修士学位論文

題目

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の分類
―パッチ確認コスト削減を目的として―

指導教員

楠本 真二 教授

報告者

藤本 章良

令和 4 年 2 月 2 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

令和 3 年度 修士学位論文

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の分類
—パッチ確認コスト削減を目的として—

藤本 章良

内容梗概

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を出力する場合が多く，パッチの適用にあたっては利用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，この実行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 83% 削減した．また，正誤の異なるパッチの別グループへの分類可能性について調査したところ，調査対象とした 41 個の題材のうち，28 個で分離に成功した．加えて分類に成功した題材の特徴，自動テスト生成回数と分類精度の関係性についても調査を行なった．

主な用語

自動プログラム修正，パッチ評価，自動テスト生成

目次

図目次

表目次

1 はじめに

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]．APR が行う修正とは，一部の仕様を満たしていないプログラムに対して，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ことである．例えばテストベースの 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．

しかしこれらの 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，バグの修正が不完全，あるいは新たなバグが混入した状態のパッチを出力する場合が多い [5, 6, 7, 8, 9]．こうしたパッチをオーバーフィットなパッチと呼ぶ [5, 6, 10]．テストベースの APR の場合においては，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おらず別のテストスイートには失敗してしまう状態である [5]．オーバーフィットなパッチの存在は APR が抱える大きな問題のひとつである [5, 10, 11]．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, 8, 9]，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 [12] が調査によって明らかになっている．

そのため APR ツールの利用者は出力されたパッチをプログラムに適用する前に，オーバーフィットなパッチを避けるため，それらが真に正しいかを個別に目視で確認する必要がある．しかしながら，パッチの確認はコストの大きい作業である [13]．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．前者に関して，オーバーフィットなパッチを生成しづらい APR の研究も行われている [14, 15, 16, 17]．しかし，オーバーフィットなパッチの完全な除去は達成できておらず，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といえる．

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る．

よって同じグループではパッチを個別に確認せず正誤判定ができ，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振る舞いによる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筆者らは考えた．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．

本研究では，Defects4J ベンチマーク [18] 内の 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，評価実験を行った．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 83%，中央値で 87% 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．さらに，正誤の異なるパッチの分離に与える影響，および自動テストの生成回数とパッチの分離精度の関係についても調査した．

以降，2 節では研究動機として，APR の出力パッチ確認コスト削減に向けたアイデアを実例を交えて説明し，3 節では提案手法について説明する．4 節では，本研究で設定した Research Question についての説明を行い，5 節でその Research Question に答えるために行った実験の内容と結果および考察について述べる．6 節では本研究の妥当性の脅威について述べる．7 節で本研究の関連研究を，最後に 8 節で，本研究のまとめと今後の課題について述べる．

2 研究動機

APR ツールの利用において，各パッチのオーバーフィットの有無を利用者自身で確認することが必須となっている．パッチの全てを個別に目視で確認するのは効率が悪く，多大なコストを要する．そこで，APR ツールが出力するパッチは類似している場合が多く，振る舞いが同じパッチも多々存在するという特徴を利用する．APR ツールが出力するパッチの中で，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 APR の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と筆者は考えた．APR ツールの利用者は同じ振る舞いを持つパッチのグループをひとつ取り出す．そのグループに属するパッチのひとつについてオーバーフィットか否かを確認すれば，そのグループの他のパッチも同様にオーバーフィットかどうかがわかるため，他のパッチは確認する必要がなくなる．すなわち，確認すべきパッチの数を，APR ツールが出力したパッチ数から同じ振る舞いを持つパッチのグループ数に削減可能である．大量のパッチが出力された場合においても，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できる．

以下の図 1 は，APR ツールが同じ振る舞いを持つパッチを複数出力した実際の例である．先行研究 [19] において，Defects4J ベンチマーク [18] 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，開発者によって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，メソッド `add` における引数 `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 `NullPointerException` が発生する．同様に `null` を与えると例外 `NullPointerException` が発生するパッチが複数確認された．これらのパッチは “メソッド `add` の引数に `null` を与えると例外 `NullPointerException` 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 “メソッド `add` 引数に `null` を与えると例外 `NullPointerException` 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，そのグループ全体はオーバーフィットなパッチであることがわかる．よって，そのグループに含まれる他のパッチは確認せずに済み，コストの削減につながる．

```
 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 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．

提案手法では，振る舞いが同じパッチをグループ化するために，自動テスト生成技術を利用する．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．すなわち，パッチ適用後のプログラム（以降では，パッチ適用後のプログラムを単にパッチと呼ぶ）から生成されたテストスイートは，そのパッチの振る舞いを示している．したがって，あるパッチ p_k が別のパッチ p_l から生成されたテストスイートに通過する場合は，p_k は p_l の仕様を満たす振る舞いをするパッチであるといえる．よって p_k，p_l がそれぞれから生成したテストスイートを相互に通過すれば，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 または p_l のどちらか一方が生成したテストスイートに失敗した場合，p_k と p_l は異なる振る舞いを持つパッチである．

提案手法はこのアイデアに基づき，全てのパッチから生成したテストスイートを相互に実行し，振る舞いが同じパッチのグループを得る．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．

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．

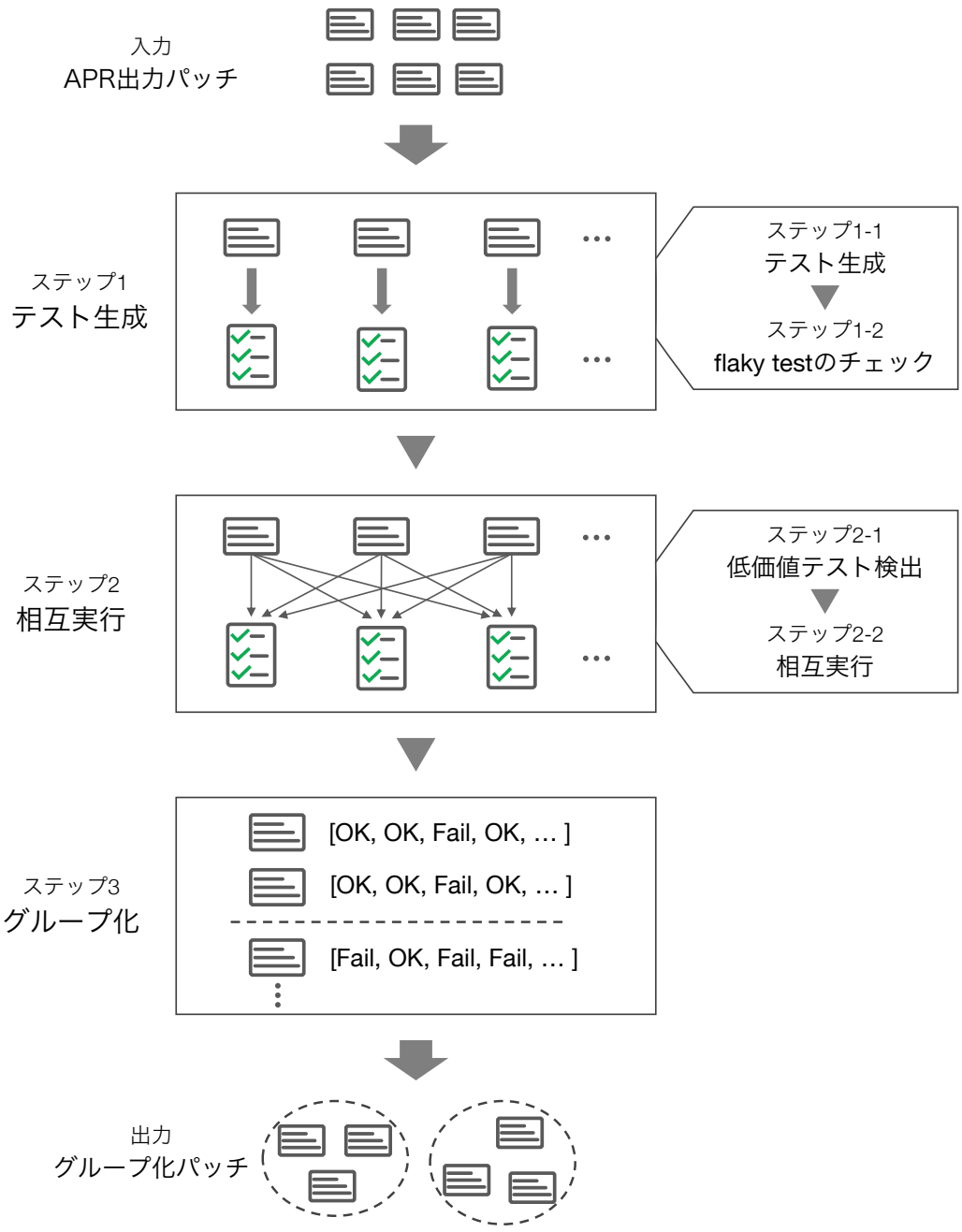

入力
APR出力パッチ
ステップ1
テスト生成
ステップ1-1
テスト生成
ステップ1-2
flaky testのチェック
ステップ2
相互実行
ステップ2-1
低価値テスト検出
ステップ2-2
相互実行
ステップ3
グループ化
[OK, OK, Fail, OK, …]
[OK, OK, Fail, OK, …]
[Fail, OK, Fail, Fail, …]
出力
グループ化パッチ

図 2　提案手法の概要

3.1 ステップ 1：各パッチからテストスイート生成

ステップ 1-1：テストスイート生成

入力として与えられた APR が出力したパッチ $p_1, ..., p_n$ 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．ここでは，APR が修正したクラス全てを対象にテストを生成する．例えば，パッチ 1 では `ClassA`，パッチ 2 で `ClassA`，`ClassB` を修正したとする．この場合では，パッチ 1，2 の両方で `ClassA`，`ClassB` を対象としたテストを自動生成する．

ステップ 1-2：flaky test のチェック

次に，生成された各テストスイートが 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，同じ実行環境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 [20]，本手法の有効性が低下する可能性があるため，ステップ 2 で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 tc_{ij} 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．自動テスト生成技術は，入力のプログラムが通過するようなテストを出力する．したがって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．それにも関わらず失敗するテストケースは，実行のタイミングによって成否が変化する flaky test であると考えられる．

3.2 ステップ 2：生成テストとパッチの相互実行

ステップ 2-1：価値の低い自動生成テストの除去

自動テスト生成技術によって生成されるテストスイートの中には，パッチによる振る舞いの違いの発見に不十分，すなわち全てのパッチが通過するテストケースが存在する．その一例を図 3 に示す．これらは自動テスト生成ツールで実際に生成されたテストケースである．図 3(a) は，インスタンス化した直後にそのインスタンスのフィールドの値を確認するテストケースである．APR ツールによってフィールドの初期値やコンストラクタの書き換えが行われなければ，どのパッチも同じ実行結果となる．図 3(b) は，クラスのインスタンス化時にコンストラクタに `null` を渡すことによって例外 `NullPointerException` の発生を期待するテストケースである．この例でも同様に APR ツールがコンストラクタ内で引数が `null` だった場合に関する処理を変更していなければ，どのパッチも実行結果は同じになると考えられる．図 3(c) は，テストメソッド中に assert 文がひとつも存在しない例である．このテストケースは例外が発生しない限り，どのパッチも失敗することはない．

こうしたテストケースは全てのパッチが通過するため，グループ化に全く寄与しない，価値の低いテ

