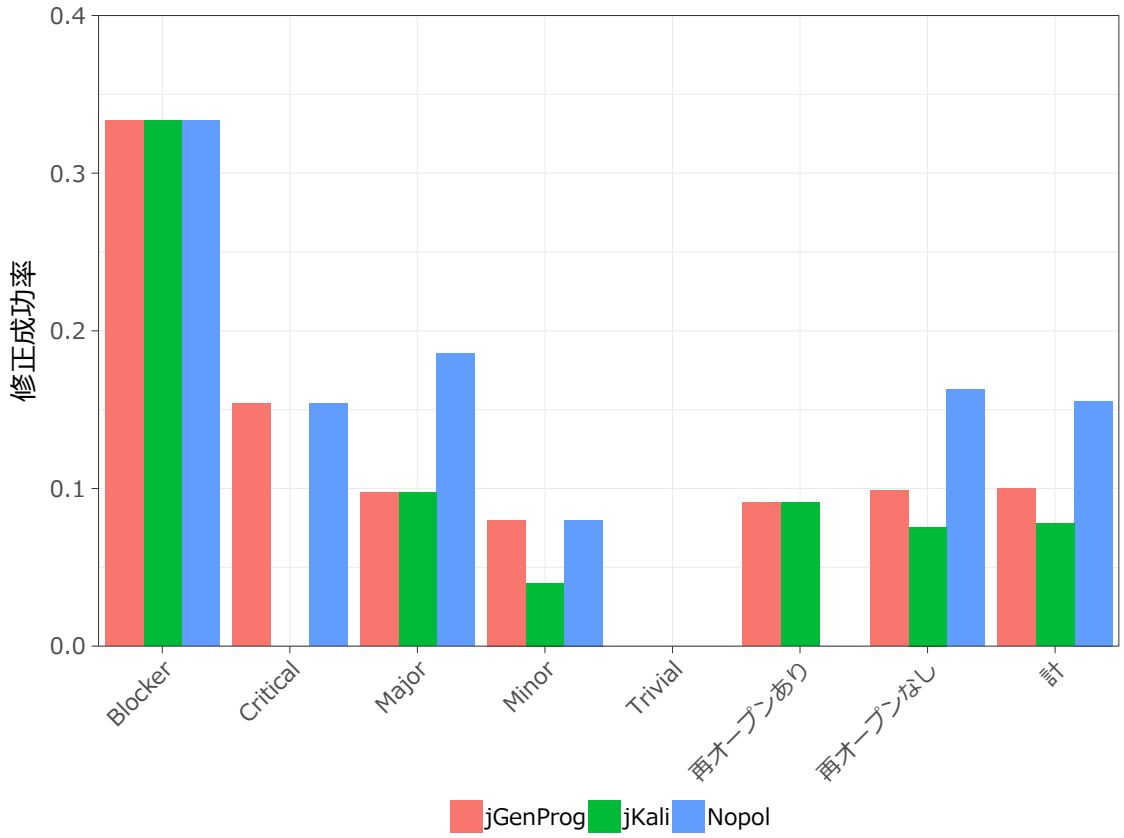

図 11: 拡張後のデータセットにおける優先度別および再オープンの有無別の修正成功率

以上の条件設定における検定結果を表 8 に示す．検定の結果，どの組み合わせでも有意水準 5%の下で有意な差は見られなかった．

以上の調査より，優先度が高いほど修正成功率が高くなる見込みはあるが，現時点での調査では，統計的に有意な差があるとは言えないという結論を得た．特に Blocker の欠陥数は少なく，データセットのさらなる拡張が必要である．

再オープンの有無に関しては，jGenProg, jKali に関しては明瞭な差はなく，Nopol は再オープンされた欠陥を 1 つも修正できなかったが，統計的有意な差が見られなかった．

表 8: 優先度別および再オープンの有無別の修正個数

	Blocker /Critical	Critical /Major	Major /Minor	Minor /Trivial	再オープンあり /再オープンなし
jGenProg	0.4893	0.3989	0.4892	0.9216	0.6906
jKali	0.1875	0.2862	0.1771	0.9608	0.6027
Nopol	0.4893	0.5642	0.0631	0.9216	0.1521

