卒 業 研 究 報 告 書

題　目　　テストケース生成ツールを用いた
自動バグ限局ツールの試作

指導教員　　水野　修　教授

崔　恩瀞　助教

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　17122047

氏　　名　　中森　陸斗

令和3年2月12日提出

テストケース生成ツールを用いた自動バグ限局ツールの試作

令和 3 年 2 月 12 日　　17122047　中森　陸斗

概　要

ソフトウェア開発において，デバッグ作業によるソースコードの修正は欠かせない工程である．しかし，ソースコードのバグ箇所を人力で探す作業は開発者にとって大きな負担となるため，バグ箇所の特定は非常に大きなコストとなる．そこで，ソフトウェアの不具合箇所を特定する手法である Spectrum-based Fault Localization (SBFL) と，ソースコードから自動でテストケースを生成するツールである EvoSuite を使用して，Java ソースコードからバグの位置を特定するツールを制作した．また，本ツールのバグ特定における有効性を確かめるため，著者が作成した Java ソースコードにおいてツールを使用し，バグ箇所をどの程度特定できるかを評価した．評価の結果，バグ箇所の特定の精度はバグの種類及びバグの存在するメソッドを呼び出すまでの命令行の長さに影響されることがわかった．

目　次

1. 緒言

デバッグとは，想定通り動作しないソフトウェアを修正する作業である．ソフトウェア開発において，デバッグはソースコードの品質を高めるためによく行われる作業の 1 つである．しかし，デバッグ作業は人力で行うとソフトウェア開発の工程における大部分を占めることとなり，非常にコストの掛かる作業となる [1].

開発者がソフトウェアのデバッグを行うとき，IDE に搭載されているデバッガを用いる方法がある．これは，開発者が自身でソースコード上にブレークポイントを決め，処理が一時中断された時点での変数の状態などを確認し，その内容に不具合が発生しているかを検証する方法である．この方法では，不具合の箇所について全く前情報がない場合，その検証にはソースコード上に網羅的にブレークポイントを設定する必要がある．また一時停止するたびに変数の内容を確認する必要があるため，非常に大きな時間が掛かる．そのため，デバッグ作業を自動的に行うツールの開発は，ソフトウェア開発全体のコストの削減に大きく寄与するものとなる．

ソースコードのデバッグ作業は主に 2 つの作業に分類でき，その内容は以下の通りである．

- ソースコード上のバグ箇所の特定 (バグ限局)
- ソースコードのバグの修正

この 2 つの作業はそれぞれ異なる難しさが存在するため，双方ともに大きなコストが掛かる．本研究においては前者に着目し，ソースコード上のバグ箇所の特定を自動的に行うツールを作成することで，ソースコード上のバグ箇所の特定に掛かるコストを削減することを目的とする．

現在，バグが潜在するモジュールやファイルを特定する手法は数多く存在する．バグレポートなどのバグ情報を使用した IR 手法 [2] や，Git などでコミットされたバージョン情報を使用した手法 [3] などが例として挙げられる．しかし，それらの手法は開発プロジェクトが十分に成熟しており，様々なバグレポートや十分なコミット数が存在するプロジェクトでのみ使用できるものである．そのため，そもそも Git などでバージョン管理されておらず，バグレポートを多人数から得られないような小規模なプロジェクトでは使用できないという問題がある．

本研究では，ソースコードのみを入力として使用できるような個人開発レベルのプロジェクトを対象に，自動的にバグ限局を行うことができるツールの制作を行った．バグ限局とは，ソースコードのどの部分にバグが存在するかを推定する方法である．ツールを具体的に説明すると，Java プロジェクト管理ツール Maven で管理された Java のプロジェクトを対象として，テストケース生成ツールで生成されたテストの結果を基に Fault Localization の手法を用いてバグ限局を行うソフトウェアである．このツールはソースコード上の各行の疑惑値を計算し，開発者にソースコードのどの部分にバグが存在するかをわかりやすく伝える．疑惑値とは，その行にバグが含まれる可能性を数値化したものである．

次に，制作したツールがどのようなソースコードにおいてバグ限局が可能であるかを検証するために，2 種類の Java ソースコードで制作したツールを適用した．その結果，バグ限局の有効性はソースコードによって異なることがわかった．

本報告書では，2 章でツールの制作に使用した諸技術についての説明を行い，3 章でツールの詳細な動作を述べる．次に 4 章で著者が制作した Java ソースコードに，本研究で制作したツールを使用した際の疑惑値の結果を示し，5 章でそれらの結果からツールの有効性の詳細と今後の課題について考察する．

2. 関連研究

2.1 Spectrum-based Fault Localization

Fault Localization とは，ソフトウェア上において不具合が存在する箇所を推定する手法である．本研究で制作するツールでは，ソースコード上のバグ箇所を特定する方法として Spectrum-based Fault Localization (SBFL) 手法を用いる．SBFL とはソースコードの動的解析手法の一種であり，あるテストケースを実行した際に成功したテストで実行された部分はバグの存在する可能性が低く，逆に失敗したテストで実行された部分にバグが存在する可能性が高いというアイデアの元で，不具合の位置をある程度まで特定する手法である．