```
@Test(timeout=4000) public void test038_Arja_0_seed1() throws Throwable {
  OpenMapRealVector openMapRealVector0 = new OpenMapRealVector();
  int int0 = openMapRealVector0.getDimension();
  assertEquals(0, int0);
}
```

(a) 初期値を期待するテストケース

```
@Test(timeout=4000) public void test11_Arja_101_seed1() throws Throwable {
  DiscreteDistribution<String> discreteDistribution0 = null;
  try {
    discreteDistribution0 = new DiscreteDistribution<String>((List<Pair<String,Double>>)null);
    fail("Expecting exception: NullPointerException");
  }
  catch (NullPointerException e) {
    verifyException("org.apache.commons.math3.distribution.DiscreteDistribution", e);
  }
}
```

(b) `NullPointerException` を期待するテストケース

```
@Test(timeout=4000) public void test19_Arja_0_seed1() throws Throwable {
  HypergeometricDistribution hypergeometricDistribution0 =
    new HypergeometricDistribution(2938,1264,1264);
  hypergeometricDistribution0.cumulativeProbability(975);
}
```

(c) assert 文が存在しないテストケース

図 3　グループ化に寄与しないテスト

ストケースである．しかし，こうしたテストケースの実行にも当然時間を要する．相互実行においては，パッチの数だけテストが実行されるため，価値の低いテストケースが多く存在する場合，その実行に必要以上に長い時間を要する．そこで，こうしたテストケースを予め除去することにより，相互実行時間の削減を図る．

このステップで行う処理の概要を図 4 に示す．価値の低いテストケースを特定するために，バグを含んだ状態のプログラムを利用する．バグを含むプログラムを用いて全てのテストケースを実行し，成功したテストケースを価値の低いテストケースとする．これは，バグを含むプログラムでも通過するテストケースは，パッチの特徴をうまく表現できていないと捉えられるからである．バグを含むプログラムとパッチ適用済みのプログラムは，APR の入力として与えたテストスイートの成否が異なるという明確な振る舞いの違いを持つ．しかし，自動生成したテストケースにおいてどちらのプログラムも通過するということはその振る舞いの差を表現できておらず，他のパッチも同様に通過する価値の低いテストケースである可能性が高い．

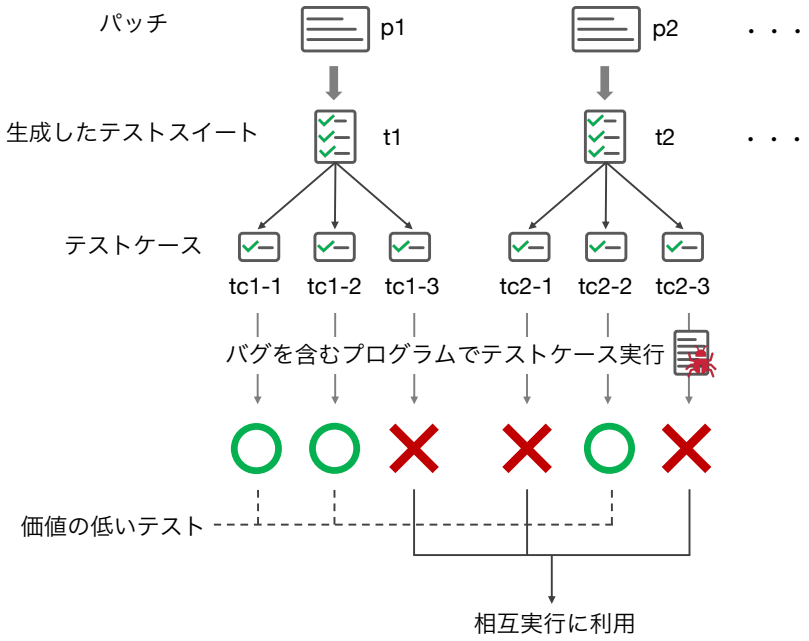


図 4　価値の低いテストの除去

ステップ **2-2**：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 5 上のように $n \times n$ の表として，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．また “Fail” 下段の括弧は失敗したテストクラスとテストケースを示す．例えば，`A-tc1` という表記はクラス A から生成したテストの tc_1 に失敗したことを表す．

3.3　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．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．これは相互実行結果の各行に対応する．ここで，p_k，p_l のテスト実行結果の一致を，任意の t_i に注目した時 p_k と p_l の失敗するテストケースが一致する場合と定義する．p_k と p_l のテスト実行結果が一致する場合，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．同じ振る舞いを持つパッチであれば，同じテストの実行結果が得られるためである．なお，p_k と p_l が互いのテストに通過することは直接確認していない．p_k と p_l のテスト実行結果が一致するが，p_k と p_l は互いのテストに通過しない可能性も存在する．しかし，ステップ 1-2 で flaky test の除去を行い p_k が t_k に通過することを保証している．よって，p_k と p_l のテスト実行結果が一致すれば 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．

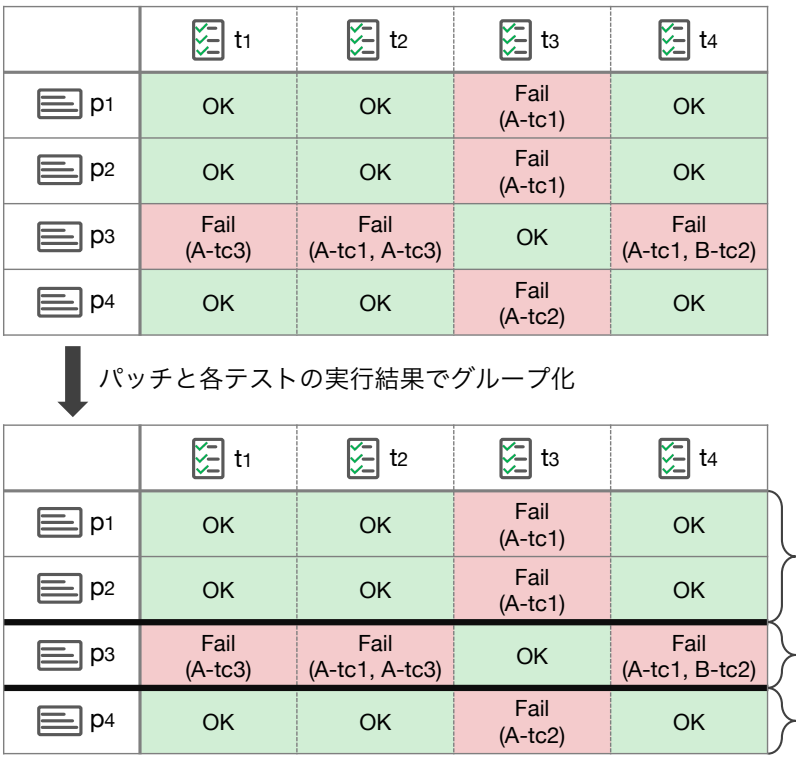


図 5　パッチの相互実行とグループ化

最終的にテスト実行結果が一致するパッチの集合を全て探し，グループ化したパッチとして結果を出力する．図 5 の例では p_1 と p_2 のテスト実行結果が一致し，これらは同じグループに分類される．一方 p_3 と p_4 のテスト実行結果はそれぞれ異なっている．p_4 は，各 t_i の成否は p_1，p_2 と同じであるが，t_3 での失敗したテストケースが異なるため，別のグループに分類される．よって，出力されるグループは $\{p_1, p_2\}$，$\{p_3\}$，$\{p_4\}$ の 3 つである．

4 Research Questions

提案手法の評価を行うにあたり，以下 4 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: 提案手法は正解パッチと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類できるか？（正誤の異なるパッチの分離は可能か？）

APR 利用者が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．仮にひとつのグループに正解パッチとオーバーフィットなパッチが混在した場合，選択したパッチによりグループ全体の正誤判定が変化するからである．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離の成否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，正誤の異なるパッチ分離の成否との関連について分析する．

RQ4: テストの生成回数は正誤の異なるパッチの分離可能性およびカバレッジに影響を与えるか？

乱択アルゴリズムを用いた自動テスト生成技術を用いる場合は，ランダム性を排除するためにテストの生成に 30 回の試行が必要との報告 [21] がある．一方で，提案手法はテスト生成回数に比例した実行時間を要する．そこでテスト生成の回数ごとに，正誤の異なるパッチ分離の成否およびカバレッジを測定し，結果がある程度収束する生成回数を調査する．特に提案手法においてはパッチの数だけテストの生成を行うため，生成するテストの総計は多くなる．そのため，早い段階で結果がある程度収束する可能性もある．

5 評価実験

5.1 実験対象

