修士学位論文

題目

メソッドの経路解析に基づくモック活用型単体テストの自動生成

指導教員

楠本 真二 教授

報告者

渡邉 凌雅

令和 7 年 1 月 28 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 6 年度 修士学位論文

メソッドの経路解析に基づくモック活用型単体テストの自動生成

渡邉 凌雅

内容梗概

単体テストの効率化を目的として，網羅率の高い単体テストを自動生成する研究が行われている．代表的な単体テスト自動生成ツールである EvoSuite は，遺伝的アルゴリズムを用いて網羅率の高いテストスイートを探索的に生成する．しかし，探索空間の限界により，命令や分岐の網羅に必要な依存オブジェクトを適切に構築できない場合がある．本研究では，テスト対象メソッドに対する経路解析に基づき，モックを活用した単体テストを自動生成する手法を提案する．キーアイデアは，探索的手法では適切に構築できない依存オブジェクトに対し，メソッドに対する経路解析で得られた情報を元に依存オブジェクトを代替するモックを構築することである．具体的には，制御フローグラフを用いて分岐までの経路を特定し，データフロー解析によって経路の実行に必要な値や依存オブジェクトの動作を導出する．その後，導出した値や動作を元にモックを構築し，分岐に到達可能なテストケースを生成する．提案手法の有効性を検証するため，10 個の Java プロジェクトに含まれる 1,977 個の分岐を対象に評価実験を実施した．その結果，提案手法によって生成したテストスイートによって EvoSuite では網羅できなかった分岐が新たに 96 個網羅され，分岐網羅率が 4.85% 向上した．

主な用語

単体テスト自動生成，EvoSuite，モック，制御フローグラフ

目次

1 はじめに 1

2 準備 3

2.1 単体テスト . . . 3

2.2 網羅率 . . . 3

2.3 単体テスト自動生成 . . . 3

2.4 EvoSuite . . . 4

2.5 EvoSuite の課題 . . . 5

2.6 モック . . . 6

2.7 制御フローグラフによるプログラム解析 . . . 9

2.8 記号実行 . . . 9

3 提案手法 11

3.1 目標分岐に到達する経路の特定 . . . 11

3.2 経路上の条件式解析と具体値の導出 . . . 13

3.3 経路上のオブジェクトに対する動作の特定 . . . 14

3.4 テストケースの出力 . . . 15

4 評価実験 17

4.1 提案手法の実装 . . . 17

4.2 評価方法 . . . 17

4.3 結果 . . . 18

5 考察 20

5.1 テスト対象クラスのインスタンス化に失敗 . . . 20

5.2 テスト対象メソッドの外部からの操作では経路通りに実行できない . . . 22

6 妥当性の脅威 25

7 おわりに 26

謝辞 27

参考文献 28

図目次

1 EvoSuite の動作概要 4
2 EvoSuite のテストスイートでは網羅されない分岐が存在するクラスの例（説明のためコードを一部変更） 5
3 EvoSuite が生成したテストケースの例（説明のためコードを一部変更） 6
4 Mockito におけるモックの生成とスタブの設定 7
5 Mockito におけるスパイの生成とスタブの設定 7
6 図 2 にのメソッドに対応する制御フローグラフ 8
7 記号実行による条件分岐の解析対象メソッド 9
8 `row != null` が成立する分岐に到達する経路の特定 12
9 引数と代入文に与える値，およびオブジェクトの動作に要求される条件の特定 13
10 `this` に対応するスパイにスタブを設定するコード片 15
11 生成されるテストケース 16
12 テスト対象クラスのインスタンス化に失敗した例 21
13 `doGet()` から生成されたテストケースの例 21
14 提案手法が適用可能となるリファクタリングを実施した後の `RequestParamExample` クラス 22
15 テスト対象メソッドの外部からの操作では経路通りに実行できない例 23
16 `checkConsistency()` から生成されたテストケースの例 23
17 プログラム構造上網羅不可能な例 24

表目次

1 プリミティブ型の値に対する基本条件 12
2 参照型の値に対する基本条件 13
3 評価対象全体で網羅された分岐数と分岐網羅率 18

1　はじめに

単体テストは，ソフトウェアのプログラムを構成する最小単位に対し，開発者がその動作を検証し仕様を満たしているか確認する工程である．単体テストを実施するにあたって，開発者はテスト対象プログラムの内部構造を網羅的に検証できるテストを作成しようとする．しかし，開発者がそのようなテストを手動で作成するには多大な労力を要する．そこで，単体テストの労力削減を目的として，テスト対象プログラムの内部構造を可能な限り網羅する単体テストを自動生成する研究が行われている．単体テスト自動生成の主流なアプローチとして，探索ベースソフトウェアテスト（Search-based Software Testing，SBST）がある [1, 2]．SBST では，メタヒューリスティックな手法 [3] を用いて，内部で定めた網羅目標を達成するテストケースの集合（テストスイート）を探索的に求める．

EvoSuite[4] は，遺伝的アルゴリズム [5] を用いて Java のテスト対象クラスに対する単体テストのテストスイートを自動生成する SBST ツールである．EvoSuite では，テストケースを構成するプログラム文を変異させながら，適応度関数を基準にテストスイートを反復的に進化させることで，網羅率の高いテストスイートを生成する．しかし，EvoSuite はテスト対象クラス内の全ての行や分岐を網羅するテストケースを生成できるわけではない．その原因の一つとして，特定の行や分岐を網羅するために必要なテスト対象クラスの依存オブジェクトを適切に構築できないことが挙げられる．しかし，適応度関数の地形に起因する問題 [6, 7] により，依存オブジェクトを適切に構築するテストケースを探索的に求めることは困難である [8]．

本研究では，テスト対象メソッドに対する経路解析に基づき，モックを活用した単体テストを自動生成する手法を提案する．本研究のキーアイデアは，EvoSuite のような探索的手法では適切に構築できない依存オブジェクトに対し，解析的手法とモックを用いて依存オブジェクトを構築することである．具体的には，テスト対象メソッドの経路解析を通して，網羅に必要な依存オブジェクトの動作を特定し，その動作を再現するモックを生成する．モックとは，実オブジェクトを模倣したオブジェクトである．モックに対しては，スタブと呼ばれる操作によってそのオブジェクトの動作を自由に設定できる．この性質を活用し，特定した動作を再現するモックを自動生成し，依存オブジェクトの代替として使用する．

提案手法では，テスト対象メソッドにおける網羅したい分岐（目標分岐）に対して，その分岐へ確実に到達するテストケースを生成する．これを実現するために，実行されるメソッドの経路を固定し，かつ経路に含まれるプログラム文が確実に実行されるテストケースを構成する．経路通りに実行されることを保証するには，以下の条件を保証したテストケースを生成する必要がある．

条件 1　経路に含まれる分岐条件が常に経路通りに評価される．

条件 2　経路上の全ての命令が確実に実行される．すなわち，実行途中で例外が発生しない．

まず，経路上に含まれる分岐条件に登場する変数が使用されている箇所を探索し，条件 1 を満たすために必要な各変数に対する条件式を抽出する．その後，記号実行 [9] における条件式の解析と具体値の導出手法を応用し，抽出した条件式を満たす具体値を導出する．次に，経路に含まれるオブジェクト操作（メソッド呼び出し，フィールド参照，インスタンス生成）を解析し，条件 2 を満たすために必要な依存オブジェクトの動作を特定する．その後，その動作を再現した依存オブジェクトのモックを生成する．最後に，依存オブジェクトをモックで代替したテストケースを構成し出力する．

提案手法の有効性を確認するために，提案手法を実現するプログラム MockTestGen を実装し評価実験を実施した．EvoSuite 単体で生成されたテストスイートをベースラインとし，提案手法で生成したテストケースを追加することで，EvoSuite 単独と比較して新たに分岐がどれだけ網羅されたかを検証した．10 個の Java プロジェクトに含まれる 1,977 個の分岐に対して評価実験を実施したところ，EvoSuite では網羅できなかった分岐が新たに 96 個網羅され，分岐網羅率が 4.85% 向上した．

2 準備

2.1 単体テスト

単体テストとは，ソフトウェアのプログラムを構成する最小単位に対し，開発者がその振る舞いを検証し仕様を満たしているか確認する工程である．単体テストはソフトウェア実装後の早期段階で実施される．そのため，プログラムに潜むバグを早期に発見し，全体のテスト工程における修正作業に伴うコストを削減することができる．

単体テストの手法の一つに構造テストがある．構造テストでは，テスト対象プログラムの内部構造に着目し，プログラムの制御フローやデータフローを網羅的に検証する．構造テストを十分に実施するには，テスト対象プログラムの内部構造を網羅的に検証できるテストを作成する必要がある．

2.2 網羅率

テストの網羅性を測る基準として網羅率 [10] が使われる．網羅率は，プログラムに含まれる命令や分岐のうちテストによって実行された割合を表す指標である．開発者は，テスト対象プログラムの網羅率を可能な限り最大化するテストを作成しようとする．しかし，プログラム内の全ての命令や分岐を網羅的に検証するためには，各経路や条件ごとに異なる入力や環境を用意する必要がある．そのため，開発者が網羅率の高いテストを人力で作成するには多大な労力を要する．

