テスト自動生成を用いたプログラム機能差の検出

- プログラミング教育における自動評価を目的として -

出田 涼子[1,a)] 柗本 真佑[1,b)] 井垣 宏[2] 佐伯 幸郎[3] 福安 直樹[4] 楠本 真二[1]

概要：プログラミング教育における一つの実施形態として，ソフトウェアテストを用いた方法が採用されつつある．この方法では，プログラムの仕様を単体，あるいは結合テストとして用意しておき，学生はそのテスト通過を目指してプログラムを作成する．客観的なプログラム仕様の確保，及び動作確認の自動化等様々な利点がある一方で，学生個々の独自仕様や機能拡張といった工夫の検出は難しい．さらには，与えられたテストには通過するが，その解決方法が十分に一般化されていないケースの検出も困難である．本研究の目的は，テストを用いたプログラミング教育における学生プログラム間の機能差の自動検出である．本稿では，この目的に対する 2 つの課題を整理し，その片方の実現方法を提案する．この提案手法では，学生のプログラムに対しテスト自動生成を適用し，生成テストを他の学生プログラムに対して相互に実行する．さらにテスト実行時の経路情報に基づき，テストをクラスタリングすることで，機能差検出能力を持つテストの集合を得る．本稿では，予備実験として実施した数行程度のプログラミング題材に対する提案手法適用の結果について報告し，残された 2 つ目の課題の実現方法について検討する．

1. はじめに

ソフトウェアテストを活用したプログラミング教育方法（以降，テストベース教育）が数多く提案されている [4], [6], [10]．この手法では，課題の仕様を単体あるいは結合テストとして定義しておき，学生はその事前テストの通過を目指してプログラムを作成する．これにより，客観的な課題仕様の確保，及び課題達成確認の自動化等，学生と教員の両者に対する様々な利点が得られる．また，テスト駆動開発 [2] や継続的インテグレーション [3] をはじめとするアジャイルな開発手法との親和性も高く，プログラミング能力の取得のみならず，実践的なソフトウェア開発の経験を得るという副次的な効果も期待できる．

しかしながら，テストベース教育では学生個々の機能拡張や工夫の検出が困難という課題が存在する．拡張や工夫は事前テストの範囲外の実装であり，学生ソースコード間における機能差の一種であると解釈できる．この機能差の典型例はパラメタの妥当性確認処理であり，オブジェクトの null 確認や，整数値の範囲確認等が挙げられる．さらには，テストでは未確認の独自仕様や機能拡張といった機能差も考えられる．これら学生の努力に該当する部分を検出し，採点への反映や適切なフィードバックを実施することで学習効果の向上が期待できる．

さらに，肯定的な機能差のみならず，事前テストへの過剰適合という否定的な機能差も存在すると考えられる．例えば，総和計算という課題に対して `assert(sum(3,4,5), 12)` というテストが定義されているとする．ここで `for n in [3..5]` のような加算対象の値を固定した実装や， `return 12` 等加算処理すら行わない実装は，事前テストには通過するものの，一般化が不十分な実装であり課題内容を達成しているとはいえない．工夫と同様に検出し，その実装が抱える問題を適切にフィードバックするべきである．

機能差の検出方法としては，テストの充足が一つの選択肢として考えられるが，存在しうる全ての振る舞いをテスト化することは根本的に不可能である [7]．また過度なテストは，ソフトウェア品質という側面では効果的ではある一方で，教育の場ではいい戦略とはいえない．課題仕様の本質が曖昧になるだけでなく，プログラミングにおける学生の自由な発想を阻害する要因となるためである．例えば，イースターエッグ[*1]のようなプログラミングの楽しさに寄与するユーモアの実装は，テストで無機質に排除するより

1 大阪大学
2 大阪工業大学
3 神戸大学
4 和歌山大学
a) r-izuta@osaka-u.ac.jp
b) shinsuke@ist.osaka-u.ac.jp

*1 ソフトウェア内に含まれた隠し機能の意味．例えば，標準入力を受け付ける課題に対して，特殊な文字入力時に特殊な振る舞いをする等．

© 1959 Information Processing Society of Japan

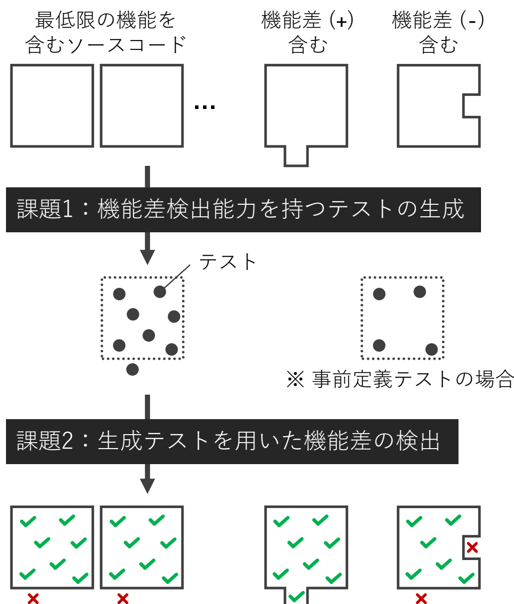

図 1: 機能差の自動検出における 2 つの課題

```java
public PortNumber {
  int number;
  public PortNumber(int n) {
    this.number = n;
  }
  public getNumber() {
    return number;
  }
  public toString() {
    if (number == 22) return "ssh";
    if (number == 80) return "http";
    ...
  }
}
```

