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.

自動テスト生成を利用した自動プログラム修正出力パッチ分類の試み

藤本　章良† 　肥後　芳樹† 　柗本　真佑† 　楠本　真二† 　安田　和矢††

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
†† (株) 日立製作所 研究開発グループ 社会システムイノベーションセンタ
〒 244-0817 神奈川県横浜市戸塚区吉田町 292 番地
E-mail: †{a-fujimt,higo,shinsuke,kusumoto}@ist.osaka-u.ac.jp, ††kazuya.yasuda.fd@hitachi.com

あらまし　ソフトウェア開発において効率的なデバッグを行うため，自動でプログラムを修正するツールが数多く提案されている．自動プログラム修正ツールは，バグを含むプログラムとテストスイートを与えると，そのテストスイートを通過するようにプログラムを修正したパッチを出力する．しかし，与えられたテストスイートに過剰に適合した結果，完全にバグを修正しきれていない状態のパッチが出力される場合が多く，開発者によるパッチの確認が必要である．また自動プログラム修正ツールは，修正候補として複数のパッチを出力する場合があり，パッチ数が多い場合はそれらの確認に多大なコストを要する．そこで本研究では，パッチの確認コスト削減を目的とし，自動テスト生成技術を利用して，パッチを振る舞いごとにグループ化する手法を提案する．振る舞いが同じパッチのグループの中から 1 つのパッチを選択し確認することで，そのグループに含まれるパッチ全ての正誤を判断でき，目視によるパッチの確認回数を削減可能である．提案手法では，各パッチに対して自動テスト生成を行い，パッチと生成したテストを相互に実行し，この結果を用いて分類を行う．本稿では，提案手法の評価のため 1) 確認すべきパッチ数の削減割合，2) 正誤の異なるパッチの別グループへの分離可能性について調査を行なった．その結果，提案手法は確認すべきパッチ数を平均で 90%削減し，調査対象とした 41 個の題材のうち，26 個で分離に成功した．
キーワード　自動プログラム修正，パッチ評価，自動テスト生成

1. まえがき

自動プログラム修正（Automated Program Repair: APR）とは，バグを含むプログラムから自動的にバグを取り除く技術である．ソフトウェア開発に必要なコストのうち，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]．APR ツールの利用によって，こうしたデバッグ作業を効率化する試みがなされている．現在，様々な APR ツールが提案されている [3] [4]．これらのうちテストベースの APR ツールは入力としてバグを含むプログラムの他にテストスイートを必要とする．入力するテストスイートをそのプログラムが満たすべき仕様とし，APR ツールはそのテストスイートを通過するようにプログラムを修正したパッチを出力する．

これらの APR ツールは，入力するテストスイートに過剰に適合した結果，全てのテストケースに通過するものの，完全にバグを修正しきれていない状態であるオーバーフィットなパッチを出力する場合が多い [5] [6]．したがって，APR ツールの利用者は，出力されたパッチが真に正しいかを目視で確認する必要がある．しかしながら，APR ツールが大量のパッチを出力する場合，あるいは複数の APR ツールを同時に利用する場合に確認すべきパッチの数が非常に多くなり，全てのパッチを確認するために多大なコストが生じる．そのため APR ツールを用いたとしても，必ずしもデバッグ作業を効率化できるとはいえない．

真にデバッグを効率化するには，オーバーフィットなパッチを生成しないこと，APR ツールが出力するパッチの確認コストを削減することの 2 通りの方法が考えられる．しかし現時点では，APR ツールが出力するパッチの大半をオーバーフィットなパッチが占めており [6] [7]，オーバーフィットのない APR ツールの実現はまだまだ難しい．

そこで本研究では，パッチ確認コストの削減を目的とし，APR ツールが出力したパッチを振る舞いごとにグループ化する手法を提案する．振る舞いが同じパッチのグループから 1 つのパッチを選択し確認することで，そのグループに属する他の全てのパッチが正しいか否かについても同時に判断でき，確認すべきパッチの個数を削減できる．提案手法ではパッチのグループ化に自動テスト生成技術を用いる．自動テスト生成技術は，与えたプログラムの振る舞いからテストを生成するため，生成されたテストに他のパッチが通過可能かどうかで振る舞いの一致を判断できると著者らは考えた．全てのパッチからテストを自動生成し，パッチと生成テストの全ての組を相互に実行する．この結果を用いて，パッチのグループ化を行う．

