一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

網羅率とSBFL適合性を用いた手動テストと自動生成テストの比較調査

清水ささら†　肥後　芳樹†

† 大阪大学大学院情報科学研究科 〒565–0871 大阪府吹田市山田丘 1–5
E-mail: †{simizu-s,higo}@ist.osaka-u.ac.jp

あらまし　ソフトウェアの品質はシステム開発の成功に不可欠であり，特に大規模で複雑なソフトウェアではバグのリスクが高いため，早期の発見と修正が重要である．テスト工程は開発プロセスにおいて欠かせないが，手動でのテスト作成や実行は時間を多く要するため，効率的なテスト手法が求められている．そのため，自動でテストを生成するツールの研究が行われており，それらのツールはソースコードや仕様に基づいてテストケースを自動生成し，開発者の負担を軽減する．本研究では，Java プログラムに対して自動テスト生成ツールを用いてテストを生成し，開発者が作成したテストと自動生成されたテストの網羅率および Fault Localization（FL）ツールを適用した結果を比較する．オープンソースリポジトリ Defects4J の Java プロジェクトを利用し，手動テストとして開発者が作成したテストケースを使用する．自動テスト生成には EvoSuite を利用し，生成された自動テストケースと手動テストケースを比較することで，それぞれの強みと弱点を明らかにすることを目的としている．

キーワード　単体テスト，テスト自動生成，網羅率，SBFL

1. はじめに

ソフトウェア開発において，単体テストは欠かせない工程の一つであり，コードの品質向上やバグの早期発見に大きく寄与する．しかし，単体テストを手動で作成することは非常に手間がかかり，時間的な制約や人的リソースの制限が課題となる．特に大規模なプロジェクトでは，すべての関数やクラスに対して網羅的なテストを作成することは現実的ではない．

これらに問題に対して，単体テストを自動生成するツールが存在する．これらのツールは，プログラムコードを解析し，テストケースを自動的に生成することで，開発者の負担を軽減する．代表的なツールには，EvoSuite [1] や Randoop [2] などが挙げられる．

テスト自動生成ツールを用いた既存研究はいくつか存在する．Fraser らは，複数のテスト自動生成ツールに対して故障箇所の特定にどの程度役立つのかを調査した [3]．対象となったツールは，Randoop、EvoSuite、Agitar [4] の三つである．その結果，自動生成されたテストスイートは全体の 55.7 %の故障を検出したが，個々のテストスイートのうち故障を検出したのはわずか 19.9 %であることがわかった．

Serra らは，手動で作成されたテストケースと自動生成されたテストケースを比較する研究を行っている [5]．コードカバレッジ，ミューテーションスコア，バグ検出能力の三つの観点から評価が行われた．この研究では．過去 10 年間における自動テスト生成技術の進展を評価するために，手動で作成されたテストケースと EvoSuite，Randoop，JTExpert [6] が生成したテストケースを対象とした．その結果，自動生成ツールはコードカバレッジとミューテーションスコアで優れた性能を示したものの，バグ検出能力では手動テストに劣ることがわかった．また，手動と自動生成テストを組み合わせることで，それぞれの弱点を補完し合う可能性が示されている．本研究でも手動で作成されたテストと自動で生成されたテストを比較しているが，本研究では SBFL 適合性を用いて評価している点においてこの先行研究とは異なる．

Bhatia らは，ChatGPT を活用して Python プログラムの単体テストスクリプトを生成し，その性能を既存の単体テスト生成ツール Pynguin [7] と比較した [8]．評価は，網羅率，正確性，可読性の観点で実施され，ChatGPT は網羅率において Pynguin と同等の性能を示し，場合によっては Pynguin を上回る結果が得られた．しかし，ChatGPT が生成したアサーションの約 3 分の 1 に誤りが含まれていることも報告されている．また，ChatGPT と Pynguin が見逃した文の重複が少ないことから，両者を組み合わせることで，単体テスト生成の性能を向上させられる可能性が示された．

松田らは，自動バグ修正におけるテストケースの構成が修正結果に及ぼす影響を調査した [9]．彼らは，5 種類のバグパターンを対象にテストケースを操作し，生成されたパッチの数や正確性，修正に要した時間を評価した．その結果，バグの種類に応じて成功テストケースと失敗テストケースの数を適切に調整することが重要であることがわかった．また，過剰適合を抑制するには，成功テストケースを増やすことが特に効果的であることも示された．

Watanabe らは，プログラムの構造が自動テスト生成ツールによるテストスイートの網羅率に与える影響を調査している [10]．この研究では，EvoSuite を用いて生成されたテストスイートを分析し，網羅率が低かった要因を四つのパターン（特定の値や型の要求，実行不可能コード，マルチスレッド処理）に分類した．また，これらの課題に対する解決方法も考察し

ており，特定の条件を満たす値や型については，ダミー分岐の挿入による対処が可能である一方で，マルチスレッド処理などの問題に関しては，ツールそのものの改良が必要であることが明らかとなった．