(a) 模範解答

```java
@Test public void test1() {
  int number = new PortNumber(22).getNumber();
  assertEquals(22, number);
}

@Test public void test2() {
  int number = new PortNumber(80).getNumber();
  assertEquals(80, number);
}
```

(b) コンストラクタに対する事前テスト

図 2: PortNumber クラスの模範解答と事前テスト

も，何かしらの手法で検出し賞賛するべきであると著者らは考える．

本研究の最終的な目標は，テストベース教育における学生ソースコード間での機能差の自動検出である．この目標に対し，**図 1** に示す通り以下 2 つの課題を定める．

- 課題 1：機能差の検出能力を持つテストの生成と分類
- 課題 2：生成テストを用いた機能差の自動検出

課題 1 では学生の開発した複数のソースコード群から，事前テストのみでは実現できない機能差検出能力を持つテストを生成する．課題 2 では，得られたテストを各ソースコードに適用し，その機能差の有無を検出する．

本稿では課題 1 の達成を目標として，テスト自動生成，及びテスト実行時の経路情報を活用したテストのクラスタリング手法を提案する．この手法では，まず全ての提出ソースコードに対してテスト自動生成を適用する．これにより機能差の検出能力を持つテストを得る．さらに，生成テストを全ての学生間で相互に適用し，その実行経路の計測により各テストの動作確認手順を抽出する．最後に，各テストの経路をベクトル化しクラスタリングすることで，同じ動作確認手順を持つテストを洗い出す．また，本稿では予備実験として行った数行程度のプログラミング題材に対する適用結果について報告する．最後に課題 2 の実現方法について検討する．

2. Motivating Example

数行程度のプログラミング題材に基づいて，本研究の動機を説明する．題材は TCP/IP プロトコルにおけるポート番号を表す PortNumber クラスである．当該クラスに対する模範解答，及びコンストラクタに対する事前テストを**図 2** に示す．PortNumber クラスは，図 2 (a) に示すように，以下の要素から構成される．

- ポート番号を表す int 型フィールド `number`（2 行目）
- コンストラクタ（3〜5 行目）
- `number` に対する getter メソッド（6〜8 行目）
- `number` の文字列化メソッド（9〜13 行目）

ここで，本クラスの最初の実装課題として，図 2 (b) に示すような，コンストラクタに対する 2 つの事前テストが与えられたとする．この事前テストを通過する 3 種類のコンストラクタの例を**図 3** に示す．図 3 (a) の学生 A は模範解答と全く同じコンストラクタであり，事前テストを通過する最低限の機能のみを実装している．他方，図 3 (b) (c) の学生 B と C は，ポート番号の妥当性確認処理を実装しており，いずれも負の値を受け付けないコンストラクタである．異常系処理という工夫を含んでいると解釈できる．ただし，負の値指定時の異常系の処理内容は異なっており，B は代替値ゼロを代入し，C は実行時例外を発生させている．

B と C は，模範解答には含まれない肯定的な機能を含む実装であり，検出し適切な評価を与えるべきである．しかし，図 2 (b) の事前テストは最低限のコンストラクタ処理内容のみを確認しており，この工夫を検出する能力を持たない．目視による確認は一つの選択肢ではあるが，大学の演習をはじめとする提案手法の適用シーンを考えると受講生は 10〜100 人であり，現実的ではない．本節の例のような単純な課題であれば，キーワードベースやファイル比較，静的なソースコード解析技術を用いた検出も可能であるが，課題内容が複雑な場合その検出能力の低下が避けられない．

```java
1  public PortNumber(int n) {
2    this.number = n;
3  }
```

(a) 学生 A のコンストラクタ

```java
1  public PortNumber(int n) {
2    if (n < 0) {
3      this.number = 0;
4    } else {
5      this.number = n;
6    }
7  }
```

(b) 学生 B のコンストラクタ

```java
1  public PortNumber(int n) {
2    if (n < 0) {
3      throw new IllegalArgumentException();
4    }
5    this.number = n;
6  }
```