本稿では，Defects4J Math プロジェクトのバグに APR ツールを適用して得られたパッチに対して提案手法を適用し，1) 確認すべきパッチ数の削減割合，2) 正誤の異なるパッチの別グルー

プへの分離可能性 の 2 点について評価実験を行なった．その結果，提案手法により確認すべきパッチの数が，平均で 84%，中央値で 90%削減可能であることが明らかになった．また，出力されたグループ化の結果を目視で確認したところ，半数以上の題材において，正誤の異なるパッチを別のグループに分離することができた．

2. 背景と関連研究

2.1 自動テスト生成

自動テスト生成とは，与えたプログラムに対してその振る舞いを最大限網羅できるような入力値を持つテストを生成する技術である．現在多くの自動テスト生成技術が研究されており，ランダムな入力からテストを生成する Randoop [8] や遺伝的アルゴリズムを用いる EvoSuite [9] など様々な手法が提案されている．

2.2 自動プログラム修正（APR）

自動プログラム修正とは，バグを含むプログラムから自動でバグを取り除く技術である．言い換えれば，一部の仕様を満たしていないプログラムに対して，APR は全ての仕様を満たすようにプログラムの振る舞いを変更する．APR はソースコードを改変しプログラムの振る舞いを変更する Behavioral repair と，実行中のプログラムに変更を加える State repair の 2 種類に大別される [4]．例えば前者は，バグを含むプログラムとともにテストスイートやステートマシンを仕様として与えると，これらを満たすように変更を加えたプログラムをパッチとして出力する．本研究では特に，テストスイートを仕様として利用する APR を想定する．

2.3 オーバーフィットなパッチ

APR が出力するパッチの中で，与えた仕様に過剰に適合し完全にバグを修正しきれていない，または別のバグが混入したパッチをオーバーフィットなパッチと呼ぶ．テストベースの APR の場合，オーバーフィットなパッチは与えたテストスイートは全て通過するが，十分に一般化ができていない状態である [5]．オーバーフィットなパッチの存在は APR において大きな問題とされている．オーバーフィットなパッチは，APR が出力するパッチの大多数を占めており [6] [7]，バグが修正されるよりもオーバーフィットによって既存の正しい機能が破壊される場合が多いこと [10] が調査によって明らかになっている．

2.4 APR 生成パッチ評価

APR 利用者の負担を軽減するため，本研究以外にも APR 出力パッチの評価に関する研究が存在する．特に個々のパッチがオーバーフィットかを評価する研究が広く行われている [11]．それらの研究は，正解パッチの利用の有無で 2 つに分けられる．

正解パッチを用いない手法として，Xiong らはバグを含むプログラムから自動テスト生成技術を用いてテストスイートを生成し，パッチの正誤を分類した [12]．各パッチに生成したテストを適用し，バグを含む状態のプログラムとの実行時の振る舞いの類似度，およびテストの実行経路を利用する．

正解パッチを利用する手法には，RGT（Random testing in Ground Truth）を用いたパッチ評価手法がある．RGT とは，正解パッチから自動生成されたテストスイートであり，RGT に失敗するパッチをオーバーフィットと判断する．Ye らは，OSS で発生したバグに対して APR が出力したパッチの成否を，RGT を用いて分類した [13]．DiffTGen [14] は，バグを含むプログラムからテストケースを生成し，パッチの適用前後で結果が異なるテストケースを得る．正解パッチと生成パッチのテスト実行結果が異なれば，そのパッチをオーバーフィットであると判断する．

```
public Complex add(Complex rhs)
  throws NullArgumentException {
  MathUtils.checkNotNull(rhs);
+ if (isNaN || rhs.isNaN) {
+     return NaN;
+ }
  return createComplex(real + rhs.getReal(),
      imaginary + rhs.getImaginary());
```

(a) 正しい修正パッチ

```
public Complex add(Complex rhs)
  throws NullArgumentException {
+ if (isNaN || rhs.isNaN) {
+     return NaN;
+ }
  MathUtils.checkNotNull(rhs);
  return createComplex(real + rhs.getReal(),
      imaginary + rhs.getImaginary());
```

(b) オーバーフィットなパッチ 1

```
public Complex add(Complex rhs)
  throws NullArgumentException {
- MathUtils.checkNotNull(rhs);
+ if (isNaN || rhs.isNaN) {
+     return NaN;
+ }
  return createComplex(real + rhs.getReal(),
      imaginary + rhs.getImaginary());
```