2.3 単体テスト自動生成

単体テスト自動生成とは，単体テストの効率化を目的として，テスト対象プログラムのソースコードをもとに網羅率の高い単体テストを自動的に生成する技術である．これまでにランダムな入力値を基にした手法 [11, 12] や，記号実行 [9] を活用した手法 [13, 14, 15, 16] など，様々なテスト自動生成手法が提案されてきた．

特に，メタヒューリスティックな手法 [3] を用いて探索的にテストを生成する探索ベースソフトウェアテスト（Search-based Software Testing，SBST）[1, 2] は，多様かつ膨大な入力の組み合わせが考えられる大規模で複雑なプログラムに対しても網羅率の高いテストを自動生成できるため，近年での単体テスト自動生成の主流である．SBST では，探索アルゴリズムを使用して，内部で定めた網羅目標を達成するテストスイート（テストケースの集合）を探索的に求める．代表的な SBST ツールとして，遺伝的アルゴリズム [5] を用いて指定した網羅基準を満たすテストスイートを生成する EvoSuite[4] がある．

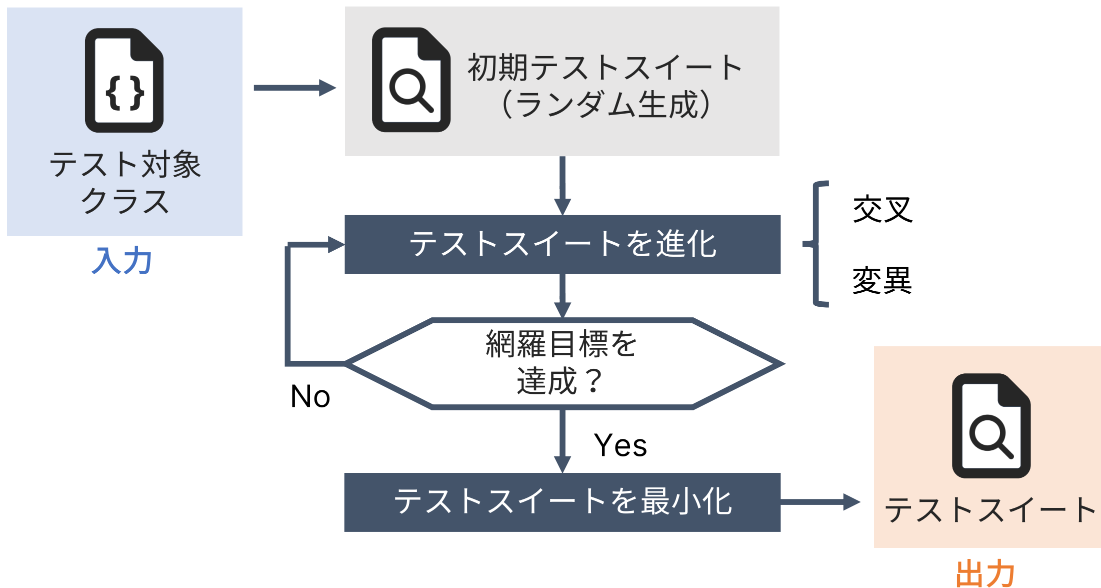

図 1　EvoSuite の動作概要

2.4　EvoSuite

EvoSuite は，Java 向けの代表的な SBST ツールであり，テスト生成の探索アルゴリズムとして遺伝的アルゴリズムを使用する．遺伝的アルゴリズムとは，生物の進化過程を模倣した近似解探索アルゴリズムである．解の候補を個体，個体が進化する方向性を適応度関数として表現し，個体の集合に対して進化的操作を繰り返し適用することで適応度の高い個体を探索する．EvoSuite では，テストケースを個体，プログラムの行や分岐の実行に必要な値までの距離を適応度関数として表現し，網羅目標を効率的に達成するテストスイートを生成する．また，EvoSuite には目標選択アルゴリズムの改良 [17] や，ハイブリッド探索 [18]，動的記号実行 [16]，テスト容易性変換 [19] など，これまでのテスト自動生成分野で得られた知見が取り入れられている．EvoSuite は，他の SBST ツールと比較して網羅率の高いテストスイートを生成することが可能であり，2021 年の SBST ツールコンペティションでは参加した SBST ツールの中で最も網羅率の高いテストスイートを生成した [20].

図 1 に EvoSuite の動作概要を示す．まず，テスト対象クラスのメタデータを収集し，それを元にテストケースをランダムに生成して初期世代を構築する．その後，交叉や変異といったプログラム文を変更する操作を適用してテストケースを進化させる．交叉は複数のテストケースのプログラム文を部分的に組み合わせて，新たなテストケースを生成する操作である．一方，変異はテストケースの一部を変更する操作を指す．これらの進化的な操作は，網羅目標を達成するか探索時間の上限に達するまで繰り返される．最後に，実行箇所が重複するテストケースを削除する最適化を実施したのち，テストスイート

```java
public class Template {

  private Sheet sheet;

  private int width() {
    // ...
  }

  public int getRowHeight(int r) {
    int rh = 0;
    int w = this.width();

    for (int c = 0; c < w && w < 10; c++) {
      Reference ar = new AbsoluteReference(r, c);
      HSSFRow row = this.sheet.getRow(ar.row());
      if (row != null) {
          rh = Math.max(rh, row.getHeight());
      }
    }
    return rh;
  }
}
```

図 2　EvoSuite のテストスイートでは網羅されない分岐が存在するクラスの例（説明のためコードを一部変更）

として出力する．

2.5　EvoSuite の課題

EvoSuite は，テスト対象クラス内の全ての行や分岐を網羅するテストスイートを生成できるわけではない [21]．110 個の Java プロジェクト [22] に対し，EvoSuite v1.0.6 で生成されたテストスイートの行網羅率は 73.6%，分岐網羅率は 75.7% に留まる [23]．その原因の一つとして，特定の行や分岐を網羅するために必要な内部状態を持つ依存オブジェクトを構築できないことが挙げられる．ここでの依存オブジェクトとは，テスト対象クラスの動作を決定する要因となるオブジェクトを指し，テスト対象クラス自身のインスタンスおよびその静的メンバや，テスト対象クラスが依存する外部クラスのインスタンスおよびその静的メンバが該当する．依存オブジェクトの内部状態を適切に制御するには，コンストラクタに適切な引数を渡したり，setter のような内部状態を変更するメソッドを実行したりする必要がある．しかし，EvoSuite におけるテスト探索では依存オブジェクトを適切に構築できない場合がある．これは，EvoSuite で定義されている適応度関数が，適応度関数の地形に起因する問題 [6, 7] により，依存オブジェクトを構築するためのプログラム文の探索には適切に作用しないためである [8]．

図 2 は，EvoSuite が依存オブジェクトを適切に構築できないことが原因で網羅できない分岐が存在するクラスの例である．また，図 3 は，図 2 のメソッドに対して生成されたテストケースの例である．

```
1  @Test
2  public void test00() {
3    // シートに行とセルを作成する処理がない
4    // (row != null を満たせない)
5    HSSFWorkbook wb0 = new HSSFWorkbook();
6    HSSFSheet sheet0 = wb0.createSheet(…);
7    List<Style> styles = new ArrayList<>();
8    styles.add(new Style(…));
9
10   // …
11
12   Template template0 = new Template("" , sheet0, 2, styles);
13   template0.getRowHeight(1);
14
15   // …
16 }
```

図 3　EvoSuite が生成したテストケースの例（説明のためコードを一部変更）

EvoSuite が生成したテストケースでは，16 行目の `row != null` の `true` 分岐に到達することができない．`row != null` の条件を満たすためには，`this.sheet.getRow(ar.row())` が適切な行オブジェクト `row` を返す必要がある．このメソッドの実行結果には，`Template` クラスの依存オブジェクトである `sheet` の内部状態や設定された値が影響を及ぼす．しかし，EvoSuite が生成したテストケースでは，`sheet` に行とセルを作成する処理が生成されていないため，`row != null` を満たすことができない．

2.6　モック

モックとは，実オブジェクトを模倣したオブジェクトであり，テスト対象メソッドが持つ依存オブジェクトを代替し，単体テストにおけるテストケースの独立性を高める目的で広く使用されている [24]．単体テストの目的は，テスト対象メソッドそれ自体の動作を検証することである．しかし，多くの場合，テスト対象メソッドの動作は外部クラスや Java 外部のシステムの動作に依存する．例えば，自パッケージ内の別クラスや別パッケージに属するクラス，データベースやファイルシステムといった Java 外部のシステムが該当する．そこで，モックを使用してテスト対象メソッドを依存オブジェクトから切り離すことで，テスト対象メソッド単体の動作を検証したテストを作成することが可能となる．