具体的な方法を次に述べる．まずソースコードをテストするために作成されたテストケースを実行し，テストケース毎に実行されたソースコードの行を記録する．次に，記録されたテスト毎の実行位置データから成功ケースで実行された行と失敗ケースで実行された行を集計し，各行にバグの存在する可能性を算出する．このとき，バグの存在する可能性を算出する方法として，ソースコードの各行に対して，バグの存在する可能性を数値化した「疑惑値」という値を計算する．ソースコードの行の疑惑値が高いほど，その行にバグが存在する可能性が高いと言える．本研究では，疑惑値の計算方法として Ochiai の計算式 [4] を利用する．ソースコードの行 s の疑惑値 $suspicious(s)$ の計算式は式 2.1 の通りである．

$$suspicious(s) = \frac{Fail(s)}{\sqrt{TotalFail \times (Fail(s) + Pass(s))}} \tag{2.1}$$

ここで，$TotalFail$ は全テストケースにおける失敗したテストの数，$Pass(s)$ および $Fail(s)$ はそれぞれ s を通過した成功テストの数と失敗テストの数である．Ochiai の計算式はその他の数多く存在する疑惑値の計算方法と比較して，テストケースの品質及び個数の上下が疑惑値の精度に影響しにくいことが知られている [5]．

2.2 テストケース生成ツール

SBFL によって疑惑値を計算するためには，実行可能なテストケースが必要となる．大規模なプロジェクトにおいては，テストケースの実行を通してソースコード

にバグが存在するかを検証するため，テストケースを作成することが一般的である．しかし，本研究の対象である小規模なプロジェクトでは，開発者の人数が小数であることなどの理由により，十分な品質のテストケースを制作するコストは開発全体の工程において大きな割合を占めることになる．したがって，本研究の対象となるプロジェクトには実行可能なテストケースが事前に存在していないと仮定する．そのため，テストケースを自動的に生成するための手段が必要となる．

そこで，本研究ではテストケースの生成に EvoSuite[6] を使用する．EvoSuite とは，遺伝的アルゴリズムを用いた手法によって，Java で作成されたクラスのユニットテストを自動的に生成するツールである．EvoSuite の基本動作として，デバッグ対象の Java クラスファイルを 1 つ入力すると，テストケースのステートメントカバレッジをなるべく 100%に近くなるようにテストケースを生成する．その後，ステートメントカバレッジの向上に余分なテストケースを除外することによって，ステートメントカバレッジを保ちつつテストケースの個数が最小になるようなテストケースの集合を生成する [7].

3. 提案手法

本研究では，小規模なプロジェクトにおいてもバグ限局を可能にするため，ソースコードのみを入力としたツールを制作することを提案する．そのため，制作するツールは先に述べたSBFL手法とEvoSuiteを組み合わせたものとなっている．ツールの動作は以下の通りである．EvoSuiteで生成したテストケースに対してテスト毎の実行したソースコードの行を記録し，行ごとの疑惑値を計算する．その後，ソースコードの各行の疑惑値を可視化してまとめたHTMLファイルを生成する．なお，このツールは動的解析によるバグ箇所の特定を行うため，プロジェクトはその全体がビルド可能であることが前提となる．

本研究で作成したツールは2つのフェーズに分かれて動作する．以下に，この2つのフェーズの詳しい説明を通して，ツール全体のデータ及び処理の流れを説明する．また，ツール全体の処理の流れをまとめた概略図を図3.5に記載する．

3.1 フェーズ1：テストケースの生成

3.1.1 ステップ1-1：EvoSuiteでのテストケースの生成

フェーズ1では，ツールにソースコードを入力し，テストケースを生成する．まずMavenによってプロジェクト全体をコンパイルし，全ソースコードを一度クラスファイルへと変換する．次に，変換したクラスファイルをEvoSuiteに入力し，対象のソースコードに対応したユニットテストを生成する．

3.2 フェーズ2：テストの実行と疑惑値の計算

3.2.1 ステップ2-1：テスト毎の期待値の記述

フェーズ1で生成されたテストは，テスト結果が成功または失敗であるかの判別が不可能である．これは，EvoSuiteが出力するテストに含まれる判定文は，実際にテストコードを実行した際の返り値，または変数の状態を元にして生成されるからである．例えば，整数を返すメソッド`foo()`をテストするとき，その返り値が10であることが期待されている場合を考える．ここでもし，`foo()`にバグが含まれて

おり，その影響で返り値として 11 が返されてしまう場合，EvoSuite の出力するテストは「メソッド `foo()` は 11 という値を返すか？」という内容となる．このため，EvoSuite が出力したテストをそのまま実行してしまうと，すべてのテストが成功となり，SBFL 手法による疑惑値の計算ができなくなる．