実験対象は Defects4J ベンチマーク [18] の Math プロジェクトに含まれるバグに APR を適用して得られたパッチである．APR ツールが出力したパッチとして，Durieux らの先行研究 [19] の過程で生成したパッチのデータセット[*1]を利用した．このデータセットには，11 種類の APR ツール [22, 23, 24, 25, 26] 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 の 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，正解パッチとして開発者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 [9]．本実験の対象である 41 個の題材のうち，APR がオーバーフィットなパッチのみ生成した題材は 33 個だった．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 [27] のデータセット[*2]を利用した．

題材ごとのパッチ数の分布を図 6 の箱ヒゲ図に示す．ただし，実際に開発者が行った修正パッチも含まれている．題材ごとのパッチ数は，2〜620 個と様々な大きさの題材が存在し，全ての実験対象から得たパッチ数の合計は 4,230 個だった．

5.2 実験設定

提案手法では自動テスト生成ツールとして，EvoSuite [28] を利用した．実験で用いた EvoSuite のバージョンは 1.1.0 である．EvoSuite を用いた理由は，先行研究 [29] において，Evosuite が生成したテストはオーバーフィットなパッチを発見する能力が Randoop [30] よりも高いと報告されていたためである．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．

5.3 テストの自動生成と **flaky test** および価値の低いテストの除去

本実験で生成した自動生成テストの数を図 7 に示す．ステップ 1-1 で生成したテスト数，ステップ 1-2 で検出した flaky test の数，ステップ 2-1 で価値の低いテストを取り除いた後の最終的なテスト数

[*1] https://github.com/program-repair/RepairThemAll_experiment
[*2] https://github.com/program-repair/defects4j-dissection

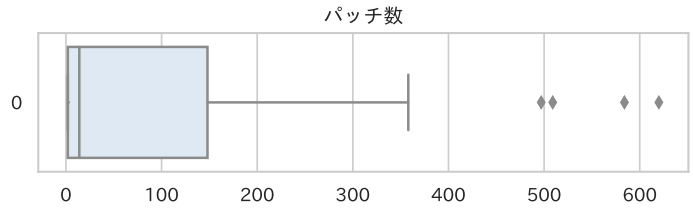


図 6　題材ごとのパッチ数

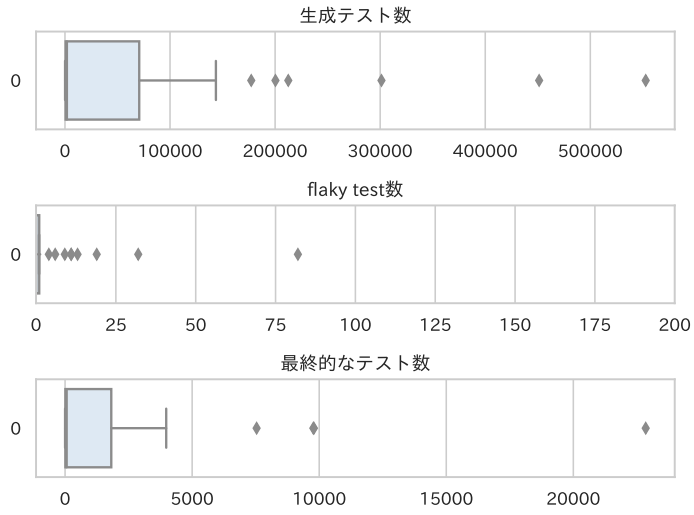


図 7　自動生成テスト数と flaky test 数，および最終的なテスト数

の分布を示している．この図では，題材ごとに生成したテスト数に関するデータを箱ヒゲ図で示している．ステップ 1-1 では合計で 2,610,554 個ものテストの生成を行なった．生成したテストのうち，flaky test は 1,424 個含まれていた．ステップ 2-1 で，価値のないテストを除去した後残ったテストの数は，75,157 個だった．これは生成したテストの 2.9% にあたる．ステップ 2-1 により実行すべきテストが大幅に減少したことがわかる．

5.4 RQ1: 確認すべきパッチ数はどの程度減少するか？

パッチの削減率という指標を次の式で定義する．

$$\text{削減率} = \left(1 - \frac{\text{提案手法が出力するグループ数}}{\text{生成パッチ数}}\right) \times 100$$

これは，確認すべきパッチ数をどの程度削減できるかを示す値である．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．各題材について，削減率を求める．ただしこの 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 8 に示す．削減率は平均で 83%，中央値 87% 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．また，最低でも Math 73 において 61.7% の削減率を達成しており，バグの種類に問わず確認すべきパッチの数を削減できることが読み取れる．

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 9 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，いくつか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修正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変更が行われやすい．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．パッチによっては共通の修正箇所以外も変更しているが，それは必ずしも振る舞いに影響を与えないことが多い．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 9 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 82%，中央値で 87% 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．

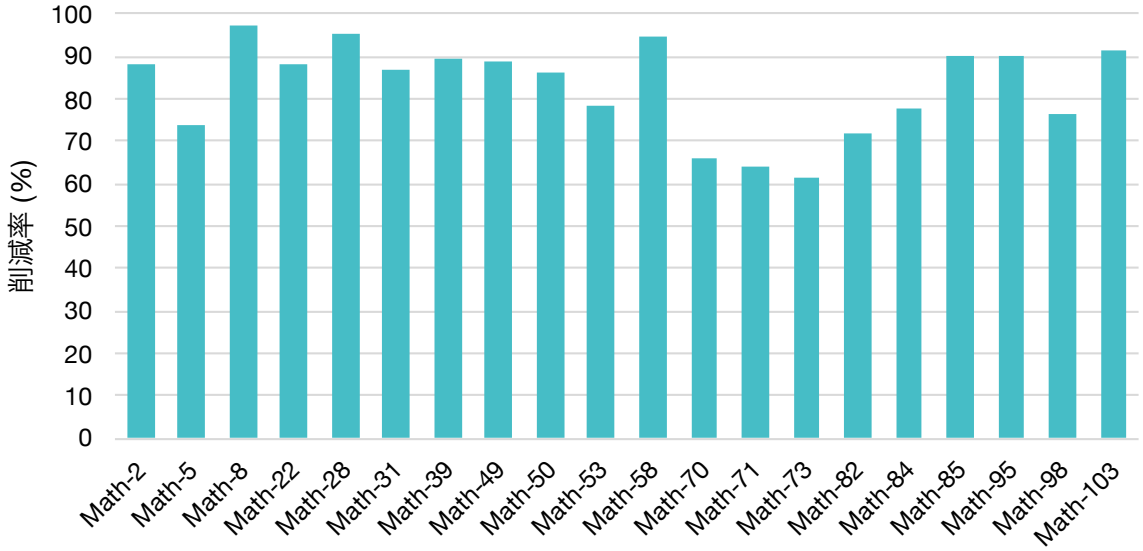


図 8　題材ごとの確認すべきパッチの削減率

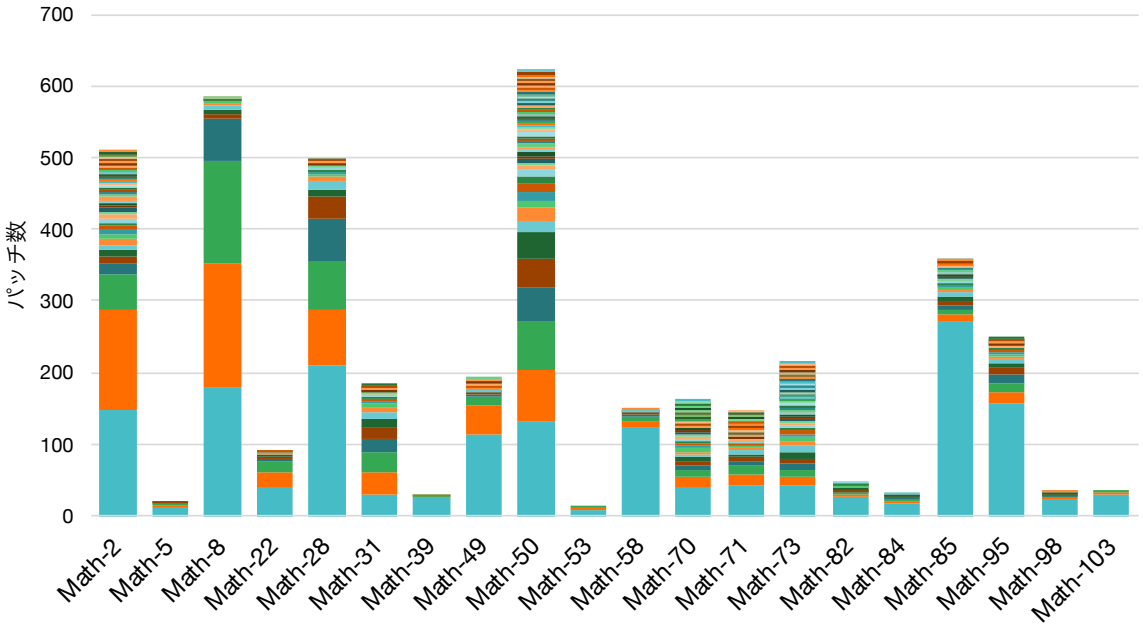


図 9　グループの分割状況

5.5 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．合計で 2,184 個のパッチを確認した．

目視確認の結果を表 1 に示す．また，図 10 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 28 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．

失敗した題材では，有効なテストケースが生成できなかったケースが多く見られた．これらの題材では価値の低いテストが多く生成され，振る舞いの差を発見できるようなテストケースがあまり生成されなかった．これらのテストが生成された要因として，想定された利用方法に則る必要があることが挙げられる．Math プロジェクトは主に数値計算を行うためのプロジェクトであるが，計算結果を取得する

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	21
	(b)APR 正解あり	7
分離に失敗	(c)APR 正解なし	12
	(d)APR 正解あり	1
	(e) 正解が複数に分割	0
合計		41

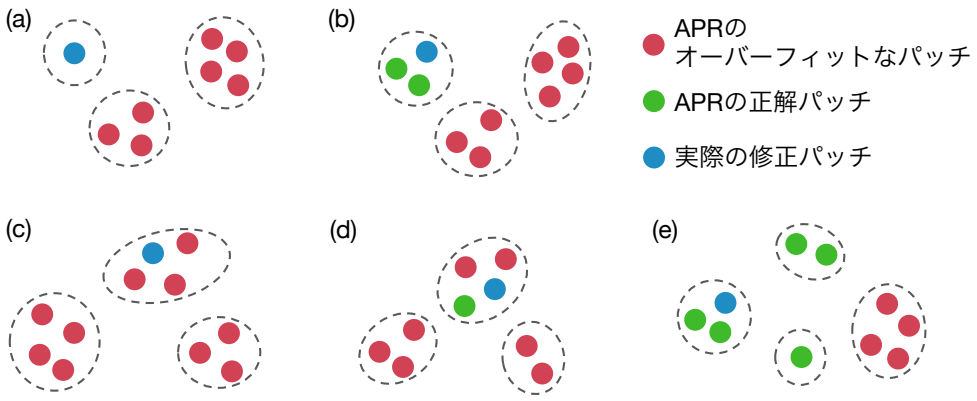


図 10　表 1(a)〜(e) と分割結果の対応

ために特定のメソッドを呼び出す必要がある．例として，計算結果を格納したインスタンスを返り値として受け取った場合，値を取り出すために適切な getter メソッド等を呼ぶ必要がある．しかし，生成されたテストは場合によっては計算結果とは関係ないメソッドを用いて assert するため，パッチ間で結果が常に同じとなる．例えば，入力として与えた配列の長さを得る getter を用いて assert した場合，どのパッチも得られる値が同じとなる．そうしたテストが多く生成された結果，パッチの分離に失敗したと考察される．

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) は存在しなかった．この結果から，異なる振る舞いを持つパッチがひとつのグループとして分割されることはあっても，同じ振る舞いを持つパッチであれば複数のグループに分割されることはないといえる．