(c) オーバーフィットなパッチ 2

```
public Complex add(Complex rhs)
  throws NullArgumentException {
- MathUtils.checkNotNull(rhs);
+ if (isNaN || rhs.isNaN) {
+     return NaN;
+ }
+ MathUtils.checkNotNull(rhs);
+ if (isNaN) {
+     return NaN;
+ }
  return createComplex(real + rhs.getReal(),
      imaginary + rhs.getImaginary());
```

(d) オーバーフィットなパッチ 3

図 1 Math 53 のバグに対し APR が出力したパッチ

3. 研究動機

先行研究 [15] において，Defects4J データセット [16] に含まれるバグ（Math 53）に対して複数の APR ツールを適用したところ，複数のパッチが得られた．出力されたパッチには，正しいパッチとオーバーフィットなパッチの両方が含まれる．一部のパッチを図 1 に示す．図 1(a) は，実際に行われた修正と一致する正解のパッチであり，図 1(b)(c)(d) はオーバーフィットなパッチである．オーバーフィットなパッチは引数 `rhs` の `null` チェックを行う前に `rhs` を参照しているため，引数に `null` を与えると例外が発生する．同様に `null` を与えると例外が発生するパッチが複数確認された．これらのパッチは “引数に `null` を与えると例外が発生する” 振る舞いを持つパッチであるといえる．

このような同様の振る舞いを持つパッチをひとつのグループとして利用者に提示することで，確認すべきパッチの数を削減できる．すなわち，あるグループに属するパッチが “引数に `null` を与えると例外が発生する” オーバーフィットなパッチであることが分かれば，そのグループの他のパッチも同様にオーバーフィットであることがわかるため，他のパッチは確認する必要がなくなる．出力された大量のパッチをいくつかの少数のグループに分割できれば，利用者の確認コストは大幅に削減可能である．

4. 提案手法

本研究の目的は，APR が出力するパッチの確認に必要なコストの削減である．本研究では，振る舞いが同じパッチをグループ化する手法を提案する．各グループから 1 つのパッチを選択し確認することで，確認すべきパッチの数を削減することを目指す．

提案手法の概要を図 2 に示す．提案手法への入力は APR が出力したパッチであり，出力はグループ化されたパッチである．

グループ化のプロセスは以下の 3 つのステップで構成される．

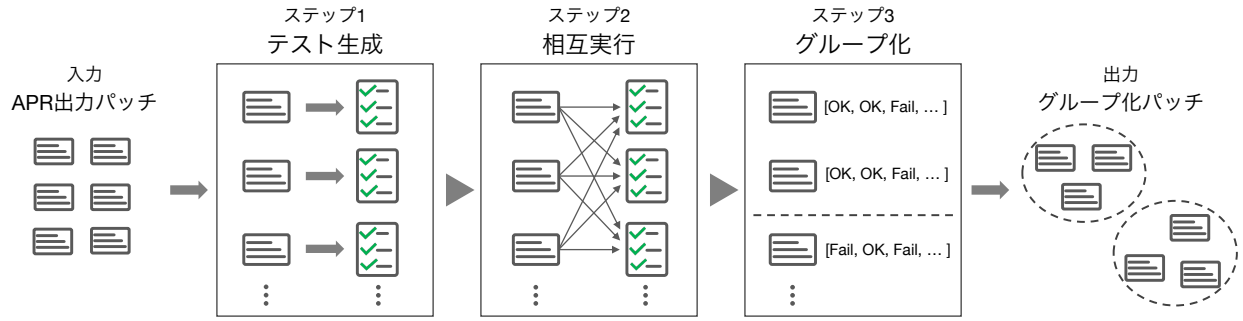

図 2　提案手法の概要

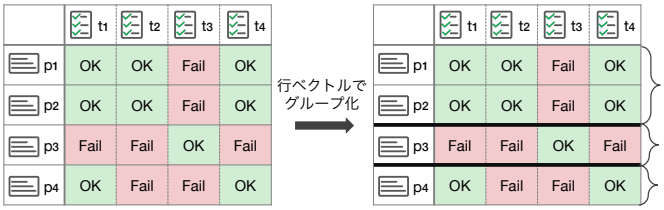

図 3　パッチの相互実行とグループ化

ステップ 1　各パッチからテストスイート生成
ステップ 2　パッチと生成テストスイートの相互実行
ステップ 3　パッチのグループ化
以降では，各ステップについて説明する．