そこで，各テストが成功であるか失敗であるかの判別方法として，各テストでインスタンス化されたオブジェクトの最終状態の期待値をユーザが入力する方法を採用する．具体的には，EvoSuite が出力したテストケースをユーザが読み，各テストにおいてインスタンス化されたオブジェクトの各属性がテストコードを終了した際にどのような値を取るべきかを JSON 形式でファイルに記述する．例えば，あるクラス `BarClass` を含む Java ソースコードをデバッグの対象とする．クラス `BarClass` の定義を図 3.1 に記載する．`BarClass` は `int` 型の属性 `a` を持っているものとする．このとき，図 3.2 のようなテストケースが生成されたとすると，`BarClass` のインスタンス `barClass0` において，テスト番号 0 では属性 `a` の値が 10，テスト番号 1 では属性 `a` の値が 20 となっていることが期待されている場合，JSON ファイルの中身は図 3.3 のようなものとなる．今回の場合では図 3.2 のテストケースを実行すると `a` の値はテスト番号 0 では 10 となり，テスト番号 1 では 20 となることがわかるため，図 3.3 の JSON ファイルに記述された期待値通りの値となるため，全てのテストケースは成功と判定される．

3.2.2 ステップ 2-2：ソースコードに通過命令を出力するコードを追加

デバッグ対象のソースコードをコンパイルし，テストケースと共に実行することによって対象のコードに含まれるクラス及びメソッド内の命令が実行される．しかし，そのまま実行するのみでは，どのテストでソースコード内のどの命令が実行されたかを記録することが出来ない．そこで，このステップではソースコードの各命令の直前に，命令がどの行番号に位置するかを標準出力に対して出力するコードを自動的に追加する．

3.2.3 ステップ 2-3：テストケースの実行

EvoSuite によって生成されたテストケースを実行する．このとき，ステップ 2-2 で追加した処理によって出力される行番号を標準出力経由で取得し，各テストがどの

```
1  public class BarClass{
2    private int a;
3
4    public void method1(){
5      a = 10;
6    }
7
8    public void method2(){
9      a = 20;
10   }
11 }
```

図 **3.1**　クラス **BarClass** の定義

```
1  @Test
2  public void test0() throws Throwable{
3    BarClass barClass0 = new BarClass();
4    barClass0.method1();
5  }
6
7  @Test
8  public void test1() throws Throwable{
9    BarClass barClass0 = new BarClass();
10   barClass0.method2();
11 }
```

図 **3.2**　生成されるテストの例

```
{
  "test0": {
    "finallyObjectState": {
      "barClass0": {
        "a": 10
      }
    }
  },
  "test1": {
    "finallyObjectState": {
      "barClass0": {
        "a": 20
      }
    }
  }
}
```

図 3.3　オブジェクトの期待値を記述した JSON ファイルの例

```
01 | 0.000 |package sort;
02 | 0.000 |
03 | 0.000 |public class BugSort{
04 | 0.000 |
05 | 0.000 |    private int[] A;
06 | 0.000 |
07 | 0.000 |    public BugSort(int[] num){
08 | 0.447 |        if(num == null){
09 | 0.000 |            A = new int[]{};
10 | 0.000 |        }else{
11 | 0.577 |            A = num;
12 | 0.000 |        }
13 | 0.000 |    }
14 | 0.000 |
```

図 3.4　出力される HTML の例

行を実行したかを記録する．またこのとき，各テストが終了した時点でのオブジェクトの最終状態を併せて記録する．オブジェクトの状態の取得には XStream[8] を使用した．これは，オブジェクトの各属性を XML 形式の文字列型に変換するためのライブラリである．

3.2.4 ステップ 2-4：各テストにおけるオブジェクトの最終状態と JSON ファイルの比較

ステップ 2-1 で作成した JSON ファイルに記述されているオブジェクトの各属性の期待値と，実際にテストを実行した際のオブジェクトの最終状態を比較し，各テストが成功であったか否かを判定する．ここで，JSON に記述されていない属性に関しては，判定に関与しない．例を挙げると，あるクラス `A` のインスタンスが持つ属性 `b` の期待値が JSON の中に書かれていないとき，属性 `b` がいかなる値となってもテスト結果に影響しない．これは，JSON に記述する不必要な期待値の数を減らすことでユーザの負担を軽減すると共に，時刻や乱数器などに影響されて値が実行毎に変化する属性を無視するための仕様である．

3.2.5 ステップ 2-5：疑惑値の算出

ステップ 2-3 で記録されたテスト毎の通過行の記録，及びステップ 2-4 で判定されたテスト結果を用いて，式 2.1 の疑惑値を算出する．算出した疑惑値は，ユーザにソースコードのどの部分が疑惑値が高いのかがわかりやすくなるよう，疑惑値の値が高い行ほど赤くなるように記述された HTML に出力される．出力された HTML の例として，HTML をブラウザで開いたときの表示を図 3.4 に表す．1 列目が行番号，2 列目がバグの含まれる疑惑値，3 列目がソースコードという並びで記載される．図 3.4 の 8 行目と 11 行目を見ると，疑惑値が 0 を超える場合，ソースコードの色が変化していることがわかる．