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 41 個中 28 個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．また，失敗する題材では有効なテストケースの生成ができなかったケースが多く見られた．

5.6 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，各題材における開発者が実際に行った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・実行可能経路数）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，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 11 は各題材ごとの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 5.5 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，正誤の異なる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，グループの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，開発者によって実際に行われた修正パッチから得た値である．分離に成功する題材は修正行数が多いが，失敗する題材との差は平均値，中央値ともに 2 行未満である．また，修正箇所に関しては分離に成功する題材と失敗する題材の間で差はほとんどなく，中央値は同じ 2 である．行うべき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．こちらは分離に成功する題材の方が，行数，循環的複雑度，実行可能経路数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

表 2　分離の正誤と題材の特徴

		分離に成功		分離に失敗	
		平均値	中央値	平均値	中央値
カバレッジ		0.79	0.84	0.75	0.77
生成パッチ数		122.8	29	68.8	4
修正の特徴	修正行数	6.5	4	4.9	3
	修正箇所	2.6	2	2.2	2
メソッドのメトリクス	行数	37.4	15	17.3	12
	循環的複雑度	12.4	6	6.6	4
	実行可能経路数	4113286.1	16	146165.9	9

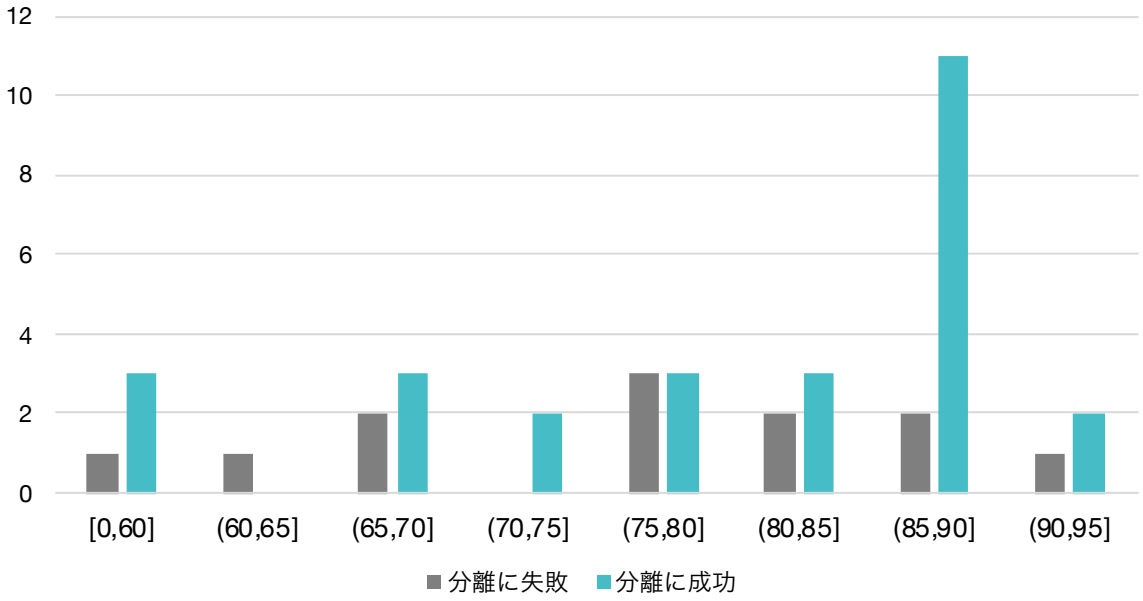
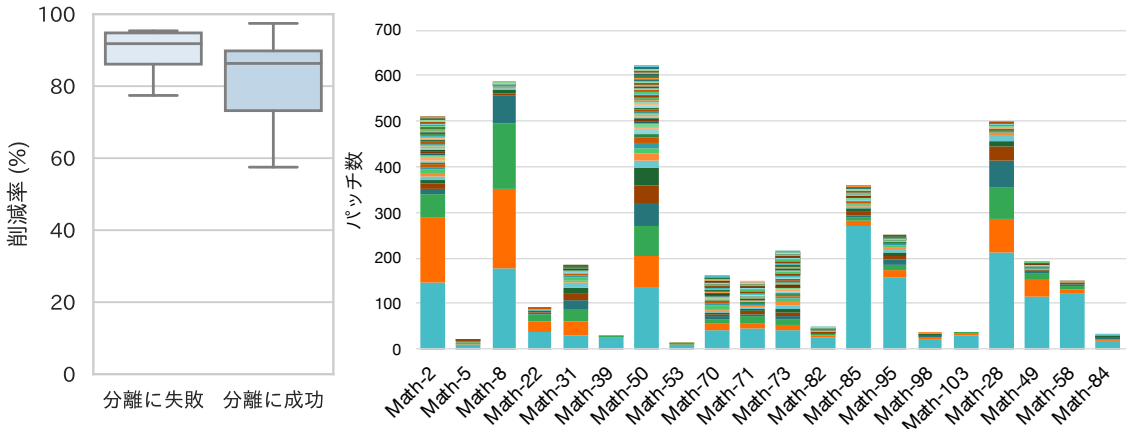


図 11　分離の成否と EvoSuite のテストカバレッジの分布



(a) 削減率と正誤の異なるパッチ分離の成否

(b) グループの分割状況と正誤の異なるパッチ分離の成否

図 12　RQ1 の調査項目と正誤の異なるパッチ分離の成否

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 12(a) は，正誤の異なるパッチの分離に成功または失敗した題材ごとに削減率を算出した結果である．分離に失敗した題材の方が，成功した題材よりも削減率が高い．分割グループ数が必要数より少なくなったために分離に失敗し，その分削減率が高くなったと考えられる．しかし，その差はあまり大きくはなく，最大値は分離に成功した題材の方が高い．マン・ホイットニーの U 検定を行ったところ，p 値は 0.156 であり，有意差は得られなかった．図 12(b) 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．分割状況からは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や APR により生成されたパッチ数，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．

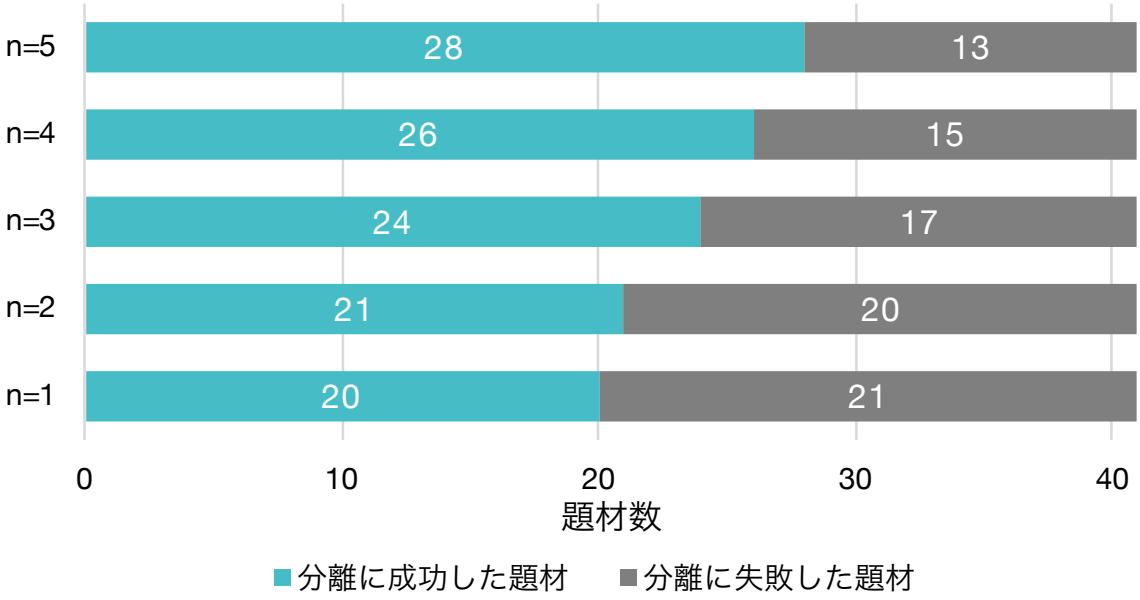


図 13　テスト生成回数ごとのパッチ分離の成否

5.7　RQ4: テストの生成回数と正誤の異なるパッチの分離可能性およびカバレッジ

テスト生成回数によって，正誤の異なるパッチの分離精度や生成したテストを実行したときのカバレッジがどのように変化するかを調査する．生成回数ごとに，分離に成功または失敗した題材の数，および各テスト生成対象クラスのカバレッジを測定する．本実験では，テストの生成回数を n とした時，EvoSuite の入力としてシード値 $1, ..., n$ を与え生成されたテストを相互実行に利用した．生成回数は $1 \leq n \leq 5$ である．

まず，各生成回数ごとに正誤の異なるパッチ分離の成否を集計した結果を図 13 に示す．テストの生成回数を増やしていくと，正誤の異なるパッチの分離に成功する題材が増えることが読み取れる．テストの生成回数を増やすことにより，パッチ間の振る舞いの差を見つけるのに有用なテストができたと考

表 3　テスト生成回数ごとのパッチ分離成功数の推移

生成回数 n	$n=2$	$n=3$	$n=4$	$n=5$
$k=1$ との成功数の差	+1	+4	+6	+8
$k=1$ との成功数の増加割合	5%	20%	30%	40%
テスト生成回数あたりの増加割合	2.5%	6.7%	7.5%	8%
$k=n-1$ 回との成功数の差	+1	+3	+2	+2
$k=n-1$ 回との成功数の増加割合	5%	14%	8%	8%

察できる．

表 3 では，各テスト生成回数 n について，テスト生成回数 k の場合との比較を行った．ここでは $k = 1$ の場合と，$k = n - 1$ の場合を比較する．その結果，$k = 1$ の時と比較して，テスト生成回数あたりの成功数の増加割合が最も高かったのは生成回数が 5 回の時であった．これは，テスト生成回数に対するパッチ分類の精度向上の効率が最も大きいことを意味する．テスト生成回数は実行時間と比例関係にあるため，言い換えれば実行時間あたりの効率が高いとも言える．また，$k = n - 1$ 回の時と比較した場合には，生成回数 3 回が最も増加割合が高くその後も成功数が常に増加している．これらからテスト数を増やすほど精度が向上し，テスト生成回数 5 回まではテスト生成に対する精度向上の効率も良いことがわかった．

次に，各生成回数ごとのカバレッジを図 14 に示す．図中の箱ヒゲ図において，値はあるテスト生成対象クラスについて，そのクラスから生成したテストの実行時におけるカバレッジを示している．例えば，テスト生成対象クラス `classA` のカバレッジは，`classA` から生成したテストを実行した時の `classA` のカバレッジである．パッチごとに各クラスについてカバレッジの計測を行い，平均値または中央値でカバレッジの集計を行った．カバレッジの計測には OSS である JaCoCo[*3]を用いた

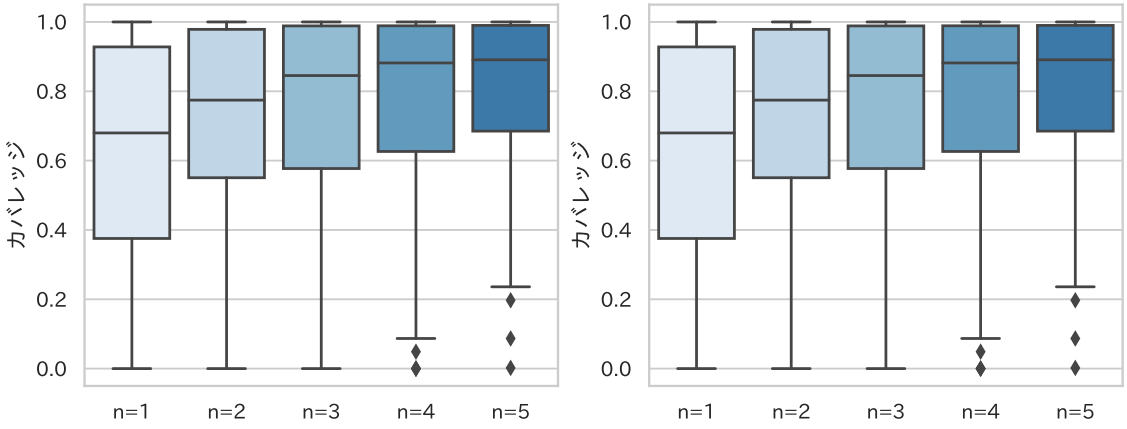
また，カバレッジが測定できないクラスは省略した．ステップ 2-1 において，あるクラスから生成したテストケースの全てに対してバグを含むプログラムが通過し，実行すべきテストケースの数が 0 となったクラスがこれに該当する．

この結果から，テストを生成するごとにカバレッジが増加することが読み取れる．C0 カバレッジに注目すると，テスト生成回数 3 回までの増加率が高い．また，生成回数 4〜5 回は中央値は大きく変化しない一方で第一四分位数は大きく増加している．生成回数を増やすと，カバレッジが比較的低いクラスについてカバレッジが向上することが読み取れる．