(c) 学生 C のコンストラクタ

図 3: 事前テストを通過するコンストラクタの実装例

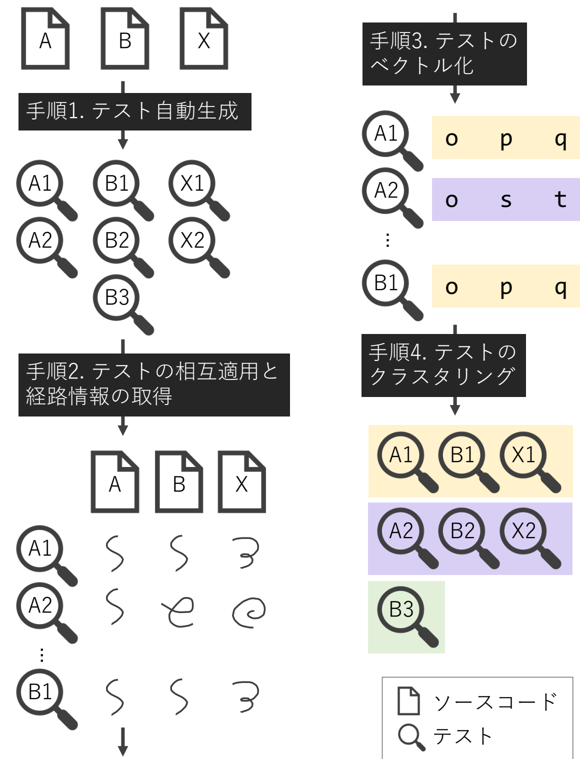

図 4: 提案手法の流れ

3. 提案手法

3.1 概要

本研究の目的は，学生の提出した複数ソースコード間での機能差の自動検出である．本研究における機能差検出のキーアイデアは，テスト自動生成 [1] の活用にある．テスト自動生成ではソースコードを入力とし，その振る舞いを確かめる多数のテストを自動的に生成する．この技術の適用により，複数の学生ソースコードから振る舞いの違い，すなわち機能差を検出しうるテストを生成し，そのテストを再度ソースコードに適用して機能差を洗い出す．

この手法の実現に際して，以下 2 つの課題を定める．

- 課題 1：機能差の検出能力を持つテストの生成と分類．単純なテスト生成技術の適用では大量のテストが生成される．この生成テストの中には全ソースコードで共有する，すなわち課題要件の範疇のテストが大量に含まれる．機能差検出能力を持つテストも得られるが，複数の学生が同様の機能を実装していた場合，複数の同機能を試すテストが生成される．最終的な機能差検出処理においては，目視によるテスト内容の意味づけが必要なため，これら多数のテストを適切に分類する必要がある．
- 課題 2：生成テストを用いた機能差の自動検出．テスト自動生成では，同じテスト類（例えば null 確認テスト）であっても，その期待値の確認処理（assert 部分）が矛盾するテストが多数生成される．分類されたテストから適切な assert 部を抽出し，対象となるソースコードにテストを再適用することで機能差の検出を試みる．

本稿では，課題 1 の達成を目指して，テスト自動生成を活用したテストのクラスタリング手法を提案する．課題 2 については 5 節で考察する．

課題 1 に対する提案手法の全体の流れを図 **4** に示す．手法は 4 つの手順から構成されており，入力は学生の提出したソースコード，出力は生成テストのクラスタリング結果である．本図は 2 節と同様，PortNumber クラスを想定している．2 人の学生が提出したソースコード A と X が課題の最低限の実装（図 3 (a)）を，B のみが負の値の指定という異常系を考慮した実装（図 3 (b)）を行っている．以降の節では，図の流れに従って 4 つの手順それぞれについて説明する．

3.2 手順 1．テスト自動生成

まず，テスト自動生成技術を適用して，全ての提出ソースコードから単体テストを生成する．テスト自動生成はこれまで数多くの研究が行われており [1]，EvoSuite[5] や Randoop[8] 等のオープンソースツールも公開されている．テスト自動生成では，遺伝的アルゴリズム等の探索的メタアルゴリズムや実行パスの解析等を用いて，網羅率を最大化するようにテストを生成する．

この手順 1 の目的は，機能差の検出能力を持つテストを得る点にある．テストベース教育においては，教員の用意した事前テストも存在するが，その動作確認は課題の要件の範疇であることが多く，機能差の検出能力を持たない．また一般的なプログラミングにおいては，教員やユーザが想定していない工夫やバグも数多く発生するため，その全

© 1959 Information Processing Society of Japan

体を事前に把握してテスト化することは現実的ではない．

それに対し，テスト自動生成技術は網羅率の最大化を目的としてテストを生成する．ソースコード間における機能差は分岐等の違い，すなわち実行パスの違いとして現れることが多い．よって，網羅率最大化を目的関数としたテスト生成によって，その機能差を検出しうるテストが得られると考えられる．

図 4 では，3 人の学生の提出ソースコード ABX から，それぞれ複数のテストが生成されている．ソースコード AX と比べて B のみ生成テスト数が多い．これは B が負の値の指定という異常系を考慮しており，他のソースコードと比べて実行パスが多いためである．