ステップ 1：各パッチからテストスイート生成

入力として与えられたパッチ $p_1, ..., p_n$ をそれぞれ修正前のプログラムに対して適用し，全てのパッチ適用済みプログラム（以降では，パッチ適用済みプログラムを単にパッチと呼ぶ）から自動テスト生成技術を用いてテストスイートを生成する．生成されたテストスイートをそれぞれ $t_1, ..., t_n$ とする．

次に，生成された各テストスイートに対して flaky test を含むかどうかチェックする．flaky test とは，同じプログラムであるにも関わらず，実行するたびにテストの成否が変化するテストである．flaky test は開発に悪影響を与えるテストであることが知られており [17]，本手法の有効性が低下する可能性があるため，相互実行するテストから取り除く．p_i が t_i に含まれるテストケースに失敗した場合，そのテストケースを flaky test であると判断する．各パッチは自身から生成した全てのテストケースに通過するはずであるが，失敗するのは実行ごとの成否の変化が原因と考えられるためである．

ステップ 2：生成テストとパッチの相互実行

ステップ 1 で自動生成したテストと APR が出力したパッチを全ての組み合わせで相互に実行し，結果を記録する．最終的に図 3 左のように，パッチと自動生成テストの実行結果が記録される．図中の “OK” は t_i に含まれるテストケースを全て通過，“Fail” は 1 つ以上のテストケースに失敗したことを表す．

ステップ 3：パッチのグループ化

ステップ 2 で得た相互実行の結果からパッチのグループ分けを行う．自動テスト生成技術を用いて得られるテストは，与えたプログラムの振る舞いをテストスイートとして規定していると捉えることができる．したがって，パッチ p_k，p_l がそれぞれから生成したテスト t_k，t_l を互いに通過すれば，パッチ p_k および p_l は同じ振る舞いを持つパッチであるとみなせる．一方，パッチ p_k がテスト t_l に失敗，またはパッチ p_l がテスト t_k に失敗した場合，p_k と p_l は異なる振る舞いを持つパッチであると判定する．パッチの振る舞いが異なる場合でも，必ずしも互いに失敗するテストが生成される訳ではないため，どちらか一方のみが失敗した場合でも振る舞いが異なるパッチとして扱う．

グループ化には，パッチ p_k とテストスイート $t_1, ..., t_n$ の成否を用いる．p_k と各テストの成否は，要素数 n で通過または失敗の 2 値を持つベクトルであるとみなすことができ，相互実行結果の行に対応する．このベクトルは，p_k がどのような特徴を持つかを示すベクトルと見ることができる．すなわちベクトルの各要素について，テストスイート t_j がある振る舞いの特徴を表しており，p_k が t_j に通過した場合その特徴を持ち，失敗した場合はその特徴を持たない．p_k と p_l のテスト実行結果のベクトルが完全に一致する場合，p_k と p_l は同じ振る舞いの特徴を持つパッチであると判定する．p_k が t_k に通過するため，p_k と p_l のベクトルが完全一致すれば p_k と p_l が互いのテストに通過することが保証される．最終的にベクトルが一致するパッチの組を全て探し，グループ化したパッチとして結果を出力する．図 3 の例では p_1 と p_2 の行ベクトルが一致し，これらは同じグループに分類される．一方 p_3 と p_4 のベクトルはそれぞれ異なっており，出力されるグループ数は 3 となる．

5. Research Questions

提案手法の評価を行うにあたり，以下 3 つの Research Question（RQ）を設定した．

RQ1: 提案手法により確認すべきパッチ数はどの程度減少するか？

入力として与えたパッチ数と，出力されるパッチのグループ数の関係について調査する．入力パッチ数と出力グループ数が同じであれば，パッチの確認コストは削減されない．提案手法がパッチをある一定程度のグループ数に分割できる能力を持ち，確認コストを削減可能であることを示すため，パッチ数に対してどの程度のグループ数になるかを調査する．