また，モックしたオブジェクトのメソッド呼び出しやフィールド参照に対し，開発者が戻り値や例外などを任意に設定することができる．これをスタブという．適切なスタブの設定によって特定の条件下での依存オブジェクトの挙動を再現できるため，特定のシナリオに沿ったテストケースを作成することが可能となる．

様々なプログラミング言語において，モックを実現するためのフレームワークが開発されており，Java

```
1 // モックオブジェクトを宣言
2 Sheet sheetMock = mock(Sheet.class);
3
4 // モックオブジェクトの振る舞いを設定（スタブ）
5 // "sheet.getRow()が呼ばれると0を返す"
6 when(sheet.getRow()).thenReturn(0);
7
8 sheet.getRow(); // 0が返される
```

図 4　Mockito におけるモックの生成とスタブの設定

```
1 // 実オブジェクトをスパイとして宣言
2 Template templateSpy = spy(new Template(…));
3
4 // 実オブジェクトの振る舞いを上書き:
5 // "width()が呼ばれると3を返す"
6 doReturn(3).when(templateSpy).width();
7
8 templateSpy.width(); // 3が返される
```

図 5　Mockito におけるスパイの生成とスタブの設定

では Mockito[*1]や EasyMock[*2]のようなモックフレームワークが代表的である．ここでは，Mockito を使ったモック生成およびスタブの設定について説明する．図 4 は，Mockito を使って Sheet クラスのモックを作成し，getRow() に対するスタブを設定した例である．この例では，Mockito の mock() で Sheet クラスのモックを作成し，Mockito の when(...).thenReturn(...) 構文を用いて getRow() が常に 0 を返すようスタブを設定している．

Mockito では，モックの他にスパイを作成することが可能である．スパイとは，実際のオブジェクトを基に作成された特殊なモックであり，元のオブジェクトの動作を引き継ぎながら，特定のメソッドに対して選択的にスタブを設定できるという特徴を持つ．図 5 は，Mockito を使って Template クラスのスパイを作成し，width() に対するスタブを設定した例である．この例では，Mockito の spy() で Template クラスのスパイを作成し，doReturn(...).when(...) 構文を用いて width() メソッドが常に 3 を返すようスタブを設定している．一方で，Template クラスが持つ他のメソッドについては，元の Template オブジェクトの実装通りに実行される．

本研究では，テスト対象メソッドの分岐を網羅することを目的として，その分岐へ至る経路上で使用されるオブジェクトをモックに代替し，そのオブジェクトの動作を模倣するスタブを設定する．一方，テスト対象メソッドが定義されているクラスのインスタンスについてはモックではなくスパイを用い

[*1] https://site.mockito.org/
[*2] https://easymock.org/

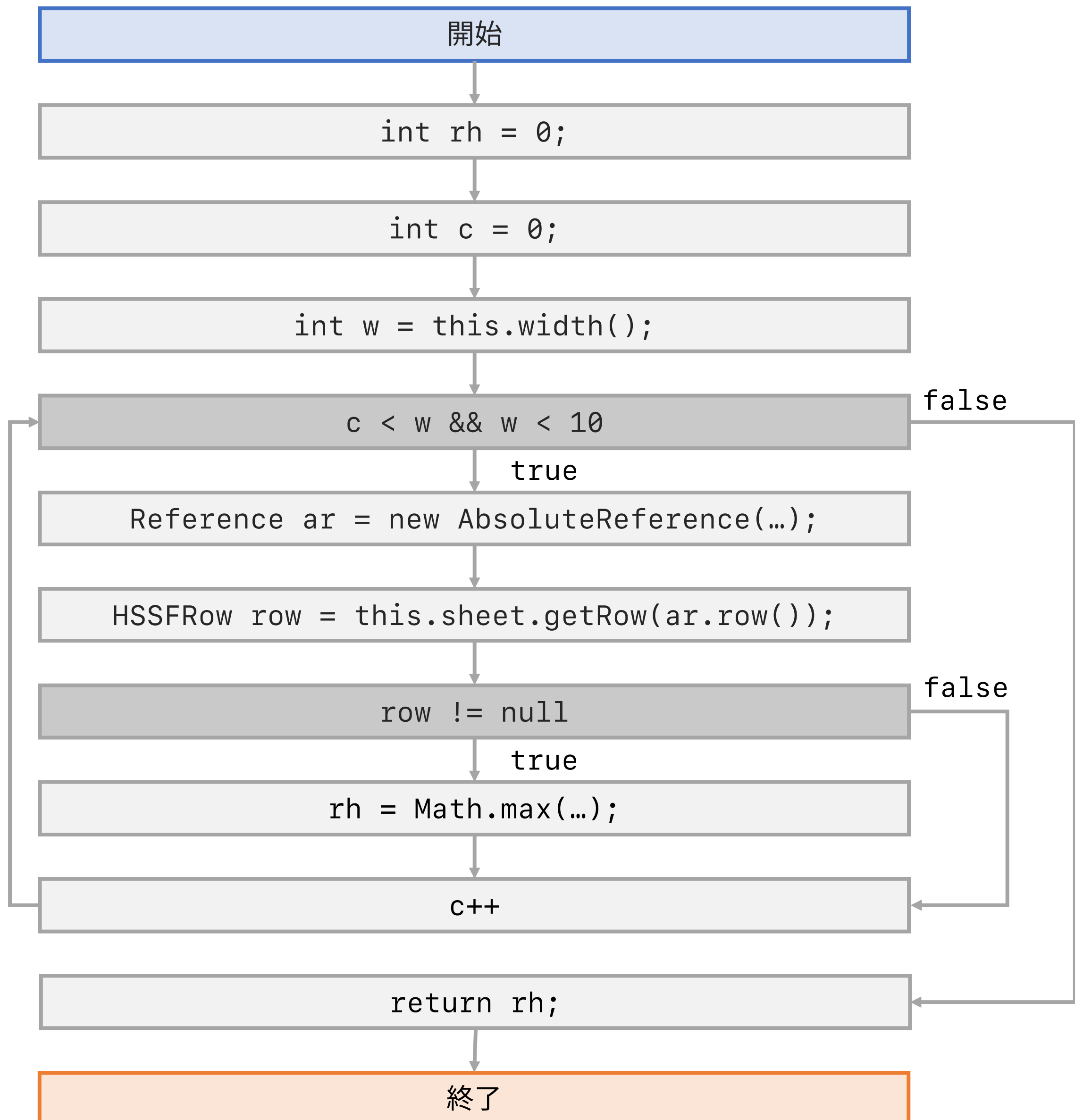

図 6　図 2 にのメソッドに対応する制御フローグラフ

る．テスト対象メソッドそのものをモックで代替すると，本来の処理が実行されなくなり，テストの意義が損なわれてしまうためである．スパイを使用することで，テスト対象メソッドの元の動作を保持しつつ，特定のメソッドやフィールドの動作に対して適宜スタブを設定する．これにより，テスト環境から不要な依存要素を排除しながら，テスト対象メソッドの動作を正しく検証できるようになる．

```
1 public void method(int x) {
2   int y = x + 1;
3   int z = y * 2;
4
5   if (z > 10) {
6     // ...
7   }
8 }
```

図 7　記号実行による条件分岐の解析対象メソッド

2.7　制御フローグラフによるプログラム解析

制御フローグラフとは，プログラムに含まれる命令の実行経路を表現したグラフである．一般には，途中に制御の分岐がない連続する命令列（基本ブロック）を頂点で表現し，基本ブロック同士の制御の流れを辺で表現する．制御フローグラフは，到達不能コードの検出といったプログラムの制御構造の解析に活用される．

また，制御フローグラフはプログラムに対するデータフロー解析にも使用される．制御フローグラフの各頂点に登場する変数や定数を追跡することで，プログラムにおけるデータの流れを特定することができる．データフロー解析はコンパイラによるプログラム最適化で応用される．例えば，変数の定義と使用の関係の特定による冗長な計算の除去 [25] に使用される．

本研究では，基本ブロックの代わりに文や条件式を頂点としたグラフを使用する．本研究の提案手法（3 節）では，個々の文や条件式を単位としてプログラムの解析を実行するためである．図 2 のメソッドに対する，個々の文や条件式を頂点とした制御フローグラフを 図 6 に示す．制御フローグラフには，メソッドの実行開始地点を表す開始頂点とメソッドの実行終了地点を表す終了頂点が含まれ，個々の文や条件式が実行順に対応するように辺が接続される．また，`c < w && w < 10` や `row != null` のような分岐を伴う頂点では，分岐先の頂点に遷移する複数の辺を持つ．

本研究では，テスト対象メソッドに対して制御フローグラフを構築し，網羅したい分岐に到達するための経路を特定する．また，経路に含まれる依存オブジェクトの動作を特定するために，制御フローグラフによるデータフロー解析を実施し，依存オブジェクトの定義と使用の関係を追跡する．