C1 カバレッジも C0 カバレッジと同様の傾向が見られた．ただし，C1 カバレッジに関してはテストの生成回数を増やしても第一四分位数に変化はなかった．

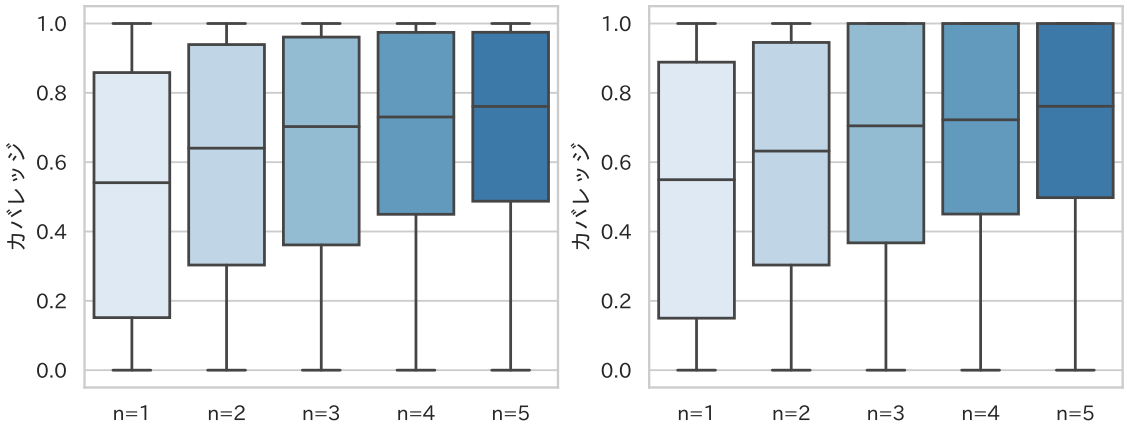
> **RQ4** への回答：テスト生成回数の増加によって，正誤の異なるパッチ分離の精度およびカバレッジが向上した．精度についてはテスト数を増やすほど向上が見られた．カバレッジは，生成回数 3 回目までの向上が著しい．

[*3] https://www.jacoco.org/jacoco/



(a) C0 カバレッジ（平均値）

(b) C0 カバレッジ（中央値）



(c) C1 カバレッジ（平均値）

(d) C1 カバレッジ（中央値）

図 14　テスト生成回数とカバレッジの変化

6　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．そのため筆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 ベンチマークを用いたが，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．

7 関連研究

7.1 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いる．例えば，ランダムな入力からテストを生成する Randoop [30] や T3 [31]，遺伝的アルゴリズムを用いる EvoSuite [28]，記号実行を用いる SUSHI [32, 33] など様々な手法が提案されている．本研究においては，ステップ 1-1 で使用する自動テスト生成技術は，これらの手法を入れ替えたり組み合わせたりすることも可能である．

7.2 自動プログラム修正（APR）

Monperrus の調査によると 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，この Behavioral repair を想定して研究を行なった．

Behavioral repair の中でも特にテストスイートを仕様として利用する手法が多く提案されている．この手法は大きく 2 つに分けられる．1 つ目の手法は，生成と検証に基づく方法である．生成と検証に基づく手法は，プログラムの一部を改変し，テストを全て通過可能か検証を行う．これを繰り返すことで全てのテストを通過する，すなわち修正後のプログラムを得る．GenProg [34] は遺伝的プログラミングを用いて，修正を行う．これらは，プログラム文の挿入や削除を繰り返し，プログラムを徐々にバグのない状態へと近づける．GenProg 以外にも遺伝的アルゴリズムを取り入れた手法が存在する [22, 23, 35]．変更パターンを用いる手法も存在する．PAR [36] や TBar [37] は頻出するバグ修正パターンをあらかじめ収集し，バグのあるプログラムに適用する手法である．

2 つ目は意味論に基づく手法である．これは与えたテストスイートから満たすべき仕様を特定し，プログラムを合成する手法である．Nopol [38] や DynaMoth [25] といった手法がある．

7.3 オーバーフィットなパッチ

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 [5, 10, 11]．そのためオーバーフィットなパッチ出力の有無も，APR の性能評価において重要な要素である．オーバーフィットなパッチの調査を含め，APR が出力したパッチを調査する研究がこれまでに多く行われている．Qi ら [6] や Martinez ら [8] は，生成と検証に基づく APR ツールが生成したパッチについて調査し，Le ら [7] は意味論に基づく APR ツールが生成したパッチについて調査した．Liu ら [9] は様々な手法

を用いる 16 種の APR を用いて調査を行なっている．これらの調査から，APR はその手法を問わずオーバーフィットなパッチを出力し，それらは全出力パッチの大部分を占めることが明らかになった．Motwani らは，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と結論付けた [12]．Zhongxing らはオーバーフィットの種類を，バグを修正しきれていないていない状態，新たなバグを混入した状態，両者の混合状態に分類した [39]．Nopol [38] を用いて評価した結果，どのパターンのオーバーフィットも存在することが確認された．

これらの研究結果から，オーバーフィットなパッチを生成しづらくする工夫を取り入れた APR 手法が提案されている．バグ修正におけるアンチパターンの導入 [14]，頻出する修正のコンテキストの利用 [15]，経験則に基づくルールを利用する手法 [16, 17] が存在する．

テストベースの APR において，入力として与えるテストスイートとオーバーフィットなパッチの関係に関する調査も行われている．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]．テストケースの数やテストスイートのカバレッジが不十分な場合，オーバーフィットなパッチを生成しやすい [5, 12, 40, 41]．しかしカバレッジが高ければ，わずかにオーバーフィットなパッチが生成されやすくなるという報告 [12] もあり，オーバーフィットなパッチの根本的な解決には至っていない．

7.4 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自動で評価する研究が広く行われている [11]．一方で，本研究のようにパッチをグループ化するという手法は，筆者の知る限りでは存在していない．個々のパッチ評価に関する手法は，評価時の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 [10]．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．これは，正しいパッチは成功するテストにおいてバグを含む状態のプログラムと似たような実行経路を持ち，失敗するテストでは異なる実行経路を持つという仮説を持つ．Opad [42] は fuzz testing [43] とメモリ監視ツール[*4]を用いる．オーバーフィットなパッチによって新たに発生するバグやメモリに関する問題を発見する．また，パッチを静的解析して得られる特徴や行われた修正の種類をもとに，オーバーフィットの有無を判定する手法も提案されている [16, 17]．

機械学習を用いてパッチのオーバーフィットの有無を判定する研究も行われている．Ye らは，AST およびその差分を用いて，ソースコードに含まれるプログラム要素と AST の変更操作，および修正

[*4] https://valgrind.org/

のパターンをパッチの特徴量を抽出し，パッチのオーバーフィットの有無を判定する手法を提案した [44]．Haoye らは複数の予測モデルを用いて比較実験を行い，BERT [45] が最も優れていたと結論づけた [46]．Csuvik らは，予測モデルに与えるパッチの表現方法による違いが予測精度に与える影響について調査している [47]．ただし，機械学習を用いる手法は事前に用意する学習用データとして，正解またはオーバーフィットのラベル付が行われたパッチが必要である．

正解パッチを利用する手法には，DiffTGen [48] がある．DiffTGen 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．そのテストケースに対して正解パッチと APR 生成パッチの両方を実行し，それらの実行結果が異なれば，APR 生成パッチをオーバーフィットであると判断する．その他にも RGT（Random testing in Ground Truth） [49] 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，Defects4J ベンチマーク [18] 上のバグに対し APR が出力したパッチを，RGT を用いて分類した [29]．さらに Ye らは，QuixBugs ベンチマーク [50] にも同様の実験を行い RGT の有効性を評価している [51]．