RQ2: 提案手法は正解・オーバーフィットなパッチをそれぞれ別のグループに分離できるか？

提案手法を利用してパッチを確認する場合，各グループから代表パッチを 1 つ選びそのパッチのみ確認を行う．そのためグ

ループ全体が正解かどうかを判定するためには，各グループを構成する全てのパッチが正解またはオーバーフィットであることが必要である．この RQ では提案手法の有効性を示すため，出力された各グループが全て正解またはオーバーフィットなパッチのみから構成されているかを目視で調査する．

RQ3: 正誤の異なるパッチ分類に影響を与える特徴はあるか？

バグの種類や修正方法，修正対象のプログラムのメトリクスといった特徴が，グループ化に影響を与えるかどうかを調査する．特に，正解パッチの分離可能性との関連性について分析する．

6. 評価実験

6.1 実験対象

実験対象は Defects4J [16] の Math プロジェクトに含まれるバグである．APR ツールが出力したパッチとして，Durieux らの先行研究 [15] の過程で生成したパッチのデータセット(注1)を利用した．このデータセットには，11 種類の APR ツールを適用して生成されたパッチが収録されており，1 つのバグに対して複数のパッチが生成されたケースも含まれている．本評価実験では，Defects4J Math プロジェクトにおいて，いずれかの APR ツールで 1 つ以上の修正パッチが生成されたバグを題材とした．タイムアウトにより EvoSuite でテストを生成できなかった Math 80, 81 は除外した．対象の題材は合計で 41 個である．

また各題材について，正解パッチとして人の手によって行われた実際の修正パッチも加えて実験を行なった．これは，APR が出力するパッチに正解が含まれていない場合が往々にして存在するためである．正解パッチを加えることで，正解とオーバーフィットなパッチの両方を含むバグ修正事例を増やし，特に RQ2 で調査する正解パッチの分離可能性を評価しやすくする．正解パッチとして，Sobreira ら [18] のデータセット(注2)を利用した．

6.2 実験設定

提案手法では自動テスト生成ツールとして，EvoSuite [9] を利用した．ステップ 1 で EvoSuite にシード値 1 から 5 を与え，各パッチあたり 5 回分テストを生成した．flaky test のチェックは各テストケースにつき 1 回実行した．合計で 808,218 個のテストケースを生成し，うち 1,216 個（0.15%）が flaky test だった．

6.3 実験結果

RQ1: 確認すべきパッチ数はどの程度減少するか？

各題材について，削減率 = 1 - (提案手法が出力するグループ数 / 生成パッチ数) を求める．これは，確認すべきパッチ数をどの程度削減できるか示す値である．この値が大きいほど確認すべきパッチ数が減少していることを示す．この RQ では，生成されたパッチ数が 10 個より多い題材のみを対象とした．これは，出力パッチ数が多く利用者による確認がより困難な場合に，提案手法が有効に働くことを示すためである．その結果を図 4 に示す．削減率は平均で 84%，中央値 90%であり，確認すべきパッチ数を大幅に削減できることがわかった．

Math 58 は他の題材に比べて削減率が低い．テストの実行結果を確認したところ，テストのタイムアウトが影響していると考察された．テストの実行時間がタイムアウト時間と同程度の場合，振る舞いの同じパッチが同じテストを実行したにもかかわらず，実行時間のわずかな差によりテスト結果が異なる場合がある．実際に，1 つのパッチに対し複数回同じテストを実行すると，通過する場合とタイムアウトにより失敗する場合の両方がみられた．これにより，同じ振る舞いのパッチが別のグループに分類された可能性が高い．タイムアウトの上限値を増やす，またはタイムアウトを無くすことにより，少数のグループを形成できる可能性がある．