7 関連研究

自動プログラム修正の手法

GenProg および Nopol に関連する手法は 2.1 節で述べたとおりである．本節では 2.1 節に加えた最近の手法について述べる．

SearchRepair [31] は人間によって書かれたコード片のデータベースを構築し，SMT ソルバを用いて最適なコード片を選択し，修正を行う手法である．SearchRepair は C 言語の欠陥を対象としており，778 個の C 言語の欠陥データセットに対し 150 個の欠陥を修正し，GenProg 等の手法では修正できない欠陥もいくつか修正に成功した．

SPR [3] は文の追加，式の変更などのプログラムへの抽象的な変更操作をスキーマとして定義し，スキーマの適用とスキーマの具体化の 2 段階で修正を行う手法である．SPR では，プログラムの修正を修正パッチの探索問題として捉え，修正を段階化することで探索空間を効率よく削減した．その結果，105 個の C 言語の欠陥データセットに対し 44 個を修正した．この修正個数は GenProg の修正個数（18 個）の倍以上となる．また，SPR を発展させたものとして，機械学習によるスキーマ選択の効率化を施した Prophet [32] がある．

PAR [7] は人間が書いた修正パッチを分析して作成した修正パターンをソースコードに当てはめる手法である．PAR は 119 個の C 言語のデータセットに対し 27 個を修正した．この修正個数は GenProg の修正個数（16 個）を上回る．また，PAR では出力した修正パッチが人間にとって読みやすいかを分析しており，89 人の学生と 164 人の開発者に対して，PAR の修正パッチと GenProg の修正パッチ，開発者の修正パッチのうちどの修正パッチが読みやすいかを目視で比較する被験者実験を行っている．一方，その評価方法に関しては批判的エッセイも投稿されている [33]．PAR のアイディアを発展させたものとして，開発履歴から自動的に修正パターンを抽出し，ソースコードに当てはめる HDRepair [34] といった手法も存在する．

DeepFix [35] はディープラーニングをを用いた修正手法であり，文法的誤りを修正することができる．DeepFix は学生への 93 個のプログラミングタスクから収集した 6971 個のエラーを含む C 言語プログラムに対し，1881 個（27%）の修正に成功した．

ACS [36] は，プログラム合成による手法であり，正確なパッチの出力を目的としている．自動プログラム修正が出力するパッチは与えられたテスト集合をすべて通過するが，そのパッチが開発者にとって受け入れがたい修正になっていることがある．例えば，機能を削除することでテスト集合を通過できてしまう場合などがある．ACS は if 文の条件式での変数の使われ方の分析や API ドキュメントを通じた API の使われ方の分析，ブーリアン型の関数や記号の使われ方の分析に基づいた制約の生成方法によってプログラム合成を行っている．ACS は Defects4J の 4 つのプロジェクトに対し 18 個の修正に成功している．

自動プログラム修正に関する調査研究

自動プログラム修正に関する研究はツールの実装だけでなく，パッチの正しさに関する調査 [11] や，異なる言語でのツールの有効性の調査 [22]，既存研究への批判的レビュー等 [33] の様々な調査研究が行われている．

Qi らは，自動プログラム修正が出力したパッチを「もっともらしい」パッチと「正確な」パッチに分類し [11]，既存の自動プログラム修正のパッチを分析した．正確なパッチとは，欠陥に対する開発者の修正パッチと意味的に等しいパッチである．一方，テスト集合をすべて通過するが正確なパッチではないものをもっともらしいパッチと定義している．もっともらしいパッチの中には機能を削除することでテスト集合を通過できてしまうパッチなどがあり，GenProg, RSRepair, および AE が出力するパッチの中には機能削除を行うパッチが含まれていることを指摘した．また，著者らは機能削除のみを行うツール Kali を作成し，機能削除に限るならば既存手法よりも高速に実行できることを示した．

Barr らは，既存のソースコードから開発者のパッチを再現できるかという視点での調査を行った [37]．この調査は，GenProg などの既存のソースコードからプログラム文を移植する自動プログラム修正が原理的に修正できる限界を示すものである．調査の結果，Java のプロジェクトにおける 15,723 コミットのうち，11%が移植によって完全に再現可能であり，43%が部分的に再現可能であることを示した．

Martinez らは，Barr らの調査 [37] に関連して，行単位およびトークン単位の移植によって再現できるコミットの割合を調査した [38]．調査の結果，Java の 6 つのプロジェクトにおける計 7,076 コミットのうち，3–17%が行単位の移植によって完全に再現可能，29–52%がトークン単位の移植によって完全に再現可能であることを示した．