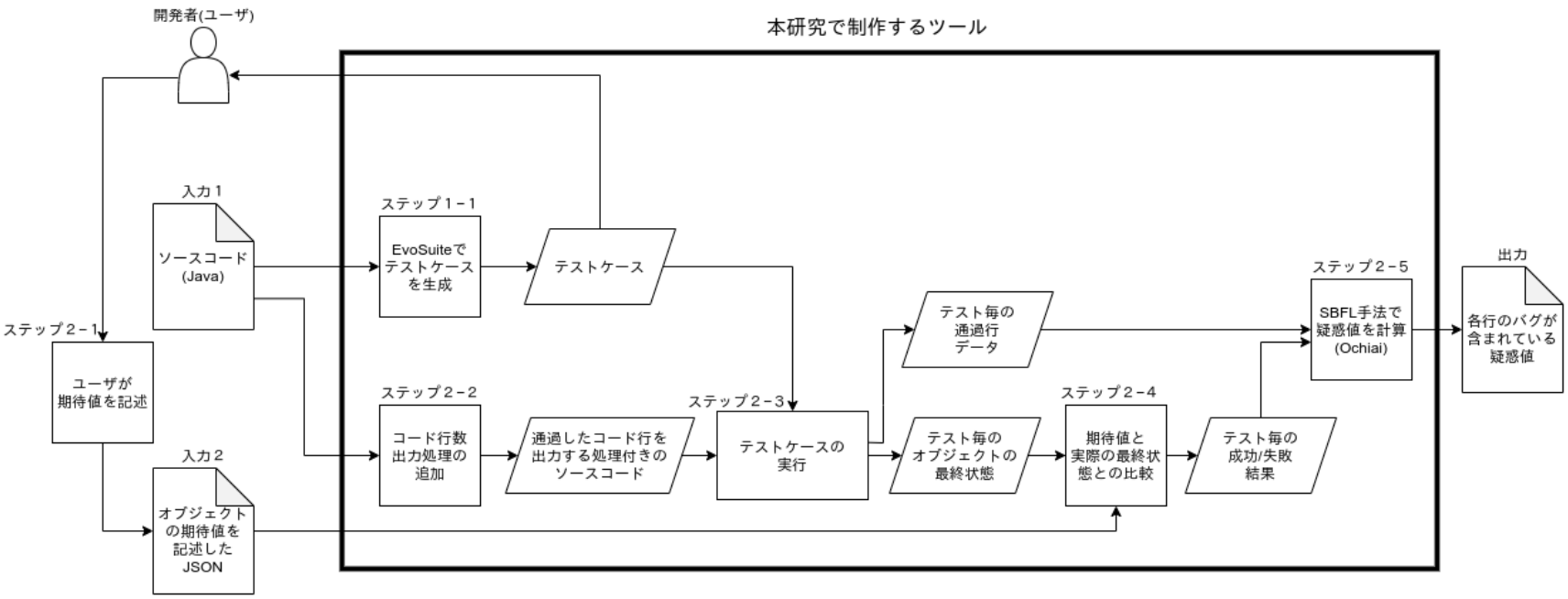
開発者(ユーザ)
本研究で制作するツール
入力1
ソースコード
(Java)
ステップ1－1
EvoSuiteで
テストケース
を生成
テストケース
ステップ2－1
ユーザが
期待値を記述
入力2
オブジェクト
の期待値を
記述した
JSON
ステップ2－2
コード行数
出力処理の
追加
通過したコード行を
出力する処理付きの
ソースコード
ステップ2－3
テストケースの
実行
テスト毎の
通過行
データ
テスト毎の
オブジェクトの
最終状態
ステップ2－4
期待値と
実際の最終状
態との比較
テスト毎の
成功/失敗
結果
ステップ2－5
SBFL手法で
疑惑値を計算
(Ochiai)
出力
各行のバグが
含まれている
疑惑値

図 3.5　ツールの概略図

4. ツールの適用例

作成したツールを Java ソースコードに対して使用し，ツールの性能を確認する．例として作成した各 Java ソースコードには，その中の 1 命令に意図的にバグを含ませる．ツールが出力する疑惑値の値がバグを含ませた行において高くなれば，ツールは正確にバグ箇所を特定できたと言える．

4.1 適用例 1：BugSort

4.1.1 BugSort の説明

ソースコード BugSort.java は，ある整数型の配列に対してバブルソートを行い，配列の中身を降順に並び替えるクラス `BugSort` を定義している．`BugSort` の属性及びメソッドの説明を表 4.1 に記載する．