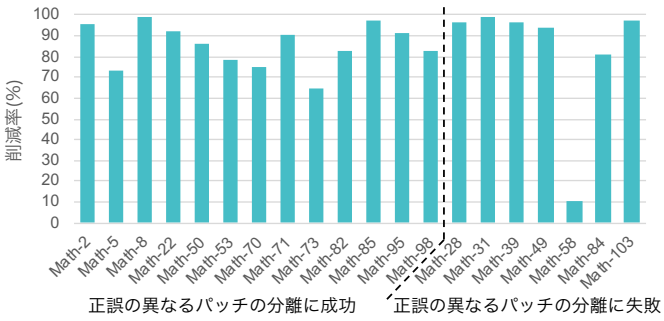

図 4　題材ごとの確認すべきパッチの削減率

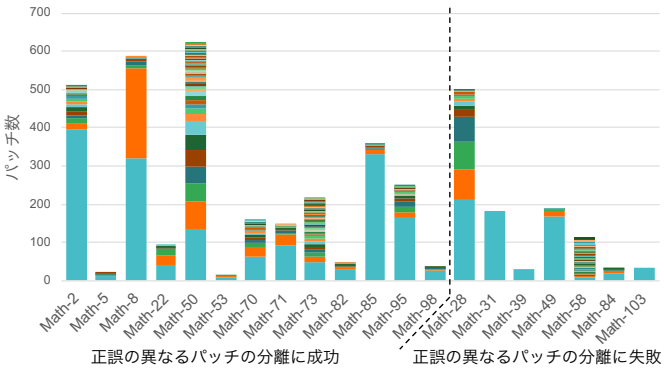

図 5　グループの分割状況

次に各題材で生成されたパッチを提案手法を用いて分割した状況を図 5 に示す．グラフの各色が 1 つのグループを表している．各題材で，いくつかのグループに分割されていることがわかる．特に，各グループに含まれるパッチ数は一定ではなく，1 つの大きいグループと複数の小さいグループに分割されている．これは，APR ツールが出力するパッチは共通する部分が多いことが原因として考えられる．バグを含むプログラムを修正するには，失敗するテストを通過させるために特定の処理が行われやすい．例えば Math 85 のほとんどのパッチは，例外を発生させないようスロー文を削除する操作が共通している．パッチによってはそれ以外の箇所も変更しているが，それは必ずしも振る舞いに影響を与えない．よって，大部分のパッチが同じ振る舞いをし，一部のパッチが異なる振る舞いをすることにより，図 5 のような結果になったと考えられる．

RQ1 への回答：提案手法は確認すべきパッチの数を，平均で 84%，中央値で 90%削減可能であり，APR ツールの利用者がパッチを確認するコストを大幅に削減可能である．

(注1)：https://github.com/program-repair/RepairThemAll_experiment
(注2)：https://github.com/program-repair/defects4j-dissection

RQ2: 正解・不正解パッチを別のグループに分離できるか？

各題材について，提案手法を適用し得られた各グループに含まれるパッチが正解かどうか目視で確認する．しかし生成されたパッチは非常に多いため，全てを目視確認することは現実的でない．そこで，グループに含まれるパッチ数が 10 個以下であればその全てを確認し，10 個以上の時は 10 個を下限としその 10%を確認した．また，予め加えた正解パッチと同じグループに属するパッチは全て目視で確認を行った．

目視確認の結果を表 1 に示す．また，図 6 に表 1(a)〜(e) の各分割状況を示す．例として (a) は，APR ツールはオーバーフィットなパッチのみ生成しており，あらかじめ加えた実際の修正パッチは，オーバーフィットなパッチとは別のグループに分離されている．表 1 の通り，全 41 個の題材のうち 26 個の題材において生成されたパッチの正誤による分離に成功した．分離ができなかった題材について，正解の振る舞いを持つパッチが 2 グループ以上に分けられた事例 (e) は Math 58 の 1 つのみだった．RQ1 で調査したように，同じテストが成功したり失敗したりするため，振る舞いが同じでも異なるグループに分けられることが原因と考えられる．

(c)，(d) の 14 個の題材について調査したところ，APR ツールが修正すべき箇所を含むクラス以外も変更している題材が 7 個存在した．今回の実験では，修正すべき箇所を含むクラスのみからしかテストを生成しなかったため，それ以外の箇所の変更による振る舞いの違いを検出できなかったと考えられる．APR ツールが修正した全てのクラスを対象にテストを生成すれば，適切にパッチを分類できる可能性がある．

分離に失敗する題材はほとんどが APR ツールによって正解を出力できないバグ (c) であった．APR ツールが正解パッチを出力できた題材である (b), (d) を比較すると分離に成功できた題材 (b) の方が多い．この結果から APR ツールが修正しやすいプログラムは，振る舞いの違いを検出しやすいと考察される．

RQ2 への回答：提案手法は半数以上の題材で正解パッチとオーバーフィットな題材を別のグループに分離することができた．さらにテスト生成対象を拡大することにより，分離精度を改善する余地も残されている．

表 1　正誤の異なるパッチの分離結果

分離に成功	(a)APR 正解なし	20
	(b)APR 正解あり	6
分離に失敗	(c)APR 正解なし	13
	(d)APR 正解あり	1
	(e) 正解が複数に分割	1
合計		41