2.8　記号実行

記号実行 [9] とは，プログラムのある経路を実行するために必要な変数の値を導出する手法である．テスト対象プログラムに対するテストケースの入力値を自動生成する手段として使用されており，記号実行を活用した単体テスト自動生成ツールが多く存在する [4, 13, 14, 15, 16]．

記号実行では，入力として具体的な値の代わりに任意の値を表す記号を設定し，その記号を用いてプ

ログラムの動作を解析する．解析の過程で，プログラムに含まれる条件式を記号による数式として表現する．その後，得られた数式を一階述語論理式で表現し，SMT（Satisfiablity Modulo Theories）ソルバを用いて条件を満たす具体値を導出する．SMT ソルバとは，与えられた一階述語論理式に対し，それを真とする解が存在するかどうかを判定し，存在するならば一つの具体解を出力するプログラムである．

図 7 に示すメソッドにおける 5 行目の `if (z > 10)` という分岐に対し，記号実行を用いてこの条件を満たす引数の値を導出する方法を説明する．まず，メソッドの引数 `x` を具体的な値ではなく記号 `x` として与え，メソッドの実行経路を追跡する．経路の追跡はメソッドの終了地点に到達するまで実施し，実行の過程で登場する変数を，具体的な定数ではなく記号による式として表現する．具体的には，変数 `y` を `x + 1`，変数 `z` を `(x + 1) * 2` と表現する．この追跡によって，`z > 10` を満たすために必要な条件式は `(x + 1) * 2 > 10` と特定できる．次に，この式を `x` についての一階述語論理式として表現し，SMT ソルバを用いて条件を満たす具体値を導出する．この場合，例えば `x = 5` が解の具体値の一つとして得られる．

本研究では，記号実行における条件式の解析と具体値の導出手法を応用し，経路に含まれる分岐条件が常に経路通りに評価されるために必要な引数と代入文に与える値を決定する．

3　提案手法

本研究では，単体テスト自動生成技術の改善を目的として，テスト対象メソッドに対する制御フローグラフを利用した経路解析に基づき，モックを活用した単体テストを自動生成する手法を提案する．提案手法は，テスト対象メソッドに含まれる網羅すべき分岐（目標分岐）に対して，その分岐へ確実に到達するテストケースを生成する．これを実現するために，実行されるメソッドの経路を固定し，かつ経路に含まれるプログラム文が確実に実行されるテストケースを構成する．具体的には，以下の二つの条件を保証するテストケースを構成する．

条件 1　経路に含まれる分岐条件が常に経路通りに評価される．

条件 2　経路上の全ての命令が確実に実行される．すなわち，実行途中で例外が発生しない．

提案手法は以下の手順で構成される．

手順 1　目標分岐に到達する経路の特定

手順 2　経路上の条件式解析と具体値の導出

手順 3　経路上のオブジェクトに対する動作の特定

手順 4　テストケースの出力

手順 1 では，メソッドの制御フローグラフを探索し，目標分岐に到達する経路を特定する．手順 2 では，手順 1 で特定した経路に含まれる条件式を解析し，条件 1 を満たすために必要な引数や代入文に与える値を導出する．手順 3 では，手順 1 で特定した経路に含まれるオブジェクトに関連した操作を解析し，条件 2 を満たすために必要なオブジェクトの動作を特定する．手順 4 では，手順 1 から手順 3 までの結果をもとに，目標分岐に到達するためのテストケースを生成する．以下，各手順の詳細について述べる．

3.1　目標分岐に到達する経路の特定

テスト対象メソッドの制御フローグラフを探索し，メソッドの開始地点から目標分岐に至るまでの経路を特定する．制御フローグラフの開始頂点を始点として深さ優先探索を実行し，最初に見つかった経路を探索結果とする．経路に制御フローを分岐させる文が存在する場合，分岐条件の評価結果が `true` または `false` のどちらに対応するかを記録する．

図 2 の `getRowHeight()` に対し，`row != null` が成立する（`row` が `null` 以外を返す）分岐を目標分岐とした場合の経路探索を考える．`getRowHeight()` を制御フローグラフで表現すると，図 8 のグラフが得られる．このグラフに対して深さ優先探索を実行すると，目標分岐に到達する一つの経路として 図 8 における青色の経路が得られる．この経路には，`for` の反復継続条件 `c < w && w < 0` および

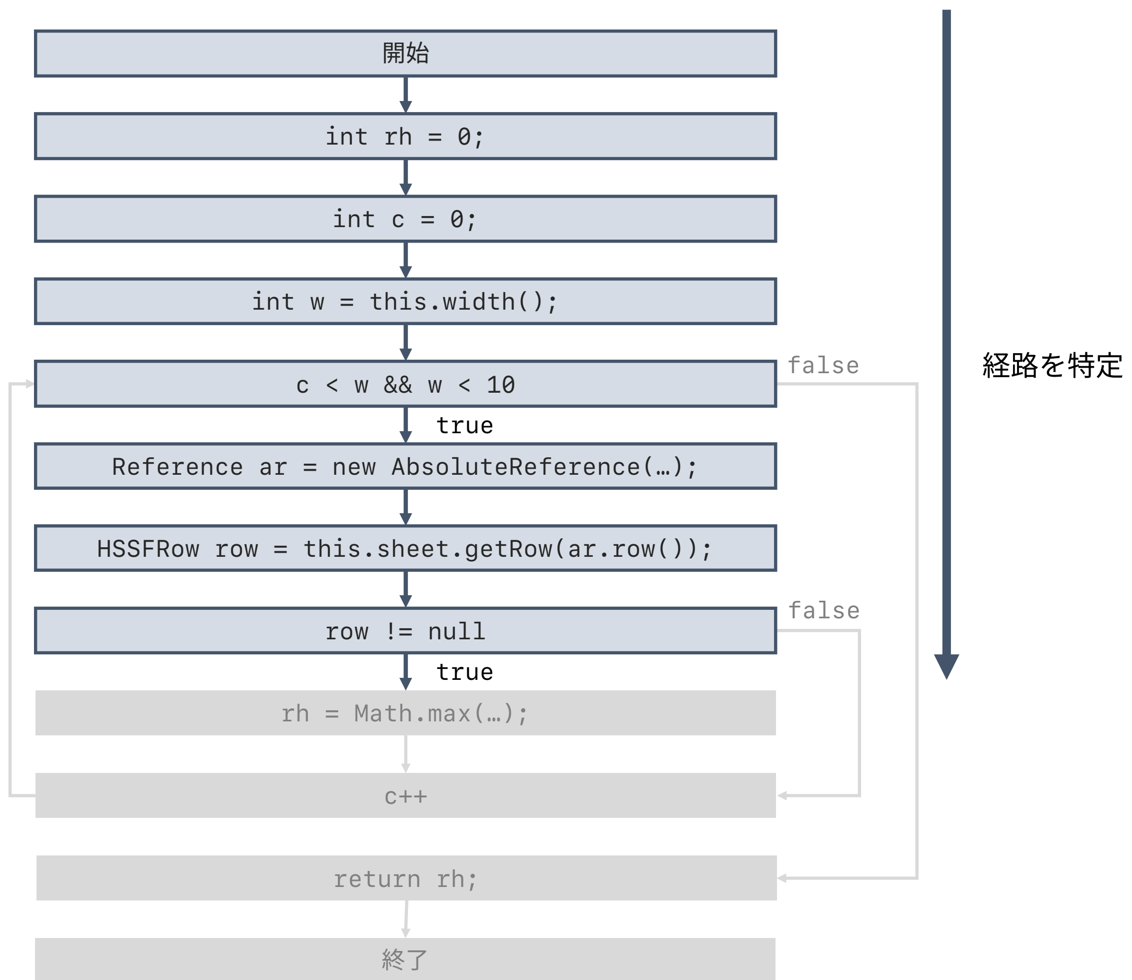

図 8　`row != null` が成立する分岐に到達する経路の特定

目標分岐 `row != null` の 2 つの条件式が含まれる．この経路においては，どちらの分岐条件も `true` と評価される．

表 1　プリミティブ型の値に対する基本条件*3

種類	説明	例
等価条件	2 値の値が一致/不一致	`a == b`, `a != b`
関係条件	2 値の値の大小関係	`a < b`, `a >= b`
算術演算条件	2 値の演算結果	`c == a + b`