A と B から生成された具体的な生成テストの一例を図 **5** に示す．この例は実際に図 3 (a) と図 3 (b) のソースコードに EvoSuite を適用して得られたテストである．4 つのテストはいずれも PortNumber クラスに特定のポート番号を指定してインスタンス化し，それが適切に代入されているかを確認している．まずテスト A1 と B1 に着目する．この 2 つのテストは指定するポート番号自体は異なるものの，いずれもポート番号として正常の範囲（0〜65535）内であり，課題の範疇のテストであるといえる．他方，A2 と B2 は負の値の指定という異常系のテストであると解釈できる．ただし，A2 ではコンストラクタで指定した負の値がそのまま代入されることを期待しているのに対し，B2 は負の値の代わりにゼロが代入されることを期待している．

テスト自動生成では，真のテストオラクルは未知である [9] ため，assert 節にはそのテストを題材に適用して得られた結果がそのまま用いられることが多い．ここでテストオラクルとは，ポート番号として負の値を受け付けないという普遍的な事実であるといえる．よって，バグを含むソースコードからはバグを期待する assert 節が生成される．テスト A2 は異常系を考慮していない，バグを含むソースコードから生成されたバグを期待するテストであるといえる．

```
1  @Test public void testA1() {
2    PortNumber portNumber0 = new PortNumber(992);
3    int int0 = portNumber0.getNumber();
4    assertEquals(992, int0);
5  }
6  @Test public void testA2() {
7    PortNumber portNumber0 = new PortNumber(-1);
8    int int0 = portNumber0.getNumber();
9    assertEquals(-1, int0);
10 }
```

(a) 学生 A から生成されたテスト

```
1  @Test public void testB1() {
2    PortNumber portNumber0 = new PortNumber(654);
3    int int0 = portNumber0.getNumber();
4    assertEquals(654, int0)
5  }
6  @Test public void testB2() {
7    PortNumber portNumber0 = new PortNumber(-4824);
8    assertEquals(0, portNumber0.getNumber());
9  }
```

(b) 学生 B から生成されたテスト

図 5: テスト自動生成ツールによる生成テストの一例

3.3 手順 2．テストの相互適用と経路情報の取得

手順 2 では，まず手順 1 で生成されたテストを全てのソースコードに対して相互に適用する．N 個のソースコードから，計 M 個のテストが生成された場合，$N \times M$ のテスト実行が行われる．さらに各テスト実行において，その実行経路を記録する．テストの相互実行と経路情報の記録は，最終的なテストのクラスタリングのための前処理である．

ここで最終的なテストクラスタリングにおいて，どのような基準でテストの類似度を算出するべきかについて考える．本研究の最終目標は機能差の自動検出であり，各テストがどのような機能を試すテストであるか，という情報を類似度とするべきだといえる．例えば，図 5 の 4 つのテストの場合，A1 と B1 は具体的なポート番号自体は異なるものの，いずれも 0〜65535 の範囲の正常系のテストであると見なせる．同様に，A2 と B2 はポート番号の違いや assert 節の違いはあるものの，0〜65535 の範囲外の一種の異常系テストであるといえる．すなわち，A1 と B1，さらに A2 と B2 を同一集合として分類するための類似度の基準を考える必要がある．

基準の一つの選択肢は，テスト実行時の成否（pass/fail）という情報の利用である．しかし，A2 のようなバグ時の挙動を期待するバグを含むテストも生成されるため，類似度の基準としては利用が困難である．例えば，A2 と B2 は同じ集合として扱うべきであるが，その assert 節は互いに矛盾しており，テストの成否も異なってしまう．また今回の題材の場合，B2 は負の値にゼロを代入する，という異常系処理に限定されている一方で，負の値の場合に例外を投げる，標準エラーにエラーの旨を通知する，等異常時の対策は多種多様であり，その対策に応じて様々な assert 節が生成されてしまう．そのためテスト成否，すなわち assert 部を含めてテストをクラスタリングした場合，テスト分類結果の過剰な細分化を引き起こす可能性がある．

よって提案手法では，テストを，動作確認の手順を示した実行部，及び実行結果と期待値との比較を行う assert 部，2 つに分解し，実行部のみの情報に基づいてテストをベクトル化する．この実行部のみの情報として，実行時の経路情報を採用する．経路情報の利用により，assert 部を排除した各テストの動作確認手順という観点からのクラスタリングが可能となる．

3.4 手順 3．テストのベクトル化

手順 2 で得られた経路情報を ID 化し，各テストのベクト

© 1959 Information Processing Society of Japan

ルを算出する．図 4 の手順 2 では各テストがソースコードの何行目を通過したか，という経路を記録するため，ソースコード間で経路の重複が発生する．しかし，ソースコード間での経路情報の比較は成り立たない．例えば，テスト A1 をソースコード A に適用した場合（以降，A1⇒A と記載）S の文字のような経路を取っており，A1⇒B も同経路 S を辿っている．しかし，2 つのソースコードの処理の内容が異なれば，同じ経路でもその経路の意味が異なるため同一視することはできない．