BugSort.java では，メソッド `buggySort()` の 1 行にバグを含ませる．`buggySort()` 以外のメソッドの実装にはバグは存在しない．`buggySort()` 内のソースコード行の疑惑値と比較するため，クラス `BugSort` にはメソッド `correctSort()` を定義する．`collectSort()` は `buggySort` のバグが無いバージョンを実装したものとなっている．`buggySort()` の定義を図 4.1 に表す．図 4.1 の 6 行目がバグとなっており，`correctSort()` では `A[j] = A[j+1];` となっている．

4.1.2 生成されたテストケース

BugSort.java をツールのフェーズ 1 に入力すると，テストを 5 個含んだテストケースが生成された．生成されたテストケースの説明を表 4.2 に記載する．

4.1.3 各テストでのオブジェクトの期待値の定義

生成されたテストケースでは，各テストにおいてクラス `BugSort` のインスタンスが生成されている．それらインスタンスがテストを終了した際の属性の値，つまり `A` の値のテスト終了時の期待値を JSON に記述する．JSON に記述した `A` の期待値を表 4.3 に記載する．

ツールのフェーズ 2 に BugSort.java 及び表 4.3 の期待値を記述した JSON ファイルを

表 4.1　クラス BugSort の属性・メソッドの説明

種類	型	名前	説明
属性	int[]	A	ソートを行う配列
メソッド	(constructor)	BugSort(int[] array)	ソートしたい配列array をA に代入する (array がnull の場合，A は要素数 0 の配列となる)
	void	correctSort()	A に対して降順バブルソートを実行する (バグなし)
	void	buggySort()	A に対して降順バブルソートを実行する (バグあり)
	void	output()	A を標準出力に出力する

```
1  public void buggySort(){
2      for( int i=0; i<A.length-1; i++ ){
3          for( int j=0; j<A.length-i-1; j++ ){
4              if( A[j] < A[j+1] ){
5                  int tmp = A[j];
6                  A[j+1] = A[j]; //buggy line
7                  A[j+1] = tmp;
8              }
9          }
10     }
11 }
```

図 4.1　メソッド buggySort の定義

入力すると，テストケースが実行され，各テストにおける A の最終値と JSON に記述された A の値が比較される．実際の値と期待値の間に相違があるテストは失敗であるとみなされる．テストを実行した際の実際の A の値とテスト結果を同じく表 4.3 に記載する．今回は，テスト番号 0 のみが失敗という結果になっている．テスト番号 0 を実行したときの実際の A を取得すると，その値は `[0,0,0,-2940,-2940,-2940,-2940,-2940]` となっていることがわかった．このことから，テスト番号 0 では明らかにソートが失敗していることがわかる．

4.1.4 疑惑値の算出

表 4.3 のテスト結果と各テストの実行時に記録された通過行のデータから，ソースコードの各行に対する疑惑値を計算する．疑惑値の計算方法は式 2.1 にある Ochiai の計算式を用いる．各テストにおける通過行及び計算された疑惑値を表 4.4 に記載する．図 4.1 の `buggySort()` の 6 行目は，コード全体では 32 行目に相当する．すなわち，BugSort.java の真にバグが含まれる箇所は 32 行目である．表 4.4 には 32 行目の行番号に米印を添付した．表 4.4 の疑惑値を見ると，32 行目を含んだ 29 行目から 33 行目までが，ソースコードの中で最も大きい値を取ることがわかる．そのため，制作したバグ限局ツールはある程度バグのある箇所を特定できたと言える．

4.2 適用例 2：BugFizzBuzz

4.2.1 BugFizzBuzz の説明

FizzBuzz とは，ある整数の配列 $(n, n+1, n+2, ...)$ に対して，その i 番目の要素 $n+i$ が式 4.1 のみを満たすとき，配列の i 番目の要素を Fizz に変換し，式 4.2 のみを満たすときは Buzz に変換，その両方の式を満たすときは FizzBuzz に変換するアルゴリズムである．なお，式 4.1 及び式 4.2 を両方満たさない場合は変換せず，数値をそのまま返す．この FizzBuzz アルゴリズムを実装した BugFizzBuzz.java を，ツールの適用対象として使用する．

$$n + i \equiv 0 \bmod 3 \tag{4.1}$$

表 4.2　BugSort.java から生成されたテストケースの説明

テスト番号	テスト内容の説明
0	配列[0,0,0,-2940,0,0,0,0] をbuggySort() によってソート
1	コンストラクタにnull を渡してbuggySort() によってソート
2	配列[0,0,0,-2940,0,0,0,0] をcorrectSort() によってソート
3	配列[0,0,0,0,0,0,0,0] をoutput() で出力
4	コンストラクタにnull を渡してcorrectSort() によってソート

表 4.3　各テストの終了時における A の期待値とテスト結果

テスト番号	A の期待値	A の実際の値	テスト結果
0	[0,0,0,0,0,0,0,-2940]	[0,0,0,-2940,-2940, -2940,-2940,-2940]	失敗
1	[]	[]	成功
2	[0,0,0,0,0,0,0,-2940]	[0,0,0,0,0,0,0,-2940]	成功
3	[0,0,0,0,0,0,0,0]	[0,0,0,0,0,0,0,0]	成功
4	[]	[]	成功