さらに，Martinez らは，Java の 14 プロジェクトにおける計 89,993 コミットの抽象構文木の差分を分析し，173 種類の文の変化を調査した [39]．また，欠陥修正に関するコミットの多くが，複数箇所の修正を要することを明らかにした．

Monperrus らは，修正パターンを用いた自動プログラム修正 PAR [7] に対する批判的レビューを行った [33]．このエッセイの中では，PAR の評価方法，特にパッチが人間にとって読みやすいかという評価の問題点を指摘しており，この指摘をはじめとして，自動プログラム修正をどのように評価すべきか，新たに自動プログラム修正を提案する際に手法をどのように位置づけるべきかといった議論を行っている．

再オープンされた欠陥に関する研究

欠陥追跡システムを用いた研究の一環として，再オープンされた欠陥に関する研究もいくつか行われている．

Shihab らは大規模なオープンソースソフトウェアにおける再オープン記録に対して以下の 4 つの

観点で調査を行った [40].

- 業務中の習慣（例：欠陥が最初にクローズされる曜日）
- 欠陥レポート（例：欠陥が発見されるコンポーネント）
- 欠陥修正（例：欠陥が最初に修正されるまでの期間）
- 開発チーム（例：欠陥修正を行う人物の経験）

また，Shihab らは再オープンされる欠陥の予測も行っており，作成された予測モデルは 62.9%の適合率，84.5%の再現率を達成した [20].

Mi らは Eclipse プロジェクトにおける 4 つのオープンソースソフトウェアに対し，再オープンされる割合，優先度との関係，再オープンされる根本的な原因の調査を行った [21]．調査の結果，再オープンされる割合は 6–10%であり，再オープンされた欠陥のうち 93%以上がシステムの通常動作に深刻な影響を与える欠陥であることを明らかにした．また，再オープンされる根本的な原因のうち，約半分が修正の失敗以外によるものであることを明らかにした．

自動プログラム修正に関連するデータセット

本調査で用いた Defects4J データセット [9] 以外にも，自動プログラム修正に関連するデータセットが存在する．

Do らは，Java,C,C++,C#のプログラムを収録した SIR データセットを公開している [41, 42]. SIR データセットは主にプログラム合成を行うツールの評価に用いられている [8, 18].

Le Goues らは，GenProg の研究において，ツールを公開するだけでなく，C 言語の 105 個の欠陥を収集したデータセットも公開している [2]．このデータセットは様々な C 言語向けのツールの評価に用いられている [3, 4, 11, 32]．また，このデータセットを拡張した，185 個の欠陥を収集した ManyBugs データセット，学生のプログラミング課題に対する提出から得られた欠陥を収集した IntroClass データセットも公開している [43, 44].

Böhme らは，C 言語の 70 個の欠陥を収録した CoREBench データセットを公開している [45, 46]. CoREBench データセットにはリグレッションテストによって発生したエラーが含まれており，Angelix の評価に用いられている [4].

自動プログラム修正に関するサーベイおよびウェブサイト

自動プログラム修正に関するサーベイは，Monperrus らによるテクニカルレポートがある [47]．この報告では，2015 年秋頃までの 178 本の研究が取り上げられており，自動プログラム修正だけでなく，デバッグ自動化に関する研究全般に触れている．また，自動プログラム修正の分類や各論文で用いられる用語の表記ゆれについても言及されている．

また，近年 GenProg をはじめ [48]，様々な手法に関してツールの公開に伴うウェブサイトが開設されている [49, 50]．ウェブサイトの公開は研究の直接的な貢献とは言えないが，論文やツール，実験の再現パッケージ等に用意にアクセスできることは，研究者だけでなく，非研究者も自動プログラム修正を理解する助けになる．さらに，Mechtaev らによる最近の自動プログラム修正に関する論文，ツール，ベンチマークをまとめたポータルサイト [10] も登場しており，自動プログラム修正に関する様々な研究へアクセスしやすくなっている．

8 調査の妥当性に対する議論

8.1 内的妥当性に対する議論

本研究では，修正時間を評価する際にツールの実行時間を用いているが，修正に失敗している場合のツールの実行時間を考慮していない．ただし，修正に失敗している場合のツールの実行時間は，ツールを強制停止させるタイムアウト時間の影響を受けてしまう．また，タイムアウト時間を設けない場合，場合によって現実的な時間内に動作を停止しないツールもある [2, 5]．そのため，ツールの真の実行時間を考慮できないことは承知の上で，修正に成功している場合のみを採用している．