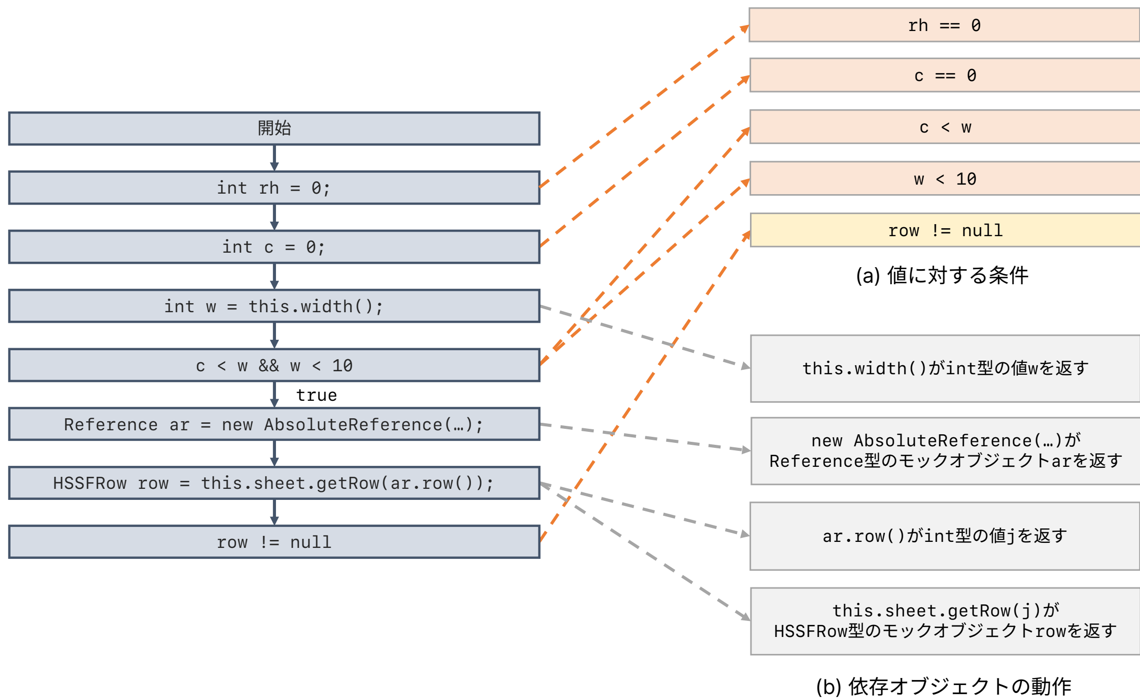

図 9　引数と代入文に与える値，およびオブジェクトの動作に要求される条件の特定

3.2　経路上の条件式解析と具体値の導出

3.1 節で特定した経路に含まれる条件式を探索し，条件 1 を満たすために必要な引数と代入文に与える値を決定する．探索にあたり，「基本条件」という概念を導入する．条件式が取りうる構文は複雑である場合が多い．そのため，条件式をそのままの形式で解析するのは困難である．そこで，本研究では基本条件と呼ぶ最小単位の式を定義し，条件式を基本条件の集合に変換した上で解析を実施する．本研究で定義した基本条件を 表 1 および 表 2 に示す．

表 2　参照型の値に対する基本条件[*4]

種類	説明	例	与える値
等価条件	2 変数の参照が一致	`a == b`	同一のモック
null 条件	変数の値が `null`	`a == null`	`null`
キャスト条件	変数の型がキャストされる	`(T) b`	`T` 型のモック

[*3] `a`，`b`，`c` はメソッド内の変数または定数を表す．
[*4] `a`，`b` はメソッド内の変数または定数，`T` は型を表す．

手順について説明する．まず，経路に含まれる条件式および変数宣言文を解析し，引数と代入文に与える値に要求される条件を基本条件の集合に変換する．次に，抽出した基本条件の集合を満たす具体的な値を決定する．プリミティブ型の値に対する基本条件に対しては，記号実行 [9] における条件式の解析と具体値の導出手法を応用して値を決定する．具体的には，プリミティブ型の値に対する基本条件を一階述語論理式に変換し，SMT ソルバを用いて基本条件の集合を満たす具体解を導出する．SMT ソルバが具体解を導出できなかった場合には，この経路に対するテスト生成をスキップする．基本条件を満たす入力が存在しないことを意味するためである．参照型の値に対する基本条件に対しては，表 2 に基づいて値を決定する．

図 8 の経路に対し，条件 1 を満たすために必要な引数や代入文に与える値を特定することを考える．この経路には，条件式として `c < w && w < 10` と `row != null` が含まれている．また，プリミティブ型に対する代入文として `int rh = 0;` と `int c = 0;` が含まれている．これらの式および代入文を基本条件の集合に変換すると，図 9 の (a) に示す基本条件を得る．プリミティブ型の値に対する基本条件については，SMT ソルバで具体解を導出すると，具体解の一つとして `rh = 0`，`c = 0`，`w = 1` が得られる．参照型の値に対する基本条件については，`row != null` より，`row` に対して与える値は，`null` ではないモックと決定される．

3.3　経路上のオブジェクトに対する動作の特定

3.1 節で特定した経路に含まれるオブジェクトに関連する操作（メソッド呼び出し，フィールド参照，インスタンス生成）を解析し，オブジェクトを代替するモックを宣言するコード，および，特定したオブジェクトの動作に対してスタブを設定するコードを生成する．

まず，呼び出し元オブジェクトに対応するモックが未生成の場合，そのオブジェクトに対応するモックを新たに宣言するコードを生成する．ただし，呼び出し元オブジェクトが `this` の場合，すなわち，テスト対象メソッドが属するクラスのインスタンスである場合については，2.6 節で述べた理由によりモックの代わりにスパイを使用する．`this` に対応するスパイを宣言するコードはテストケースを出力するタイミング（3.4 節）で生成する．そのため，この時点ではスパイを宣言するコードは生成しない．

次に，呼び出し元のオブジェクトに対応するモックやスパイに対し，そのオブジェクトに対する操作に対してスタブを設定するコードを生成する．返り値を持つ操作の場合，返り値が 3.2 節で導出した値に関連する場合にはその値を返し，関連しない場合には型に応じたデフォルト値を返す．

例えば，図 2 の 11 行目の `int w = this.width();` では，`Template` クラスの `width()` が呼び出されている．`width()` の呼び出し元オブジェクトは `this`，すなわち，テスト対象メソッドが属するクラスのインスタンスである．そこで，テスト対象メソッドが属するクラスのインスタンスに対応するスパイ `templateSpy` の `width()` 呼び出しに対し，`int` 型の変数 `w` を返すようにスタブを設定する．また，

```
1    int w = 1;
2    doReturn(w).when(templateSpy).width();
```

図 10　this に対応するスパイにスタブを設定するコード片

変数 w は経路上の条件式に登場しているため，3.2 節で導出した値（w = 1）を w の値として使用する．以上より，図 10 に示すコード片が生成される．

オブジェクトに関連する操作が代入文の右辺式として記述され，その結果として新たなローカル変数が宣言される場合，そのローカル変数が呼び出し元オブジェクトとして使用されるメソッド呼び出しやフィールド参照に対して再帰的にコードを生成する．例えば，図 2 の 14 行目の Reference ar = new AbsoluteRefrence(r, c); では，new AbsoluteRefrence(r, c) によって Reference 型変数 ar が宣言される．変数 ar は 15 行目の ar.row() で呼び出し元オブジェクトとして参照される．そのため，ar.row() に対してもスタブを設定するコードを出力する．

3.4　テストケースの出力

3.2 節および 3.3 節の結果をもとに，3.1 節で特定した経路を確実に実行するテストケースを JUnit のテストメソッドとして生成する．テストメソッドは以下に示す 4 つのコード片から構成される．

this に対応するスパイの宣言　this に対応するスパイを宣言する．テスト対象メソッドが属するクラスのコンストラクタ呼び出しに必要な引数として，参照型の場合はモックを，プリミティブ型の場合は型に応じたデフォルト値を使用する．

スパイに対するスタブの設定　宣言したスパイに対し，3.3 節で生成されたコード片を用いてスタブを設定する．

テスト対象メソッドに与える引数の生成　テスト対象メソッドに与える引数を宣言する．引数が 3.2 節で導出された値に関連する場合はその値を使用する．引数が 3.3 節で特定したオブジェクトの動作に関与する場合は，該当するコード片を配置する．いずれにも該当しない場合は，参照型の場合はモックを，プリミティブ型の場合は型に応じたデフォルト値を使用する．

テスト対象メソッドの呼び出し　生成した引数を使用してテスト対象メソッドを呼び出す．

図 2 の getRowHeight() に対し，目標分岐 row != null を満たすテストケースを生成した結果を 図 11 に示す．このテストケースでは，テスト対象クラス Template に対してスパイを生成し，width() や getRow() などのメソッドに対応するスタブを設定している．その後，テスト対象メソッドを呼び出すために必要な引数を宣言し，最後にそれらの引数を用いてテスト対象メソッドを呼び出す．