表 4.4　BugSort.java の各テストにおいて通過した行番号と疑惑値

行番号	メソッド名	通過箇所					疑惑値
		テスト 0	テスト 1	テスト 2	テスト 3	テスト 4	
8	BugSort	○	○	○	○	○	0.477
9			○			○	0.000
11		○		○	○		0.577
16	correctSort			○		○	0.000
17				○			0.000
18				○			0.000
19				○			0.000
20				○			0.000
21				○			0.000
28	buggySort	○	○				0.707
29		○					1.000
30		○					1.000
31		○					1.000
32 ※		○					1.000
33		○					1.000
40	output				○		0.000
42					○		0.000
44					○		0.000
テスト結果		失敗	成功	成功	成功	成功	

$$n + i \equiv 0 \bmod 5 \tag{4.2}$$

BugFizzBuzz.java には BugFizzBuzz クラスが定義されている．クラスの説明は表 4.5 に記載する．クラスの主な使い方の例を挙げると，まず変換する配列の開始値をコンストラクタに渡し，配列を作成して allCorrectFizzBuzz() または allBuggyFizzBuzz() によって FizzBuzz による変換を行い，output() で出力する．

BugFizzBuzz.java におけるバグは，メソッド buggyFizzBuzz() に存在する．buggyFizzBuzz() の定義を図 4.2 に記載する．図 4.2 における 6 行目がバグのある行である．本来，6 行目の条件文は図 4.2 の 2 行目の条件文の前に評価される必要があり，buggyFizzBuzz() では num が式 4.1 及び 4.2 を満たす値を取ったとしても，先に図 4.2 の 2 行目の条件式で真となってしまい，3 行目で文字列”Fizz”を返してしまう．buggyFizzBuzz() の正解バージョンである correctFizzBuzz の定義は図 4.3 に記載する．

4.2.2 生成されたテストケース

BugFizzBuzz.java をツールのフェーズ 1 に入力し，生成されたテストケースの説明を表 4.6 に記載する．

4.2.3 各テストでのオブジェクトの期待値の定義

ツールのフェーズ 2 に入力する JSON ファイルに書かれた，各テストにおけるオブジェクトの最終状態の期待値を 4.7 に記載する．

4.2.4 疑惑値の算出

各テストで通過した行と，算出した疑惑値の値を表 4.8 に記載する．図 4.2 の 6 行目は BugFizzBuzz.java では 60 行目に位置する．すなわち，BugFizzBuzz.java の真にバグが含まれる箇所は 60 行目ということである．表 4.4 と同じく，表 4.8 においてバグの存在する 60 行目には行番号に米印を記載している．

バグの含まれる 60 行目の疑惑値は最も高い値となっているが，バグが含まれる buggyFizzBuzz() の全ての行において同じ値となっており，また allBuggyFizzBuzz()

内の行においてもバグのある行と同じ疑惑値となっているため，バグ箇所の限局は BugSort.java の例より効果が薄いと言える．

表 4.5　クラス BugFizzBuzz の属性・メソッドの説明

種類	型	名前	説明
属性	int	start	配列の先頭の数値
	int[]	int_array	FizzBuzz によって変換される配列
	final int	array_size	配列のサイズ (= 10)
	String[]	string_array	変換後の配列
メソッド	(constructor)	BugFizzBuzz(int start)	配列の開始値start を取得
	void	allCorrectFizzBuzz()	correctFizzBuzz() によって配列を変換
	void	allBuggyFizzBuzz()	buggyFizzBuzz() によって配列を変換
	void	output()	string_array を標準出力に出力する
	void	makeIntArray()	int_array にstart から始まる配列を格納 (コンストラクタからのみ呼ばれる)
	void	correctFizzBuzz(int num)	num を FizzBuzz で変換した値を返す (バグなし)
	void	buggyFizzBuzz(int num)	num を FizzBuzz で変換した値を返す (バグあり)

```
1  private String buggyFizzBuzz( int num ) {
2      if ( num % 3 == 0 ) {
3          return "Fizz";
4      }else if ( num % 5 == 0 ) {
5          return "Buzz";
6      }else if ( num % 3 == 0 && num % 5 == 0 ) { //buggy line
7          return "FizzBuzz";
8      }
9      return String.valueOf(num);
10 }
```

図 4.2　メソッド **buggyFizzBuzz** の定義

```
1  private String correctFizzBuzz( int num ) {
2      if ( num % 3 == 0 && num % 5 == 0 ) {
3          return "FizzBuzz";
4      }else if ( num % 3 == 0 ) {
5          return "Fizz";
6      }else if ( num % 5 == 0 ) {
7          return "Buzz";
8      }
9      return String.valueOf(num);
10 }
```