よって，経路情報のベクトル化の際には，学生 x のパス ID 情報 P_x は任意の学生 y に対して，以下の式が成り立つ必要がある．

$$|P_x| \cap |P_y| = \phi \quad (1)$$

本手法では，テスト実行時の各行の通過情報を文字列として直列化して学生 ID を付与する．これを MD5 等の処理によってハッシュ化することで学生間の重複を回避する．例えば，学生 x にあるテストを実行した際の経路情報が 11101（4 行目のみ通過しなかった，という意味）だった場合，x11101 という文字列を生成してハッシュを得る．このようなハッシュ化処理を全ての経路情報に適用することで，式 (1) を満たしつつテストのベクトルが得られる．図 4 ではハッシュ値を簡略しているが，テスト A1 と B1 は [o,p,q]，テスト A2 は全く異なるベクトル [o,s,t] を持つことが確認できる．なお，ベクトルの成分（ハッシュ値）は ID，すなわち名義尺度でありその大小の比較には意味を持たない．

3.5 手順 4. テストのクラスタリング

最後に，手順 3 で作成したベクトルに基づいてテストをクラスタリングする．図の場合，手順 2 で生成した計 7 個のテストは最終的に 3 種類のテストに分類できている．テスト A1，B1，X1 は同じ集合として，さらに A2，B2，X2 が同じ集合として分類されている．手順 2 でも説明した通り，経路情報を活用することで，テスト内での期待値との比較（assert 部）を排除した，テストの動作確認手順を基準とした分類が可能となっている．

本節概要でも述べたとおり，提案するテストクラスタリング手法は受講者 100 人規模のソースコードに対しても，人手を要さない完全自動化が可能であり，大量の生成テストをその確認手順の類似度という観点から分類が可能である．なお，本図での題材は数行程度であり，ベクトルの完全一致のみで分類が可能であるが，より実践的な題材ではこの限りではない点に注意されたい．

4. 予備実験

本予備実験の目的は，前節で述べた提案手法により機能差の検出能力を持つテストを生成し，適切に分類できるかの確認である．そのために数行程度のプログラミング題材に対して提案手法を適用した．

4.1 実験題材

実験に用いた題材は 2 節，3 節に引き続き PortNumber クラスである．想定する機能差は 2 つの要素の組み合わせで構成される．一つはポート番号の範囲確認の有無であり，もう一つは異常時処理の種類である．範囲確認は，下限確認（`n < 0`）と上限確認（`n > 65535`），及びその両方の 3 種類である．異常時の処理は，指定ポート番号が範囲外だった際に代替値（ゼロ）を代入するか，例外を発生させる（`throw new Exception`）かの 2 種類である．

上記 2 つの要素の組み合わせの結果，**表 1** に示す 7 種類の機能差を持つ題材を作成した．題材名称は範囲確認と異常時処理の組み合わせとなっている．例えば，一番上の CheckNone は一切の範囲確認処理を行っておらず，指定された値をそのまま代入している．課題要件の最低限の実装であるといえ，図 3 (a) に相当する．CheckLow-SetAltValue では下限確認の後，下限を下回っていた場合に代替値を代入している．上限確認は行っていない．これは図 3 (b) に相当し，続く CheckLow-ThrowException は図 3 (c) に相当する．

本予備実験で想定する機能差は，2 種類の範囲確認処理を行っているか否かであり，異常時処理の内容は無視されるべきである．例えば，表 1 の下 4 行（CheckUpp-*と CheckBoth-*）は異常時の振る舞い自体は異なるものの，いずれも上限確認という機能差を持っている，と検出されることが望ましい．この機能差の定義に関しては 3.3 節を確認されたい．

4.2 実験手法

実験としてはまず，著者らが手動で 7 種類の題材ソースコードを作成した．次に，テスト自動生成ツール EvoSuite を用いて題材ソースコードのテストを自動生成した．EvoSuite の各種パラメタは初期設定とした．続いて，生成された各テストを各ソースコードに対して相互に実行した．テスト実行時には経路情報を取得するために，実行経路計測

表 1: 実験題材

題材名称	n < 0 時	n > 65535 時
CheckNone	n を代入	n を代入
CheckLow-SetAltValue	0 を代入*a	n を代入
CheckLow-ThrowException	例外発生*a	n を代入
CheckUpp-SetAltValue	n を代入	0 を代入*b
CheckUpp-ThrowException	n を代入	例外発生*b
CheckBoth-SetAltValue	0 を代入*a	0 を代入*b
CheckBoth-ThrowException	例外発生*a	例外発生*b