正解パッチを用いない手法は，バグに APR を適用して得られた未知のパッチの評価に有効であり，APR 利用者のパッチ確認コストを削減可能である．一方，正解パッチを用いる手法は正解パッチを利用する都合上，APR でバグを修正しその出力パッチを評価する使い方はできない．しかし，APR の性能測定の自動化という観点において，より重要な役割を果たす [29]．APR が出力したパッチの自動評価により，オーバーフィットなパッチの生成率を自動で測定できる．

本提案手法は上記の両方の用途を可能としている．正解パッチが未知の場合，提案手法により得られたグループの中からパッチをひとつ選択し，そのパッチについて確認することにより確認すべきパッチの数が減少する．正解パッチが既知の場合，提案手法により得られたグループについて，正解パッチと同じグループのパッチは正しいパッチ，正解パッチと異なるグループのパッチはオーバーフィットであると自動的に判断できる．特に，提案手法は正解テストを含む全てのパッチからテストを生成しているため，後者は RGT を用いた手法のスーパーセットになっていると捉えることができる．

パッチ評価手法を評価する研究も行われている．Le らは 35 人の開発者を被験者とし，事前にパッチの評価を行った [52]．その評価結果を用いて，DiffTGen [48] および Randoop [30] の結果と比較し，自動評価手法の精度を調査した．Shangwen らはいくつかのパッチ評価技術を収集し，それらの効果について調査した [11]．その結果，プロジェクトや APR ツールの手法により，オーバーフィットなパッチを発見可能な手法が変化し，複数の評価手法を組み合わせることでオーバーフィットなパッチを発見しやすくなるとしている．また，静的特徴量を学習することにより，高い精度でパッチ評価が行えるとも述べている．

APR が出力したパッチそのものに関する研究も行われている．Wang らは APR 生成パッチと実際

の開発者が行った修正パッチの構文的，意味的な違いを分析している [53]．さらにバグの特性とオーバーフィットの有無を調査し，修正すべき行数が多いバグほどオーバーフィットしやすいことが判明した．Yang らは，正しいパッチとオーバーフィットしたパッチにあるそれぞれの固有の不変条件の違いを調査した [54]．調査の結果，正解パッチとオーバーフィットしたパッチの間には不変条件の違いが存在し，普遍条件を用いてパッチを自動的に分類できる可能性を示した．

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 83%，中央値で 87% 削減できた．また，28 個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．加えて，本研究では Defects4J Math プロジェクトに対してのみ実験を行ったが，それ以外のベンチマークにおいても実験を行い有効性を示すことも今後の課題である．

謝辞

本研究を行うにあたり，丁寧かつ理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝申し上げます．研究におけるご指導以外にも様々な面で大変お世話になりました．誠にありがとうございました．

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝申し上げます．研究内容に関する相談から論文執筆の指導まで多大なご尽力を賜り，誠にありがとうございました．ご指導頂きました内容は，今後の人生の糧になると確信しております．

本研究に関して，研究の方向性や改善すべき点など有益かつ的確なご助言を頂きました，柗本真佑助教に深く感謝申し上げます．また，研究発表についても様々なご指摘を頂き大変勉強になりました．

研究活動を円滑に進めるにあたり，様々な支援および励ましの言葉を頂きました事務補佐員の橋本美砂子氏，前事務補佐員の神谷智子氏に深く感謝申し上げます．大変お世話になりました．

研究に関する様々なご助言を頂きました，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 2020 年卒業生の田中紘都氏，同土居真之氏，同中島望氏，同松本淳之介氏，2021 年卒業生の東英明氏，同九間哲士氏，同富田裕也氏，同中川将氏，同華山魁生氏に深く感謝申し上げます．頂いたご助言のみならず，皆様の研究への取り組み方も大変勉強になりました．

研究に関する相談のみならず日常生活の様々な面で支えて頂いた，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 年の市川直人氏，同出田涼子氏，同荻野翔氏，同前島葵氏に心から感謝申し上げます．皆様のおかげで，修士の研究生活においての様々な困難を乗り越えることができ，非常に有意義な研究生活を送ることができました．

研究生活を豊かにしていただきました，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 年の入山優氏，同古藤寛大氏，同髙市陸氏，同高木一真氏，同谷口真幸氏，同橋本周氏，同渡辺大登氏，大阪大学基礎工学部情報科学研究科 4 年の伊賀彰氏，同岩瀬匠氏，同小田郁弥氏，同竹重拓輝氏，同長谷川和輝氏，同吉岡遼氏に深く感謝申し上げます．快適な研究環境の維持にもご尽力頂きありがとうございました．

本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科，および大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

最後に，これまでの 24 年間様々な面で支えて頂き，励まして頂いた家族にも心より深く感謝いたします．

参考文献

[1] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[2] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Quantify the time and cost saved using reversible debuggers. Technical report, Cambridge Judge Business School, 2012.

[3] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67, 2019.

[4] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys*, Vol. 51, No. 1, pp. 1–24, 2018.

[5] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proc. Foundations of Software Engineering*, pp. 532–543, 2015.

[6] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proc. International Symposium on Software Testing and Analysis*, pp. 24–36, 2015.

[7] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, Vol. 23, pp. 3007–3033, 2018.

[8] 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*, Vol. 22, No. 4, pp. 1936–1964, 2017.