8.2 外的妥当性に対する議論

本研究の調査対象においては，65%以上が優先度 Major であった．一方，優先度 Critical 以上の欠陥は 6%程度，再オープンされた欠陥は 6%程度であった．そのため，Blocker，Critical，および再オープンされた欠陥の追加を試みた．欠陥の追加後の優先度 Critical 以上の欠陥は約 9%，再オープンされた欠陥は約 10%に上昇したが，Blocker の欠陥を 1 つも収集することができなかった．そのため，調査対象のプロジェクトを追加した収集が必要である．現在の調査対象を拡張するには，Lang や Math プロジェクトと同じ優先度の定義を持つプロジェクトを探す必要がある．

本研究では欠陥の特徴として優先度および再オープンの有無を用いた．優先度および再オープンの有無はどちらも欠陥レポートから得られる特徴である．一方，欠陥レポートを書いているのは欠陥の報告者および開発者であり，欠陥レポートの書き方は開発者によって様々である．そのため，開発者によって優先度の付け方が異なる可能性がある．例えば，優先度には一定の定義 [21] があるにもかかわらず，開発者によっては 5.2 節のように定義と異なる優先度をつけることがある．また，本調査とは異なるデータセットではあるが，開発者による再オープンの原因の約半分が修正の失敗以外の原因で行われているという調査が存在する [21]．そのため，本調査に用いたデータセットでも修正の失敗以外の原因で再オープンが行われている可能性がある．

9 おわりに

本研究では，オープンソースの Java プロジェクトにおける 170 個の欠陥を収集したデータセットに対し，欠陥追跡システムの欠陥レポートから得られる欠陥の特徴を結びつけ，既存の自動プログラム修正の修正性能を評価した．調査の結果，jGenProg と Nopol が優先度の高い欠陥を多く修正できること，Nopol が再オープンされた欠陥を全く修正できないことを明らかにした．また，優先度別，再オープンの有無別に，修正時間や修正差分の行数を調査し，自動プログラム修正が全く修正できなかった優先度 Trivial の欠陥に対する目視分析を行った．さらに，Critial または再オープンされた欠陥を 10 個追加した調査を行い，統計的な比較を行った．

本調査は，自動プログラム修正の評価方法に，欠陥の特徴という新たな視点を与えた．既存の評価方法としては，修正個数の他，修正時間 [2]，人間にとって読みやすいか [7]，シンプルなパッチになっているか [8] など，ツールの出力を分析するものが主であった．一方で，ツールへの入力となる欠陥が持つ特徴から，ツールの性能を分析する研究はあまり見られなかった．今後，研究者が新たな自動プログラム修正を提案し，評価を行なう際には，修正個数や修正時間だけでなく，欠陥の特徴を評価に取り入れることを推薦する．欠陥の特徴を評価に取り入れることで，多角的に評価を行うことができる．

我々は，調査で得た知見から，実際のソフトウェア開発における以下の場面での自動プログラム修正の活用を提案する．

- 優先度の高い欠陥に対して jGenProg や Nopol を実行することで，高い割合で修正することができる．
- デバッグの後，バージョン管理システムへコミットする前に自動プログラム修正を実行することで，開発者による差分と，自動プログラム修正による差分の違いから，再オープンに繋がるミスを防ぐ機会を得ることができる．

今後の研究課題としては，欠陥の数および欠陥の特徴を増やした評価や，対象言語およびツールの種類を増やした評価等が挙げられる．

謝辞

本研究を行うにあたり，常に励まして頂き，暖かく見守っていただきました 楠本 真二 教授に心より感謝申し上げます．

本研究を行うにあたり，日頃より熱心なご指導ご鞭撻を頂き，活発な議論に応じていただきました肥後 芳樹 准教授に心より感謝申し上げます．

本研究に対して新しい視点からのご意見を頂き，研究室生活の中で相談に乗っていただきました柗本 真佑 助教 に心より感謝申し上げます．

日々の研究室生活において，互いに切磋琢磨し，高め合い，ときに白熱した議論を繰り広げました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の 小倉 直徒 氏，同 佐飛 祐介 氏，同 鷲見 創一 氏，同 古田 雄基 氏，同 幸 佑亮 氏に心より感謝いたします．