*a 想定する機能差 1，*b 想定する機能差 2

© 1959 Information Processing Society of Japan

ソースコード テスト	ChkNone	ChkLow-SetAltV	ChkLow-ThrowE	ChkUpp-SetAltV	ChkUpp-ThrowE	ChkBoth-SetAltV	ChkBoth-ThrowE	集合ID
ChkNone1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone3	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE2	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkLow-ThrowE3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkUpp-SetAltV4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkUpp-ThrowE4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkBoth-SetAltV4	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkBoth-SetAltV5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkBoth-ThrowE4	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkBoth-ThrowE5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
生成テスト数	3	3	3	4	4	5	5	
経路種類の数	2	4	3	4	3	5	3	

図 6: 生成テストと実行時経路の一覧

ソースコード テスト	ChkNone	ChkLow-SetAltV	ChkLow-ThrowE	ChkUpp-SetAltV	ChkUpp-ThrowE	ChkBoth-SetAltV	ChkBoth-ThrowE	集合ID
ChkNone3	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-ThrowE2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkBoth-SetAltV4	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-ThrowE2	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkBoth-ThrowE4	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkUpp-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkBoth-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkUpp-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkBoth-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5

図 7: 生成テストと実行時経路の一覧（ID1 を削除）

ツール JaCoCo[*2]を使用した．最後に，JaCoCo によって得られた経路情報に各ソースコードの題材名称を付与し，`md5sum` コマンドを用いて経路 ID となるハッシュ値を算出した．

4.3 結果と考察

実験により得られた各テストの経路 ID を**図 6** に示す．列がソースコード，行がテストを表しており，値は経路のハッシュ値である．視認性向上のため，列ごとの同一ハッシュを同一色で塗りつぶしている．テストの名称は，そのテストの生成元となったソースコード名に連番を付与した名前となっている．なお，縦方向のテストの順序は EvoSuite が生成した順序をそのまま採用しており，意味を持たないことに注意されたい．表の右列の集合 ID は，実行経路 ID から算出された各テストのクラスタリング結果である．この場合，経路の完全一致に基づいてクラスタリングを行っている．さらに，テストの各集合 ID のテスト内容を表 **2** に，集合 ID ごとの具体的な生成テストの例を**図 8** に示す．以降では，これらの結果に基づき，テスト自動生成とテストクラスタリングの 2 つそれぞれの結果について説明する．

4.3.1 テスト自動生成

まず図 6 の結果をテスト自動生成の観点から説明する．生成テストの数に関しては，CheckNone と CheckLow-*が 3 個，CheckUpp-*が 4 個，CheckBoth が 5 個と機能差を含むソースコードほど多くのテストが生成される傾向にある．CheckNone と CheckLow-*でテスト数に差がない理由は，EvoSuite のテスト生成戦略に起因する．EvoSuite は int 引数に負の値を渡すというヒューリスティックを採用しており，下限確認の有無がテスト数として表れなかったといえる．

ソースコードごとの経路の種類（ハッシュ値の種類）という観点では，やはり機能差が多いほど様々な経路を取る傾向にある．CheckNone が 2 種類のみに対し，CheckLow-SetAltValue は 4 種類，最大では CheckBoth-SetAltValue の 5 種類である．ただし，*-ThrowException は*-SetAltValue と比べて経路種類が少ない．これは例外を期待するテストが getter メソッドを呼び出さないことに起因している．例えば，生成テストの具体例図 8 (b) と図 8 (c) は，下限以下の値をコンストラクタに渡すという点では同じテストであるが，その後に getter メソッドを使って値を確認するか，例外発生を期待するかが異なる．前者のみ getter メソッドを呼び出しているため，後者と経路が異なる．

4.3.2 経路の計測とクラスタリング結果

クラスタリングの結果，多くのテストが集合 ID1 に割り振られた．表 2 に示す各集合のテスト内容，及び図 8 (a) のテストの実例を確認すると，`n` の値自体は様々ではあるものの，いずれも正常範囲（$0 \leq$ `n` ≤ 65535）のテストであり，課題要件の範疇内のテストであるといえる．この集合 ID1 のテストは全てのソースコードから生成される，共有機能であると解釈できる．

視認性向上のため図 6 の結果に対し，集合 ID1 のテストを削除し集合 ID で並び替えた結果を**図 7** に示す．図 8 (b) に示す集合 ID2 のテストは，負の値を指定してその代入値を確認するテストである．ここで代入値の期待値は負の値そのものである場合と，代替値ゼロである場合の両方が存在していた．assert 部の内容に依存することなく，負の値を試すというテストが適切に分類されていたといえる．この結果は図 8 (d) に示す集合 ID4 の，上限確認のテストでも同様であった．

他方，集合 ID3 と 5 は過剰に細分化されたテスト群であった．ID3 は `n` ≤ 0 のテストという観点では ID2 と同一，ID5 は `n` ≥ 65535 のテストという観点で ID4 と同一と分類されることが期待される結果であった．その理由に