[9] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proc. International Conference on Software Engineering*, pp. 615–627, 2020.

[10] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying Patch Correctness in Test-Based Program Repair. In *Proc. International Conference on Software Engineering*, pp. 789–799, 2018.

[11] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. Automated Patch Correctness Assessment: How Far Are We? In *Proc. International Conference on Automated Software Engineering*, pp. 968–980, 2020.

[12] Manish Motwani, Mauricio Soto, Yuriy Brun, Rene Just, and Claire Le Goues. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering*, 2020.

[13] Zachary P. Fry, Bryan Landau, and Westley Weimer. A Human Study of Patch Maintainability. In *Proc. International Symposium on Software Testing and Analysis*, pp. 177–187, 2012.

[14] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-Patterns in Search-Based Program Repair. In *Proc. International Symposium on Foundations of Software Engineering*, pp. 727–738, 2016.

[15] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-Aware Patch Generation for Better Automated Program Repair. In *Proc. International Conference on Software Engineering*, pp. 1–11, 2018.

[16] Qi Xin and Steven P. Reiss. Leveraging Syntax-Related Code for Automated Program Repair. In *Proc. International Conference on Automated Software Engineering*, pp. 660–670, 2017.

[17] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proc. Foundations of Software Engineering*, pp. 593–604, 2017.

[18] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[19] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313, 2019.

[20] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An Empirical Analysis of Flaky Tests. In *Proc. International Symposium on Foundations of Software Engineering*, pp. 643–653, 2014.

[21] Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proc. International Conference on Software Engineering*, pp. 1–10, 2011.

[22] Matias Martinez and Martin Monperrus. ASTOR: A Program Repair Library for Java (Demo). In *Proc. International Symposium on Software Testing and Analysis*, pp. 441–444,

2016.

[23] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067, 2020.

[24] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In *Proc. International Conference on Software Analysis, Evolution and Reengineering*, pp. 349–358, 2017.

[25] Thomas Durieux and Martin Monperrus. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proc. International Workshop on Automation of Software Test*, pp. 85–91, 2016.

[26] Matias Martinez and Martin Monperrus. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Proc. International Symposium on Search-Based Software Engineering*, pp. 65–86, 2018.

[27] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proc. International Conference on Software Analysis, Evolution and Reengineering*, pp. 130–140, 2018.

[28] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proc. Symposium and the European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[29] He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, Vol. 26, No. 2, 2021.

[30] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *Proc. Companion to the Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 815–816, 2007.

[31] Wishnu Prasetya. T3, a Combinator-Based Random Testing Tool for Java: Benchmarking. In *Proc. International Workshop on Future Internet Testing*, pp. 101–110, 2013.