図 4.3　メソッド **correctFizzBuzz** の定義

表 4.6　BugFizzBuzz.java から生成されたテストケースの説明

テスト番号	テスト内容の説明
0	[-1855,-1854,-1853,-1852,-1851,-1850,-1849,-1848,-1847,-1846] をallBuggyFizzBuzz() で変換
1	[-16,-15,-14,-13,-12,-11,-10,-9,-8,-7] を allCorrectFizzBuzz() で変換
2	[1754,1755,1756,1757,1758,1759,1760,1761,1762,1763] を output() で出力
3	[1754,1755,1756,1757,1758,1759,1760,1761,1762,1763] を allBuggyFizzBuzz() で変換
4	[1754,1755,1756,1757,1758,1759,1760,1761,1762,1763] を allCorrectFizzBuzz() で変換

表 **4.7**　各テストの終了時における **string_array** の期待値とテスト結果

テスト番号	string_array の期待値 (上段) と実際の値 (下段)	テスト結果
0	["Buzz","Fizz","-1853","-1852","Fizz","Buzz", "-1849","Fizz","-1847","-1846"]	成功
	["Buzz","Fizz","-1853","-1852","Fizz","Buzz", "-1849","Fizz","-1847","-1846"]	
1	["-16","FizzBuzz","-14","-13","Fizz","-11", "Buzz","Fizz","-8","-7"]	成功
	["-16","FizzBuzz","-14","-13","Fizz","-11", "Buzz","Fizz","-8","-7"]	
2	["1754","1755","1756","1757","1758","1759", "1760","1761","1762","1763"]	成功
	["1754","1755","1756","1757","1758","1759", "1760","1761","1762","1763"]	
3	["1754","FizzBuzz","1756","1757","Fizz","1759", "Buzz","Fizz","1762","1763"]	失敗
	["1754","Fizz","1756","1757","Fizz","1759", "Buzz","Fizz","1762","1763"]	
4	["1754","FizzBuzz","1756","1757","Fizz","1759", "Buzz","Fizz","1762","1763"]	成功
	["1754","FizzBuzz","1756","1757","Fizz","1759", "Buzz","Fizz","1762","1763"]	

表 4.8　BugFizzBuzz.java の各テストにおいて通過した行番号と疑惑値

行番号	メソッド名	通過箇所					疑惑値
		テスト 0	テスト 1	テスト 2	テスト 3	テスト 4	
13	BugFizzBuzz	○	○	○	○	○	0.447
14		○	○	○	○	○	0.447
18	allCorrect-		○			○	0.000
19	FizzBuzz		○			○	0.000
24	allBuggy-	○			○		0.707
25	FizzBuzz	○			○		0.707
30	output			○			0.000
31				○			0.000
32				○			0.000
34				○			0.000
38	makeIntArray	○	○	○	○	○	0.447
39		○	○	○	○	○	0.447
40		○	○	○	○	○	0.447
45	correctFizzBuzz		○			○	0.000
46			○			○	0.000
47			○			○	0.000
48			○			○	0.000
49			○			○	0.000
50			○			○	0.000
52			○			○	0.000
56	buggyFizzBuzz	○			○		0.707
57		○			○		0.707
58		○			○		0.707
59		○			○		0.707
60 ※		○			○		0.707
63		○			○		0.707
テスト結果		成功	成功	成功	失敗	成功	

5. 考察

5.1 ツールの適用例から考えられるツールの問題点

ツールの適用例では，BugSort.java と BugFizzBuzz.java の間でバグ特定の有効性に大きな違いがあった．この理由は，以下の 2 つの理由に因るものであると考えられる．

1. バグの存在するメソッドのアクセス修飾子の違い
2. バグのあるバージョンと正解バージョンとの差異の違い

1 つめの理由であるバグが存在するメソッドのアクセス修飾子の違いというのは，BugSort.java のメソッド `buggySort()` は public で宣言されており，BugFizzBuzz.java のメソッド `buggyFizzBuzz()` は private で宣言されているということである．メソッド `buggySort()` は public で宣言されているため，EvoSuite によって生成されるテストケースから直接呼び出すことができる．そのため，テストケースのステートメントカバレッジを 100%に近づけるためには，メソッド `buggySort()` を呼び出す前の処理をランダムに設定した複数のテストで `buggySort()` を呼び出す．このとき，`BugSort` のコンストラクタ以外のメソッドを呼び出すことが無いため，`buggySort()` 以外の行においては疑惑値が低くなったと言える．しかし，メソッド `buggyFizzBuzz()` は private で宣言されており，`buggyFizzBuzz()` をテストするには `buggyFizzBuzz()` を 1 回以上呼び出すメソッドをテストケースから呼び出す必要がある．このことにより，`buggyFizzBuzz()` を通過して結果が失敗であるテストは，同時に `buggyFizzBuzz()` を呼び出すメソッドを必ず通過することになり，その結果複数のメソッド内の行における疑惑値が高くなったと考えられる．