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

© 1959 Information Processing Society of Japan

```
1  @Test public void CheckNone2() {
2    PortNumber portNumber0 = new PortNumber(992);
3    int int0 = portNumber0.getNumber();
4    assertEquals(992, int0);
5  }
```

(a) 集合 1 のテストの例

```
1  @Test public void CheckNone3() {
2    PortNumber portNumber0 = new PortNumber(-1);
3    int int0 = portNumber0.getNumber();
4    assertEquals(-1, int0);
5  }
```

(b) 集合 2 のテストの例

```
1  @Test public void CheckLowThrowException2() {
2    PortNumber portNumber0 = null;
3    try {
4       portNumber0 = new PortNumber(-394);
5       fail("Expecting␣exception:␣
             IllegalArgumentException");
6    } catch(IllegalArgumentException e) {
7      verifyException("PortNumber", e);
8    }
9  }
```

(c) 集合 3 のテストの例

```
1  @Test public void CheckUppSetAltValue3() {
2    PortNumber portNumber0 = new PortNumber(65545);
3    assertEquals(0, portNumber0.getNumber());
4  }
```

(d) 集合 4 のテストの例

```
1  @Test public void CheckUppThrowException3() {
2    PortNumber portNumber0 = null;
3    try {
4       portNumber0 = new PortNumber(65558);
5       fail("Expecting␣exception:␣
             IllegalArgumentException");
6    } catch(IllegalArgumentException e) {
7       verifyException("PortNumber", e);
8    }
9  }
```

(e) 集合 5 のテストの例

図 8: 各集合のテストの例

ついて考察する．図 8 (c)，及び図 8 (e) を確認すると，いずれも例外発生を期待するテストである．加えて，assert 節における getter メソッドの呼び出しが含まれていない．getter の呼び出しはテスト実行時の経路の一種であり，この有無が経路ベクトルの違いとクラスタリング結果の過剰な細分化に繋がっていると考えられる．実際に，図 7 の CheckNone の列に着目すると，CheckNone は最低限の機能実装であるにも関わらず，765d と 258a の 2 種類の経路を取っている．765d は getter の通過を含んだテスト実行を，258a は含まないテスト実行であると読み取れる．図 7 の集合 ID を確認しても，258a の場合に集合 ID3 と 5 を取っていることが確認できる．このように例外を期待するテストは過剰に細分化される傾向にあり，対策が必要であると考えられる．

5. 提案手法の改善に向けての考察

本節では，提案手法を改善するために 2 つの考察を行う．1 つ目は今回の実験で発生した経路の過剰な細分化を防ぐための方法，2 つ目は 3 節で示した課題 2 の解決方法である．

5.1 経路の改善

過剰に細分化されたテスト群が得られる問題は，教育の現場でも発生する可能性が高い．NullPointerException 等の実行時例外は，対策している学生の方が少なく，多くの場合で発生すると思われる．その上，実行時例外はソースコード中の様々な場所で発生するため，経路の種類は無数に増えてしまう．

例外を期待するテストは自動で判別できるため，そのようなテストに対して他とは異なる処理を行うことで過剰な細分化を改善できる可能性がある．例えば図 8 (c), (e) に示す 2 つのテストは，catch 節である 7 行目に verifyException という assert 部を含んでるので例外を期待するテストであると判別できる．判別したテストに対する処理内容としては，クラスタリング時の該当テストの排除が考えられる．これによって表 2 の集合 3 と集合 5 がなくなり，代入する n の範囲だけに基づくクラスタリングができる．

また，経路情報を過剰に細分化されないための工夫として，実行部と assert 部の切り離しも考えられる．これは一時変数の導入リファクタリングで容易に行える．今回の実験では存在しなかったが，1 つのテストに assert が複数あり途中でテスト失敗する時には，先ほどの実行時例外と同様に経路が変わってしまう．実行部と assert 部の切り離しによって，より正確に実行部だけの経路で評価できる．

5.2 生成テストを用いた機能差の自動検出

以降では，3 節で示した課題 2 について考察する．課題 1 の達成によって機能差の検出能力があるテストの生成及び適切な分類が実現した．この生成テストを用いて機能差を検出するには，テストを全てのソースコードに適用し，成否を確認することが考えられる．例えば，各集合からテストを 1 つ選択し，そのテストに通過する学生はその集合が表す機能を持っているという検出方法である．

しかし，この方法では各集合からテストを 1 つ選択する際に，テスト自動生成によって様々な assert 節が生成されている点が問題となる．互いに矛盾した assert 節を持つテストも存在するため，テストの選び方次第ではテスト通

表 2: テストの集合 ID とその内容

集合 ID	代入する n の範囲	期待値
1	$0 \leq n \leq 65535$	n
2	$n < 0$	n か代替値 0
3	$n < 0$	例外発生
4	$65535 < n$	n か代替値 0
5	$65535 < n$	例外発生