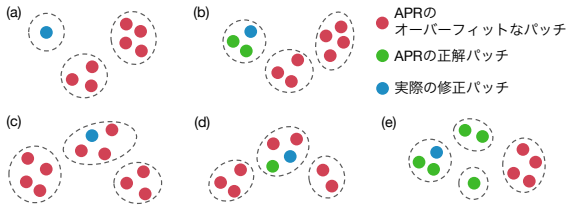

図 6　表 1(a)〜(e) と分割結果の対応

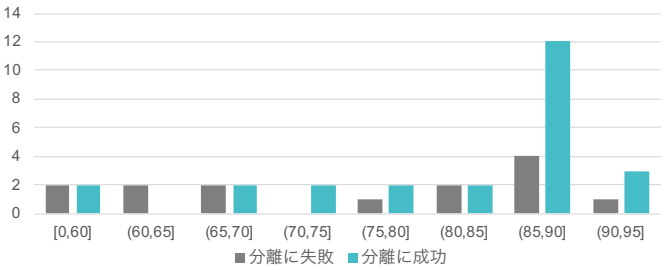

図 7　分離の成否と EvoSuite のテストカバレッジの分布

RQ3: 正誤の異なるパッチ分離に影響を与える特徴はあるか？

正誤の異なるパッチの分離に影響を与える要素を調査した．調査項目として，各題材におけるバグの修正行数および修正箇所・EvoSuite が生成したテストのカバレッジ・修正対象メソッドのメトリクス（行数・循環的複雑度・NPath 数）を計測した．

正誤の異なるパッチの分離について，分離に成功した題材と失敗した題材別でまとめた結果を表 2 に示す．EvoSuite のテストカバレッジに注目すると，テストカバレッジが高いほど分離しやすくなることがわかる．図 7 は各題材ごとのテストカバレッジのヒストグラムを示している．分離に失敗した題材よりも，分離に成功した題材の方がカバレッジの分布が高い．85%〜90%が最も多く，失敗に分離した題材より 3 倍多いことがわかった．テストのカバレッジが高いことは，テストがパッチの振る舞いを十分に網羅できていることを表しており，正解パッチとオーバーフィットなパッチを分離しやすくなっていると考えられる．

次に，生成パッチ数に注目すると分離に成功した題材の方が，パッチ数が多い．パッチ数が多いことにより，生成するテストの数も多くなった結果，分割が細分化され正誤の異なるパッチが別のグループに分離されやすくなった可能性がある．

表中の “修正の特徴” は，正解パッチで行われた修正内容から得た値である．分離に失敗する題材は修正行数が多いが，修正箇所に差はなかった．バグに行われた修正の特徴がパッチの分離に与える影響は大きくないと考えられる．

“メソッドのメトリクス” は，バグを含むメソッドを対象に計測した値である．NPath とは，実行可能な経路の総数であり，分岐が増えるほど値が大きくなる．こちらは分離に成功する題材の方が，行数，循環的複雑度，NPath の全てで高い値となった．より複雑なメソッドの方が入力に対して様々な振る舞いを持つようになり，自動生成したテストも異なる振る舞いを網羅できるようになるため，分離がしやすくなった可能性がある．

最後に，RQ1 で調査した確認すべきパッチの削減率やグループの分割状況と，正誤の異なるパッチの分離可能性の関係を調査した．図 4，5 中の点線の左側は正誤の異なるパッチの分離に成功した題材，右側は失敗した題材である．パッチの削減率，分割状況ともに目立った特徴は得られなかった．したがってパッチの削減率や分割状況から，正誤の異なるパッチの分離が適切に行われているかどうかを推定することは，難しいといえる．

RQ3 への回答：自動生成したテストのカバレッジ，および修正対象のメソッドのメトリクスが，正誤の異なるパッチの分離に影響を及ぼす可能性があることが調査結果より示唆された．

7. 妥当性の脅威

本研究の評価実験では，テストの生成に EvoSuite を用いた．他の自動テスト生成技術，あるいは手動でテストケースを追加した場合には異なる実験結果が得られる可能性がある．

RQ2 において，出力された各グループが正解パッチまたはオーバーフィットなパッチのみで構成されるかを，目視で確認した．そのため著者の主観に大きく依存する．他の APR ツール利用者が判断した場合，異なる結果となる可能性がある．