日々の研究室生活の中心となり，様々な役職で我々の生活を支えてくださりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 1 年の 下仲 健斗 氏，同 中島 弘貴 氏，同 山田 悠斗 氏，同 山本 将弘 氏に心より感謝いたします．

短い間でしたが，日々の議論を通じて共に成長を実感し合いました大阪大学基礎工学部情報科学研究科 4 年の 有馬 諒 氏，同 佐々木 美和 氏，同 谷門 照斗 氏，同 松尾 裕幸 氏，同 山田 涼太 氏に心より感謝いたします．

また，本研究に関して多くのご助言を頂くとともに，様々な面において親切なご助力，ご協力を頂きました楠本研究室の皆様に心より感謝いたします．

最後に，本研究に至るまでに，講義やセミナー等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます．

参考文献

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[2] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 3–13, 2012.

[3] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 166–178, 2015.

[4] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 691–701, 2016.

[5] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, Demonstration Track*, pp. 441–444, 2016.

[6] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, pp. 1–25, 2016.

[7] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 802–811, 2013.

[8] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, pp. 448–458, 2015.

[9] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[10] Sergey Mechtaev. Program repair. `http://program-repair.org`.

[11] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 24–36, 2015.

[12] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[13] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[14] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 356–366, 2013.

[15] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, Vol. 185, pp. 825–885, 2009.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[17] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the International Conference on Computer Aided Verification*, pp. 171–177, 2011.

[18] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 772–781, 2013.

[19] Atlassian. Jira software - issue & project tracking for software teams. `https://www.atlassian.com/software/jira`.

[20] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pp. 249–258, 2010.

[21] Qing Mi and Jacky Keung. An empirical analysis of reopened bugs based on open source projects. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 37:1–37:10, 2016.

[22] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, pp. 1–29, 2016.

[23] Apache. Math - commons math: The apache commons mathematics library. `http://commons.apache.org/proper/commons-math`.

[24] Github. About issues - user documentation. `https://help.github.com/articles/about-issues`.

[25] Apache. Lang - home. `http://commons.apache.org/proper/commons-lang`.

[26] ASF JIRA. [MATH-393] Method 'getResult()' in 'MultiStartUnivariateRealOptimizer'. `https://issues.apache.org/jira/browse/MATH-393`.

[27] Apache. apache/commons-lang: Mirror of apache commons lang. `https://github.com/apache/commons-lang/`.

[28] Apache. Math - developers guide. `http://commons.apache.org/proper/commons-math/developers.html`.

[29] Michael P Fay. Two-sided exact tests and matching confidence intervals for discrete data. *R journal*, Vol. 2, No. 1, pp. 53–58, 2010.

[30] Haruhiko Okumura. Fisher の正確検定. `https://oku.edu.mie-u.ac.jp/~okumura/stat/fishertest.html`.

[31] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 295–306, 2015.

[32] 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*, pp. 298–312, 2016.

[33] Martin 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. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 234–242, 2014.

[34] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pp. 213–224, 2016.

[35] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pp. 837–842, 2017.

[36] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, 2017. `https://arxiv.org/pdf/1608.07754.pdf` (採録決定).

[37] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317, 2014.

[38] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 492–495, 2014. track New Ideas and Emerging Results.

[39] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, Vol. 20, No. 1, pp. 176–205, 2015.

[40] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, Vol. 18, No. 5, pp. 1005–1042, 2013.

[41] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, Vol. 10, No. 4, pp. 405–435, 2005.

[42] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Software-artifact infrastructure repository: Home. `http://sir.unl.edu/portal/index.php`.

[43] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, Vol. 41, No. 12, pp. 1236–1256, 2015.

[44] Westley Weimer, Stephanie Forrest, and Claire Le Goues. Manybugs and introclass. `http://repairbenchmarks.cs.umass.edu`.

[45] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 105–115, 2014.

[46] Marcel Böhme. Corebench. `https://www.comp.nus.edu.sg/~release/corebench/`.

[47] Martin Monperrus. Automatic software repair: a bibliography. Technical report, University of Lille, 2015.

[48] Westley Weimer, Stephanie Forrest, and Claire Le Goues. Genprog | evolutionary program repair. `http://dijkstra.cs.virginia.edu/genprog/`.

[49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix. `http://angelix.io`.

[50] Fan Long and Martin Rinard. Automatic patch generation via learning. `http://groups.csail.mit.edu/pac/patchgen/`.