[32] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs. In *Proc. International Symposium on Software Testing and Analysis*, pp. 90–101, 2017.

[33] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. SUSHI: A Test Generator for Programs with Complex Structured Inputs. In *Proc. International Conference*

on Software Engineering: Companion Proceeedings, pp. 21–24, 2018.

[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[35] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kGenProg: A High-Performance, High-Extensibility and High-Portability APR System. In *Proc. Asia-Pacific Software Engineering Conference*, pp. 697–698, 2018.

[36] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proc.International Conference on Software Engineering*, pp. 802–811, 2013.

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting Template-Based Automated Program Repair. In *Proc. International Symposium on Software Testing and Analysis*, pp. 31–42, 2019.

[38] 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*, Vol. 43, No. 1, pp. 34–55, 2017.

[39] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering*, Vol. 24, No. 1, pp. 33–67, 2019.

[40] Fan Long and Martin Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proc. International Conference on Software Engineering*, pp. 702–713, 2016.

[41] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, Vol. 23, No. 5, pp. 2901–2947, 2018.

[42] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better Test Cases for Better Automated Program Repair. In *Proc. Foundations of Software Engineering*, pp. 831–841, 2017.

[43] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of

UNIX Utilities. *Communication of the ACM*, Vol. 33, No. 12, pp. 32–44, 1990.

[44] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *IEEE Transactions on Software Engineering*, p. 1, 2021.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. North American Chapter of the Association for Computational nguistics: Human Language Technologies*, Vol. 1, pp. 4171–4186, 2019.

[46] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *Proc. International Conference on Automated Software Engineering*, pp. 981–992, 2020.

[47] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. Utilizing Source Code Embeddings to Identify Correct Patches. In *Proc. International Workshop on Intelligent Bug Fixing*, pp. 18–25, 2020.

[48] Qi Xin and Steven P. Reiss. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *Proc. International Symposium on Software Testing and Analysis*, pp. 226–236, 2017.

[49] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proc. International Conference on Automated Software Engineering*, pp. 201–211, 2015.

[50] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proc. International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pp. 55–56, 2017.

[51] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. *Journal of Systems and Software*, Vol. 171, p. 110825, 2021.

[52] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. On Reliability of Patch Correctness Assessment. In *Proc. International Conference on Software Engineering*, pp. 524–535, 2019.

[53] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. How Different Is It Between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques. In *Proc. International Symposium on Empirical Software Engineering and Measurement*, pp. 1–12, 2019.

[54] Bo Yang and Jinqiu Yang. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *Proc. International Workshop on Intelligent Bug Fixing*, pp. 1–8, 2020.