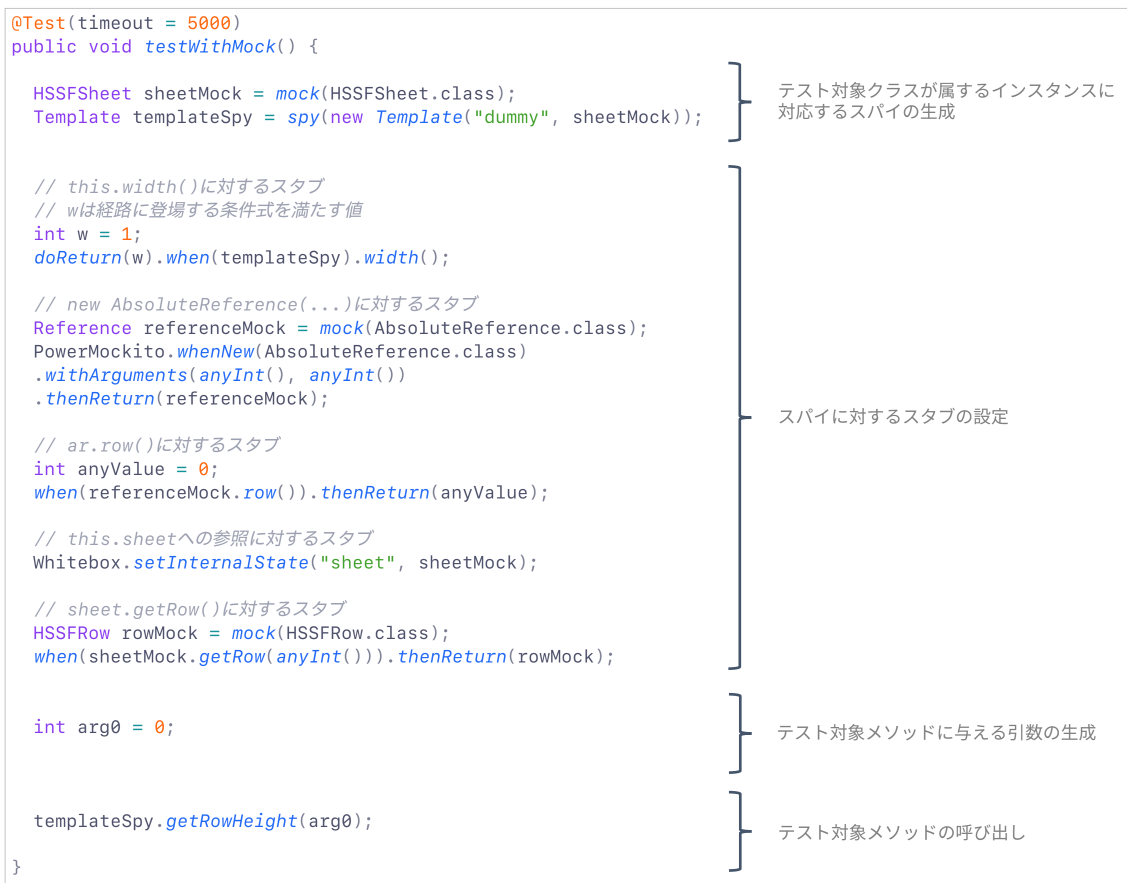

図 11　生成されるテストケース

4　評価実験

提案手法の有効性を評価するために，評価実験を実施した．本研究の目的は，EvoSuite では網羅できない分岐を補完し，より高い分岐網羅率を達成することである．そこで評価実験では，EvoSuite のみで生成されたテストケースをベースラインとし，そこに提案手法で生成したテストケースを組み合わせることで，EvoSuite 単独と比較した場合に分岐網羅率がどれだけ向上するかを検証する．

4.1　提案手法の実装

評価実験を実施するにあたり，本研究の提案手法を実現するプログラム MockTestGen を Java で実装した．評価実験で使用する Java プロジェクトが全て Java SE 6 で開発されているため，出力するテストケースのソースコードも Java SE 6 の言語仕様に準拠した形式で生成するよう実装した．

MockTestGen は提案手法を検証するための試作段階の実装である．実験では手法の基本的な有効性を確認することに重点を置いており，現時点では概念実証に特化している．そのため，`String` 型，配列型，`switch` 文を含む経路に対するテスト生成は現時点では対応していない．これらの要素については，より高度な解析手法や条件設定が必要となるため，将来的な拡張として取り組む予定である．

制御フローグラフに基づくプログラム解析には，SootUp[26] と呼ばれるバイトコード解析ライブラリを使用した．SootUp は，解析対象の Java クラスのバイトコードを Jimple[27] と呼ばれる中間表現に変換し，この Jimple を通じてプログラム解析を行うための多様な API を提供するライブラリである．

また，プリミティブ型の値を含む条件の導出（3.2 節）の実装のために，Z3[28] と呼ばれる SMT ソルバを使用した．Z3 は Microsoft によって開発されたオープンソースの SMT ソルバである．

さらに，モックやスパイを用いた依存オブジェクトの代替（3.3 節）の実装のために，モックフレームワーク Mockito および PowerMock [*5] を使用した．PowerMock は，`private` メソッドや静的メソッドといった，Mockito ではスタブできないメソッドに対するスタブを設定できる強力なモックフレームワークである．

4.2　評価方法

実験題材として，SF110 [22] データセットを使用した．SF110 は，EvoSuite の大規模評価のために開発されたデータセットで，110 個のオープンソース Java プロジェクトで構成されている．

あらかじめ，評価のベースラインとして SF110 の全プロジェクトに対し EvoSuite v1.0.6 を用いてテストスイートを生成した[*6]．EvoSuite の生成時パラメータは SF110 に同梱されている構成ファイルを

[*5] `https://github.com/powermock/powermock`

[*6] SF110 には，EvoSuite によって生成されたテストスイートが同梱されている．しかし，これらのテストスイートは古い

利用した．

次に，SF110 の各プロジェクトに含まれる全ての `public` クラスから MockTestGen によるテストケースの生成対象となる分岐を抽出した．まず，SF110 の各プロジェクトに含まれる全ての `public` クラスから，内部クラスでない具象クラスを抽出した．すなわち，抽象クラス，インターフェース，内部クラスを除外した．次に，抽出したクラスに含まれる全ての `public` メソッドの分岐を，生成対象の分岐として抽出した．

その後，抽出した生成対象の分岐に対し，MockTestGen を用いてテストケースを生成した．その結果，110 個中 10 個のプロジェクトで，MockTestGen によるテストスイートの生成とテストスイートのコンパイルに成功した．よって，これら 10 個のプロジェクトを評価対象とする．評価対象には 1,977 個の分岐が含まれており，そのうちテストケースの生成対象となった分岐は 1,676 個である．

次に，評価対象となる 10 個のプロジェクトに対し，EvoSuite が生成したテストスイートと MockTestGen が生成したテストスイートを実行し，網羅率計測ツールの JaCoCo*7を用いてテストスイートで網羅された分岐および分岐網羅率を計測した*8．

4.3 結果

表 3 に実験結果を示す．表 3 は，EvoSuite および MockTestGen で生成されたテストスイートにおける網羅された分岐数と，テストケースの生成対象となった全ての分岐，評価対象に含まれる全ての分岐をそれぞれ分母とした際の分岐網羅率である．

まず，MockTestGen 単体の結果を確認する．テストケースの生成対象となった 1,676 個の分岐のう

表 3　評価対象全体で網羅された分岐数と分岐網羅率

項目	網羅された分岐数	分岐網羅率	
		生成対象	全体
MockTestGen	378	22.55%	19.12%
EvoSuite	1,201	–	60.75%
MockTestGen + EvoSuite	1,297	–	65.60%
MockTestGen で新たに網羅された分岐	96	–	4.85%

バージョンの EvoSuite を使用して生成されているため，同梱されたデータセットを評価のベースラインとして使用するのは適切ではないと判断した．

*7 `https://www.eclemma.org/jacoco/`

*8 bpmail という名称のプロジェクトにおいて，プロジェクト内部で使われている依存関係ライブラリと MockTestGen で使われている依存関係ライブラリの一部が重複していた．そのため，bpmail については，MockTestGen の依存関係ライブラリを優先的に使用して計測を実施した．

ち，表 3 より，378 個の分岐が MockTestGen によって網羅されたことが分かる．これは生成対象の全分岐の 22.55% に相当する．

次に，EvoSuite のテストスイートと組み合わせた場合の結果を確認する．表 3 より，MockTestGen によって，EvoSuite では網羅できなかった分岐が新たに 96 個網羅され，評価対象全体の分岐に対する分岐網羅率が 4.85% 向上したことが分かる．この結果は，MockTestGen は EvoSuite が生成したテストスイートでは到達できなかった分岐を網羅していることを示している．

5 考察

提案手法の論理的な限界を明らかにすることを目的として，MockTestGen が生成したテストスイートで網羅されなかった分岐について，その原因を分析する．ただし，実装時に生成対象から除外したため網羅されなかった分岐や，MockTestGen のバグによってテストケースが生成されなかった分岐については取り扱わない．これらは提案手法に起因する本質的な限界ではないためである．

4.2 節で抽出した生成対象となる分岐のうち，MockTestGen が生成したテストスイートでは網羅されなかった分岐について，その分岐が網羅できなかった原因を目視調査した．メソッド内に網羅できなかった分岐が複数存在する場合，網羅できなかった最初の分岐のみを調査対象とした．

その結果，調査対象となる分岐 196 個のうち，生成対象から除外した分岐が 135 個，プログラムのバグが原因の分岐が 58 個であった．残り 3 個について原因を分析した結果，以下のように分類された．