本提案手法は，パッチを生成する APR ツール全てに対して利用可能である．しかし，限られた種類の APR ツールが出力したパッチに対して実験を行ったため，それ以外の APR ツールが出力したパッチで提案手法が有効に働くかどうかは不明である．また，バグの題材として Defects4J を用いたが，別のバグに対して適用した場合，同様の結果が得られるとは限らない．

8. あとがき

本研究では，APR が出力したパッチの確認コスト削減を目的に，パッチを振る舞いごとにグループ化する手法を提案した．提案手法では，自動テスト生成技術を用いて各パッチからテストを生成し，全てのパッチと生成したテストの組み合わせで相互に実行した結果を用いて分類を行う．評価実験を行った結果，提案手法は確認すべきパッチ数を平均で 84%，中央値で 90%削減できた．また，半数以上の題材で正解のパッチとオーバーフィットなパッチを別のグループに分離することができた．

今後の研究課題として，どのグループがよりオーバーフィットを起こしている可能性が高いか，またあるグループの中でどのパッチを優先的に確認すべきかを提示できるようにすることを考えている．これらの情報を利用者に提示することで，確認のコストをさらに削減できる可能性がある．これを実現するために，分離精度の更なる向上，および提示すべきパッチを選択するために，グループ化や正解パッチの分離に影響を与える特徴の分析を考えている．特徴の分析については，本研究で扱った題材の特徴以外に，収集対象を拡大することやパッチ適用前後の各種メトリクスの変化も考慮することを検討している．

文　献

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, no.1, pp.4–12, 2002.

[2] T. Britton, L. Jeng, G. Carver, and P. Cheak, “Quantify the time and cost saved using reversible debuggers,” Technical report, Cambridge Judge Business School, 2012.

[3] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” IEEE Transactions on Software Engineering, vol.45, no.1, pp.34–67, 2019.

[4] M. Monperrus, “Automatic software repair: A bibliography,” ACM Computing Surveys, vol.51, no.1, pp.1–24, 2018.

[5] E.K. Smith, E.T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” Proc. Foundations of Software Engineering, pp.532–543, 2015.

[6] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” Proc. International Symposium on Software Testing and Analysis, pp.24–36, 2015.

[7] X.-B.D. Le, F. Thung, D. Lo, and C.L. Goues, “Overfitting in semantics-based automated program repair,” Empirical Software Engineering, vol.23, pp.3007–3033, 2018.

[8] C. Pacheco and M.D. Ernst, “Randoop: Feedback-directed random testing for java,” Proc. Companion to the Conference on Object-Oriented Programming Systems and Applications Companion, pp.815–816, 2007.

[9] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” Proc. Symposium and the European Conference on Foundations of Software Engineering, pp.416–419, 2011.

[10] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, “Quality of automated program repair on real-world defects,” IEEE Transactions on Software Engineering, 2020.

[11] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, “Automated patch correctness assessment: How far are we?,” Proc. International Conference on Automated Software Engineering, pp.968–980, 2020.

[12] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair,” Proc. International Conference on Software Engineering, pp.789–799, 2018.

[13] H. Ye, M. Martinez, and M. Monperrus, “Automated patch assessment for program repair at scale,” Empirical Software Engineering, vol.26, no.2, 2021.

[14] Q. Xin and S.P. Reiss, “Identifying test-suite-overfitted patches through test case generation,” Proc. International Symposium on Software Testing and Analysis, pp.226–236, 2017.

[15] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, “Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts,” Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.302–313, 2019.

[16] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” Proc. International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[17] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” Proc. International Symposium on Foundations of Software Engineering, pp.643–653, 2014.

[18] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. deAlmeida Maia, “Dissection of a bug dataset: Anatomy of 395 patches from defects4j,” Proc. the International Conference on Software Analysis, Evolution and Reengineering, pp.130–140, 2018.

表 2　分離の正誤と題材の特徴

		カバレッジ	生成パッチ数	修正の特徴		メソッドのメトリクス		
				修正行数	修正箇所	行数	循環的複雑度	NPath
分離に成功	平均値	0.816	118.4	4.66	2.46	36.54	12.52	4820203.3
	中央値	0.867	10.5	3	2	18	6	16
分離に失敗	平均値	0.728	75.7	8	2.4	25.75	8.2	13132.3
	中央値	0.798	5	7	2	12	4	4