次に 2 つめの理由であるバグのあるバージョンと正解バージョンとの差異の違いを説明すると，BugSort.java では `buggySort()` の正解バージョンとの差異が 1 行の「変更」であったことに対し，BugFizzBuzz.java では `buggyFizzBuzz()` の正解バージョンとの差異が 1 行の「移動」であったことである．`bugSort()` においてバグ箇所を通過すると，ソートアルゴリズムが成り立たなくなり，テストは必ず失敗する．そのため，バグのある位置を通過したテストケースと通過していないテストケースの 2 通りのテストに明確に分類できるため，バグ箇所とそうでない行の間には疑惑値の差

が生じやすくなる．しかし，buggyFizzBuzz() ではバグ箇所を通ると不具合が発生するというものではなく，バグ箇所にある文を適切に移動すれば不具合が発生せずに正しく動作する．このことから，bugFizzBuzz() ではバグの発生する理由はバグ箇所の文のみではなく，正解バージョンにおけるバグ箇所の移動先に，そもそもバグ箇所の文が無いことが原因である．つまり，バグ箇所のみがバグの原因ではないと言うことができる．

上記 2 つの理由から本ツールでは，実行するには複数のメソッドを呼び出す必要があるバグや，単純にバグ箇所のみの書き換えによって修正できないバグにおいては，バグ箇所の特定が難しいことがわかる．

5.2 今後の課題

制作したツールは，まだ実際に開発されている Java プロジェクトに対して使用できるほど実用的ではない．そのため，バグ箇所特定の精度及びインタフェースの品質の向上のための今後の課題として，以下のものが挙げられる．

1. SBFL と他の手法との組み合わせの検討

   本ツールでは，バグ箇所の特定には SBFL 手法のみを使用している．SBFL 手法によるバグ限局のみでは，ツールの適用例から考えられるツールの問題点に説明した問題を解決することは難しい．そのため，SBFL 以外の手法を加えてツールに組み込み，バグ限局の精度を高めることが必要となる．

2. 期待値の記述における曖昧な表現の許可

   テストが成功かについて，現時点ではオブジェクトの最終状態の期待値と実際の属性値が完全に一致しているかどうかを基準とした．しかし，オブジェクトの期待値がユーザにとってある一つに決めにくい場合があり，そのような属性は期待値を記述できなくなるという問題がある．そこで，例えば「属性 a の値が 10 以上である」などのように，期待値に範囲をもたせる記述を許可することで，より適切にテストの成功または失敗の判定を行うことができると考えられる．

6. 結言

本実験では，小規模な Java プロジェクトにおいて使用できる，バグ箇所の特定を自動的に行うツールを作成した．

自動的に生成されたテストケースの成功または失敗の判定方法として，オブジェクトの期待値をユーザ自身が記述する方法を実装した．

また，制作したツールが実際のソースコードに対してバグ特定を行えるかを確かめるため，著者が制作した 2 種類の Java ソースコードに対してツールを適用し，疑惑値がどの位置で高くなるかを評価した．評価の結果，制作したツールによるバグ限局の有効性はソースコードによって異なり，問題のある行のみを書き換えることによってバグを解消できるソースコードや，バグの存在するメソッドをテストケースから直接呼び出すことができるソースコードではより高い精度でバグ限局が可能であることがわかった．

これらの結果から，ソフトウェア開発におけるデバッグの工程に掛かるコストの削減に貢献し，バグ箇所を特定するアプリケーションの制作の足掛かりとなるツールを作成することができたと言える．

謝辞

本研究を行うにあたり，研究テーマの決定や使用するツールの選択，報告書の記述に至るまで全ての面で丁寧なご指導を頂きました，本学情報工学・水野修教授，崔恩瀞助教及び名古屋大学情報学研究科・吉田則裕准教授に厚く御礼申し上げます．また，本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻・近藤将成先輩，西浦生成先輩をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

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

[2] A. Koyuncu, K. Liu, T.F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, “ifixr: bug report driven program repair,” Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp.314–325, 2019.

[3] D.L. Shaowei Wang, “Version history, similar report, and structure: Putting them together for improved bug localization,” 22nd International Conference on Program Comprehension (ICPC 2014), pp.53–63, 2014.

[4] R. Abreu, P. Zoeteweij, and A.J. Van Gemund, “An evaluation of similarity coefficients for software fault localization,” 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06), pp.39–46, IEEE, 2006.

[5] R. Abreu, P. Zoeteweij, and A.J. Van Gemund, “On the accuracy of spectrum-based fault localization,” Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), pp.89–98, IEEE, 2007.

[6] evosuite.org, EvoSuite — Automatic Test Suite Generation for Java, （オンライン）, 入手先〈https://www.evosuite.org/index.html〉（参照 2020/11/18）.

[7] G. Fraser and A. Arcuri, “Whole test suite generation,” IEEE Transactions on Software Engineering, vol.39, no.2, pp.276–291, 2012.

[8] XStream - About XStream, （オンライン）, 入手先〈https://x-stream.github.io/index.html〉（参照 2021/12/15）.