- テスト対象クラスのインスタンス化に失敗（1 個）
- テスト対象メソッドの外部からの操作では経路通りに実行できない（2 個）

以下，これら 2 つの事例について説明する．

5.1 テスト対象クラスのインスタンス化に失敗

テスト対象メソッドが属するクラスのインスタンスのスパイを生成する際，外部オブジェクトや外部システムへの依存が原因でインスタンス生成に失敗する場合，提案手法では有効なテストケースを生成できない．スパイを生成するには，スパイされるオブジェクトの実インスタンスの生成が必要であるため，クラスのインスタンス化の際に実行されるインスタンスフィールドの初期化やコンストラクタの実行をモックで代替することはできない．

図 12 はその実例である．これは imsmart プロジェクトに含まれる `RequestParamExample` クラスのコードの抜粋である．同クラスの `doGet()` に含まれる `firstName != null` という分岐を網羅するテストケースを MockTestGen で生成すると，図 13 に示すテストケースが得られる．

このテストケースでは，5 行目でスパイを宣言する際に `RequestParamExample` クラスのインスタンス化を実行している．このとき，`RequestParamExample` クラスは `rb` というインスタンスフィールドを持ち，このフィールドは 図 12 の 4 行目に示す `ResourceBundle rb = ResourceBundle.getBundle("LocalStrings");` によって初期化される．しかし，`ResourceBundle.getBundle()` はシステムのロケールに依存するため，このテストケースは実行時に失敗する．

このような場合に提案手法を適用可能にする方法の一つとして，テスト対象メソッドが属するクラス

```
public class RequestParamExample extends HttpServlet {

  // 初期化を伴ったインスタンスフィールドの宣言
  ResourceBundle rb = ResourceBundle.getBundle("LocalStrings");

  // ...

  public void doGet(
    HttpServletRequest request,
    HttpServletResponse response
  )
      throws IOException, ServletException {

    // ...

    // 目標分岐
    if (firstName != null || lastName != null) { ... }

    // ...
  }
}
```

図 12　テスト対象クラスのインスタンス化に失敗した例

```
public class RequestParamExampleMockedTest {
  @Test(timeout = 5000)
  public void test0() throws Throwable {
    // スパイの宣言
    RequestParamExample cut = PowerMockito.spy(new RequestParamExample());

    // 依存オブジェクトを代替するモックの生成とスタブの設定
    // ...

    // テスト対象メソッドの呼び出し
    cut.doGet(request, response);
  }

  // ...
}
```

図 13　doGet() から生成されたテストケースの例

に対し，依存性注入を可能にするリファクタリングを実施することが考えられる．依存性注入とは，あるクラスの依存オブジェクトを，そのクラスの呼び出し元から提供する方法である．この手法により，クラスの設計が依存オブジェクトの詳細な実装に依存しなくなる．その結果，モックを用いた依存関係の代替が容易となりテスト容易性が向上する．

図 14 は，rb の値をコンストラクタの引数として提供できるように RequestParamExample クラス

```java
public class RequestParamExample extends HttpServlet {

  // 初期化を伴わないインスタンスフィールドの宣言
  ResourceBundle rb;

  // コンストラクタで依存オブジェクトrbを注入
  public RequestParamExample(ResourceBundle rb) {
    this.rb = rb;
  }

  public void doGet(
    HttpServletRequest request,
    HttpServletResponse response
  )
      throws IOException, ServletException {

    // ...

    // 目標分岐
    if (firstName != null || lastName != null) { ... }

    // ...
  }
}
```

図 14　提案手法が適用可能となるリファクタリングを実施した後の RequestParamExample クラス

をリファクタリングした例である．リファクタリング後のクラスに提案手法を適用すると，rb にモックを代入する実行可能なテストケースが生成される．

5.2　テスト対象メソッドの外部からの操作では経路通りに実行できない

提案手法では，プログラム解析によって目標分岐までの経路を特定し，条件を満たす具体値を導出する．しかし，分岐条件の成立に必要な操作がプログラムの制御構造に埋め込まれており，解析による外部からの操作では経路通りに実行できないメソッドが存在する．

図 15 はその実例である．これは pbmail プロジェクトに含まれる MenuItemList クラスのコードの抜粋である．同クラスの checkConsistency() に含まれる result == true を偽とする分岐を網羅するテストケースを MockTestGen で生成すると，図 16 に示すテストケースが得られる．

図 16 のメソッドでは，8 行目から 11 行目において i < children.size() を true に設定するスタブを設定している．よって，このテストケースを実行すると result == true の条件分岐まで到達する．一方で，図 17 の run() では，3 行目の boolean result = true; により result は常に true で初期化される．そのため，result == true を偽とする分岐の網羅は実現できない．

一方で，5 行目から 8 行目までの for ループでの処理には，result に対し false を代入する命令が

```java
public class MenuItemList {
  public boolean checkConsistency() {
    boolean result = true;
    for (int i = 0; i < children.size() && result == true; i++) {
      if (!(children.get(i) instanceof MenuItemList)) result = false;
      MenuItemList childList = (MenuItemList) children.get(i);
      if (!childList.getMenuItem().equals(parentMenuItem)) result = false;
      result = childList.checkConsistency();
    }

    return result;
  }

  // ...
}
```

図 15　テスト対象メソッドの外部からの操作では経路通りに実行できない例

```java
@Test(timeout = 5000)
public void test5() throws Throwable {
  // スパイの宣言
  IMenuItem arg0 = mock(IMenuItem.class);
  MenuItemList cut = PowerMockito.spy(new MenuItemList(arg0));

  // 依存オブジェクトを代替するモックの生成とスタブの設定
  Vector $stack4 = mock(Vector.class);
  Whitebox.setInternalState(cut, "children", $stack4);
  int $stack5 = 1;
  when($stack4.size()).thenReturn($stack5);

  // テスト対象メソッドの呼び出し
  cut.checkConsistency();
}
```

図 16　checkConsistency() から生成されたテストケースの例

存在する．よって，for ループ内の経路を一巡するような経路に対しテストケースを生成した場合，このような分岐においても網羅可能なテストケースを生成することができると考えられる．これは，提案手法における経路探索アルゴリズムを改良することで実現可能であり，今後の課題である．

一方で，マルチスレッド環境下で実行されるクラスのように，プログラム構造上経路が存在しない場合も存在する．図 17 はその実例であり，nekomud プロジェクトに含まれる SelectionThread クラスのコードの抜粋である．このクラスの run() に含まれる this.finished == false を偽とするテストケースを生成する場合を考える．this.finished を true に変更する操作は stop() 内の this.finished = true;（15 行目）でのみ定義されている．しかし，stop() は run() 内部で呼び出

```java
class SelectionThread implements Runnable {

  private boolean finished;

  public void run() {
    this.finished = false;

    // 目標分岐
    while (this.finished == false) {
      // ...
    }
  }

  public void stop() {
    this.finished = true;
    // ...
  }
}
```

図 17　プログラム構造上網羅不可能な例

されることがない．SelectionThread クラスはマルチスレッド環境下で実行されるクラスだからである．そのため，run() や stop() はこのスレッドの実行元となるスレッドから実行される．すなわち，run() の制御フローにおいて this.finished を true にする経路は存在しない．以上より，この分岐を網羅するテストケースを生成することは不可能である．

6 妥当性の脅威

提案手法の実装および評価実験で使用した評価対象は，いずれも Java SE 6 の言語仕様に準拠している．そのため，異なる Java バージョンを対象とした場合，異なる結果が得られる可能性がある．

提案手法の実装には，モックフレームワークとして Mockito や PowerMock を使用した．モックで代替可能なオブジェクトの範囲は，使用するフレームワークの仕様に依存する．例えば，Mockito では `final` クラスに対するモックを生成することや，`java.lang.Object#equals()` や `java.lang.Object#hashCode()` に対するスタブを設定することはできない．そのため，他のモックフレームワークを用いた場合，モックの代替可能なオブジェクトの種類が変化し，評価実験において異なる結果が得られる可能性がある．

評価実験のベースラインとして使用した EvoSuite は，遺伝的アルゴリズムで探索的にテストスイートを生成する．この性質上，出力されるテストスイートにランダム性が含まれる．そのため，EvoSuite の実行構成や実行時のシード値によっては，異なる結果が得られる可能性がある．

7　おわりに

本研究では，テスト対象メソッドに対する経路解析に基づき，モックを活用した単体テストを自動生成する手法を提案した．提案手法では，メソッドに含まれる分岐ごとに到達経路を特定し，その経路に対する解析を実行する．この解析により，経路を確実に実行するために必要な値や依存オブジェクトの動作を抽出する．次に，抽出した値や依存オブジェクトの動作を元に依存オブジェクトを代替するモックを構築し，分岐に到達可能なテストケースを生成する．