© 1959 Information Processing Society of Japan

過の意味も大きく異なる．例えば，図 4 の紫の集合から図 5 (a) のテスト A2 を選択した時，テストの通過は下限確認を行っていないことを示すが，図 5 (b) のテスト B2 と選択した時，テストの通過は下限確認を行っていることを示す．

この問題を解決するために，assert 節に基づくクラスタリングの追加実施を検討している．3 節で説明した手法によってテスト集合を分類し，その中でさらにテスト実行時の成否に基づくベクトルによってクラスタリングする．これによって，動作確認手順によって分類された集合の中に，assert 節によって分類された部分集合が形成される．

最後に，これまでに得られた集合を元に機能差を検出する．動作確認手順と assert 節の 2 つの基準に基づく部分集合を得ているので，各部分集合からテストを 1 つ選択し，各提出ソースコードに対するそのテストの成否を確認する．テストの通過は，テストを適用したソースコードにその機能があることを意味し，テスト失敗はその逆である．一部のソースコードだけ通過する機能こそがソースコード固有の機能差であると考えられる．

テスト成否によって自動的に判別できるのはどのテスト，すなわちどの機能がどの学生に存在するかだけであり，テストの意味は別の手法を用いて付与する必要がある．意味の付与は，現状では目視を想定している．人手を要するが，確認するべき内容はブラックボックステストに隠蔽されるため，全提出ソースコードの目視確認と比べると教員の負担が少ない．また，教員の負担を減らすためには全ての部分集合のテストを確認するのではなく，テストを通過するソースコードが少ないテストから優先的に確認することも有効であると考えられる．

6. おわりに

本研究では，テストベース教育におけるソースコードの適切な祭典を目的として，提出ソースコード間の機能差の自動検出に向けた課題を整理した．さらに，テスト自動生成，及びテスト実行時の経路情報を活用したテストクラスタリング手法を提案した．また，予備実験として行った数行程度のプログラミング題材に対する手法の適用結果について報告した．実験結果から，提案手法はテストの動作確認手順を基準としてテストを分類できることを確認した．また，長期的な目標を達成するためにクラスタリングしたテストから機能差を検出する手法を考察した．

今後は以下の課題に取り組む予定である．まず，予備実験で発生した過剰なテストの細分化の対策が必須である．また，課題 2 として挙げた生成テストを用いた機能差の自動検出は残された重要な課題である．課題 2 の達成により，多数のソースコードを対象とした全自動での機能差検出が実現できる．さらに，検出した機能差が肯定的な内容か，否定的な内容かを推測する技術についても検討中である．特に否定的な機能は多くの場合，事前テストへの過剰適合に起因しており，いくつかのヒューリスティックにより検出可能であると考えている．教育現場への適用も重要な課題であり，著者らの実施しているプログラミング演習への導入を検討している．

謝辞 本研究の一部は，JSPS 科研費 18H03222，17K00500，19K02973，19K03001 の助成を得て行われた．

参考文献

[1] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P., Bertolino, A., Jenny Li, J. and Zhu, H.: An orchestrated survey of methodologies for automated software test case generation, *Journal of Systems and Software*, Vol. 86, No. 8, pp. 1978–2001 (2013).

[2] Desai, C., Janzen, D. and Savage, K.: A Survey of Evidence for Test-Driven Development in Academia, *SIGCSE Bulletin*, Vol. 40, No. 2, pp. 97–101 (2008).

[3] Eddy, B. P., Wilde, N., Cooper, N. A., Mishra, B., Gamboa, V. S., Shah, K. M., Deleon, A. M. and Shields, N. A.: A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses, *In Proc. Conference on Software Engineering Education and Training*, pp. 47–56 (2017).

[4] Edwards, S. H. and Pérez-Quiñones, M. A.: Experiences Using Test-Driven Development with an Automated Grader, *Journal of Computing Sciences in Colleges*, Vol. 22, No. 3, pp. 44–50 (2007).

[5] Fraser, G. and Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software, *In Proc. ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pp. 416–419 (2011).

[6] Ihantola, P., Ahoniemi, T., Karavirta, V. and Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments, *In Proc. Koli Calling International Conference on Computing Education Research*, pp. 86–93 (2010).

[7] Kaner, C., Cem, C. and All, K.: The Impossibility of Complete Testing (1997).

[8] Pacheco, C. and Ernst, M. D.: Randoop: Feedback-Directed Random Testing for Java, *In Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 815–816 (2007).

[9] Pastore, F., Mariani, L. and Fraser, G.: CrowdOracles: Can the Crowd Solve the Oracle Problem?, *In Proc. International Conference on Software Testing, Verification and Validation*, pp. 342–351 (2013).

[10] Restrepo-Calle, F., Ramírez Echeverry, J. J. and González, F. A.: Continuous assessment in a computer programming course supported by a software tool, *Computer Applications in Engineering Education*, Vol. 27, No. 1, pp. 80–89 (2019).

© 1959 Information Processing Society of Japan