EvoSuite では網羅できない分岐が提案手法によってどれだけ網羅できるか検証するため，提案手法を実現するプログラム MockTestGen を実装し，1,977 個の分岐を評価対象とする評価実験を実施した．具体的には，EvoSuite が生成したテストスイートに対し，提案手法が生成したテストケースを追加することで，新たに分岐がどれだけ網羅されるか検証した．その結果，分岐が新たに 96 個網羅され，分岐網羅率が 4.85% 向上した．

今後の課題として，MockTestGen のプログラム改良が挙げられる．MockTestGen は本稿執筆時点では試作段階の実装である．そのため，`String` 型，配列型，`switch` 文を含む経路に対するテスト生成には対応していない．また，MockTestGen にはバグが含まれており，一部のケースでテストが生成されない問題が確認された．そこで，MockTestGen に対して機能追加やバグ修正を行い，生成可能な経路の種類を増やすことを目指す．

また，提案手法で生成したテストケースがテスト対象プログラムのバグ検出能力に与える影響を調査することも今後の課題である．網羅率は，テスト対象プログラムの内部構造に基づく指標であり，テストスイートの網羅率が高いことが，必ずしも高いバグ検出能力を示すわけではない．さらに，モックを用いたテストケースは偽陽性を引き起こす可能性がある [29]．すなわち，テスト対象プログラムにバグが存在し，本来はテストに失敗すべき場合であっても，モックに設定されたスタブの動作によってはテストが成功してしまう場合がある．提案手法で生成されたモックには，実際のオブジェクトの動作と異なる動作がスタブとして設定される可能性があり，これが偽陽性の原因となることが考えられる．そこで，ミューテーション解析 [30] 等を活用し，提案手法で生成したテストケースのバグ検出能力や偽陽性の度合いを検証する予定である．

謝辞

本研究を進めるにあたり，多くの方々から多大なるご支援とご協力を賜りました．ここに心より感謝の意を表します．

楠本真二教授には，輪講や中間発表において的確な助言やご意見をいただきました．また，時折お菓子やドリンクを差し入れてくださり，研究を進めていく上での励みとなりました．

ソフトウェア工学講座の肥後芳樹教授には，学部 4 年次より 3 年間にわたり，継続的かつ丁寧なご指導を賜りました．毎週の研究ミーティングでは多くの貴重なアドバイスをいただき，研究を進める上での指針となりました．特に，研究が行き詰まった際にはお忙しい中個別ミーティングを開いてくださり，親身にご助言をいただきました．また，研究会や国際会議への投稿に挑戦する機会を与えてくださり，論文執筆の過程では細やかな添削とご指導を通じて，多くのことを学ぶことができました．特に英語論文の執筆に際しては，何度も対面でご指導をいただき，大変貴重な経験となったと感じております．研究会や国際会議での発表経験は，今後の人生において人前での発表やコミュニケーションを行う上での大きな糧となると感じております．

柗本真佑准教授には，輪講や中間発表の際に貴重な助言やご意見をいただくとともに，研究の過程で直面する課題への向き合い方や，研究を超えて作業や仕事そのものに取り組む姿勢について多くの学びを得ることができました．これらを通じて培った考え方は，今後の研究活動や問題解決の場面においても非常に役立つものとなると感じております．

事務補佐員の橋本美砂子様には，日々の研究活動を支えていただきました．毎朝のコーヒーマシンの準備や，国内外の出張における各種手続きのご尽力など，そのお力添えなしには研究を円滑に進めることはできませんでした．

研究室の学生には，日々の何気ない会話を通じて支え合い，時には一緒に食事や旅行を楽しむなど，研究の枠を超えた交流が，本研究を進める上での大きな活力となりました．

家族には，日々の励ましと金銭的な支援を通じて，多大なる支えをいただきました．家族の存在があったからこそ，研究に専念することができました．

最後に改めて，研究を支えてくださった方々に深く感謝申し上げます．

参考文献

[1] McMinn, P.: Search-Based Software Test Data Generation: A Survey, *Journal on Software Testing, Verification and Reliability*, Vol. 14, No. 2, pp. 105–156 (2004).

[2] McMinn, P.: Search-Based Software Testing: Past, Present and Future , in *Proceedings of International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163 (2011).

[3] Reeves, C. R. ed.: *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc. (1993).

[4] Fraser, G. and Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, in *Proceedings of Foundations of Software Engineering*, pp. 416–419 (2011).

[5] Srinivas, M. and Patnaik, L.: Genetic Algorithms: A Survey, *Journal on Computer*, Vol. 27, No. 6, pp. 17–26 (1994).

[6] Albunian, N., Fraser, G. and Sudholt, D.: Causes and Effects of Fitness Landscapes in Unit Test Generation, in *Proceedings of Genetic and Evolutionary Computation Conference*, pp. 1204–1212 (2020).

[7] Aleti, A., Moser, I. and Grunske, L.: Analysing the Fitness Landscape of Search-Based Software Testing Problems, *Journal on Automated Software Engineering*, Vol. 24, No. 3, pp. 603–621 (2017).

[8] Lin, Y., Ong, Y. S., Sun, J., Fraser, G. and Dong, J. S.: Graph-Based Seed Object Synthesis for Search-Based Unit Testing, in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1068–1080 (2021).

[9] King, J. C.: Symbolic execution and program testing, *Journal on Communications of the ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[10] Ammann, P. and Offutt, J.: *Introduction to software testing*, Cambridge University Press (2017).

[11] Pacheco, C. and Ernst, M. D.: Randoop: Feedback-Directed Random Testing for Java, in *Proceedings of Conference on Object-oriented Programming Systems and Applications Companion*, pp. 815–816 (2007).

[12] Csallner, C. and Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java, *Journal on Software: Practice and Experience*, Vol. 34, No. 11, pp. 1025–1050 (2004).

[13] Sen, K., Marinov, D. and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, *Journal*

on SIGSOFT Software Engineering Notes, Vol. 30, No. 5, pp. 263–272 (2005).

[14] Braione, P., Denaro, G., Mattavelli, A. and Pezz è, M.: SUSHI: A Test Generator for Programs with Complex Structured Inputs, in *Proceedings of International Conference on Software Engineering: Companion Proceeedings*, pp. 21–24 (2018).

[15] Tillmann, N. and de Halleux, J.: Pex–White Box Test Generation for .NET, in *Proceedings of Tests and Proofs*, pp. 134–153 (2008).

[16] Godefroid, P., Klarlund, N. and Sen, K.: DART: Directed Automated Random Testing, in *Proceedings of Conference on Programming Language Design and Implementation*, pp. 213–223 (2005).

[17] Panichella, A., Kifetew, F. M. and Tonella, P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets, *IEEE Transactions on Software Engineering*, Vol. 44, No. 2, pp. 122–158 (2018).

[18] Harman, M. and McMinn, P.: A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search, *IEEE Transactions on Software Engineering*, Vol. 36, No. 2, pp. 226–247 (2010).

[19] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A. and Roper, M.: Testability Transformation, *IEEE Transactions on Software Engineering*, Vol. 30, No. 1, pp. 3–16 (2004).

[20] Vogl, S., Schweikl, S., Fraser, G., Arcuri, A., Campos, J. and Panichella, A.: Evosuite at the SBST 2021 Tool Competition, in *Proceedings of International Workshop on Search-Based Software Testing*, pp. 28–29 (2021).

[21] Watanabe, R., Higo, Y. and Kusumoto, S.: Impacts of Program Structures on Code Coverage of Generated Test Suites, in *Proceedings of the International Conference on Product-Focused Software Process Improvement*, pp. 355–362 (2023).

[22] Fraser, G. and Arcuri, A.: A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite, *ACM Transactions on Software Engineering and Methodology*, Vol. 24, No. 2, pp. 1–42 (2014).

[23] Results for version 1.0.6, `https://www.evosuite.org/files/1_0_6/results.html`(Accessed at 2025-01-09).

[24] Mostafa, S. and Wang, X.: An Empirical Study on the Usage of Mocking Frameworks in Software Testing, in *Proceedings of International Conference on Quality Software*, pp. 127–132 (2014).

[25] Kennedy, K.: Use-Definition Chains with Applications, *Journal on Computer Languages*, Vol. 3, No. 3, pp. 163–179 (1978).

[26] Karakaya, K., Schott, S., Klauke, J., Bodden, E., Schmidt, M., Luo, L. and He, D.: SootUp: A Redesign of the Soot Static Analysis Framework, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 229–247 (2024).

[27] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot: A Java Bytecode Optimization Framework, in *Proceedings of CASCON First Decade High Impact Papers*, pp. 214–224 (2010).

[28] de Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340 (2008).

[29] Arcuri, A., Fraser, G. and Just, R.: Private API Access and Functional Mocking in Automated Unit Test Generation, in *Proceedings of International Conference on Software Testing, Verification and Validation*, pp. 126–137 (2017).

[30] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R. and Fraser, G.: Are Mutants a Valid Substitute for Real Faults in Software Testing?, in *Proceedings of International Symposium on Foundations of Software Engineering*, pp. 654–665 (2014).