本研究では，手動で作成されたテストと自動生成されたテストを網羅率．SBFL スコアの二つの観点から比較する．人間が作成したテストケースと自動生成されたテストケースの性能を比較し，それぞれの強みと弱点を明らかにすることを目的としている．この比較を通じて，手動テストと自動生成テストをどのように組み合わせるべきか，またどのような場面でそれぞれを活用すべきかについての知見を得ることを目指す．

2. 準　　備

2.1 単体テスト

ソフトウェア開発においては，プログラムが開発者の想定通りに動作するかどうかを検証するために，ソフトウェアテストが実施される．ソフトウェアテストは，テスト対象の粒度に応じて複数の段階で実施される．

単体テストは，テスト対象の最小単位である関数やメソッドに対して行うテスト工程である．この工程はプログラム実装後の早期段階で実施されるため，比較的早期にバグや問題を特定できるという利点がある．その結果，単体テストはソフトウェア開発において重要不可欠な工程となっている．

単体テストの作業効率を向上させるために，テスト自動化フレームワークが広く活用されている．テスト自動化フレームワークは，単体テストを実行するためのソフトウェア環境やテストケースの記述方法を提供するものであり，テスト実行の自動化や実行結果のレポート生成などの機能を持つ．例えば，Java 用の代表的なテスト自動化フレームワークとして JUnit があり，Eclipse や IntelliJ IDEA などの統合開発環境でサポートされている．

単体テストを実施するには，テスト対象のコードに対応するテストスイートを用意する必要がある．テストスイートとは，特定のテスト目標を達成するために作成されたテストケースの集合を指す．一方，テストケースは，テスト項目の最小単位であり，テスト対象への入力と期待される結果の組み合わせで構成される．

2.2 テスト自動生成技術

単体テストのテストスイートを作成する作業は多大な 労力を要する. そこで，単体テストのテストスイートを自動的に 生成する研究が行われている．代表的なツールには，EvoSuite や Randoop などが挙げられる．

本研究では EvoSuite を用いる．EvoSuite は Java プロジェクトに対するテストを自動生成するツールである．一つの Java クラスに対して，単体テストの自動化フレームワークである JUnit 形式のテストスイートを自動的に生成する．ハイブリッド探索，動的記号実行，テスト可能性変換などの探索的アプローチを利用し，テスト対象のコードをできるだけ多くカバーできるようなテストスイートを生成する．最初に，複数のランダムなテストケースを生成し，そこから探索的アプローチを繰り返し適用して進化させる．最も高いコードカバレッジを持つテストスイートがカバレッジ基準を保って最小化されるため，生成される単体テストは必要最低限に抑えられている．この EvoSuite は多くの既存研究で利用されている．

2.3 研 究 目 的

本研究では，開発者によって作成されたテストと自動生成ツールで生成されたテストを網羅率．SBFL スコアの二つの観点から比較する．開発者が作成したテストケースと自動生成されたテストケースの性能を比較し，それぞれの強みと弱点を明らかにすることを目的としている．先行研究では扱われていない SBFL スコアを用いて比較を行うことで，それぞれの特徴が新たな視点から明らかになることが見込まれる．この比較を通じて，手動テストと自動生成テストをどのように組み合わせるべきか，またどのような場面でそれぞれを活用すべきかについての知見を得ることを目指す．これにより、ソフトウェア開発現場におけるテスト設計プロセスの改善に寄与することが期待される．

3. 評 価 指 標

本研究で評価指標として用いる網羅率，SBFL スコアについて説明する．

3.1 網　羅　率

網羅率とは，テスト対象コードにおいてテストされた割合を表す指標である. 網羅率を計測する際の基準を網羅基準と呼ぶ．網羅基準にはさまざまな種類があり，代表的な基準として命令網羅，分岐網羅がある．本研究で用いる，命令網羅，分岐網羅について説明する．命令網羅では，テスト対象コードに含まれる実行可能な行のうち，テストスイートによって実行された割合を計測する．命令網羅によって網羅率を計測する場合，以下の式を用いる．

$$\text{命令網羅率} = \frac{\text{テストスイートによって実行された行数}}{\text{対象コードに含まれる実行可能な行の総数}} \tag{1}$$

分岐網羅では，テスト対象コードに含まれる分岐に対し，その分岐条件が真となる場合と偽となる場合をテストスイートによって網羅できたかどうかで計測する. 分岐網羅によって網羅率を計測する場合，以下の式を用いる．

$$\text{分岐網羅率} = \frac{\text{テストスイートによって実行された分岐の数}}{\text{対象コードに含まれる分岐の総数}} \tag{2}$$

3.2 SBFL

プログラム中の欠陥箇所を推測する技術の一つに，Spectrum-Based Fault Localization (SBFL) [11] が存在する．テスト実行時にどの文が実行されたかの実行経路情報をもとに，推測を行う．失敗したテストケースで実行された文は，欠陥箇所である可能性が高く，成功したテストケースで実行された文は，欠陥箇所である可能性が低いという考えに基づく．

SBFL による欠陥箇所の特定方法について説明する．まず，

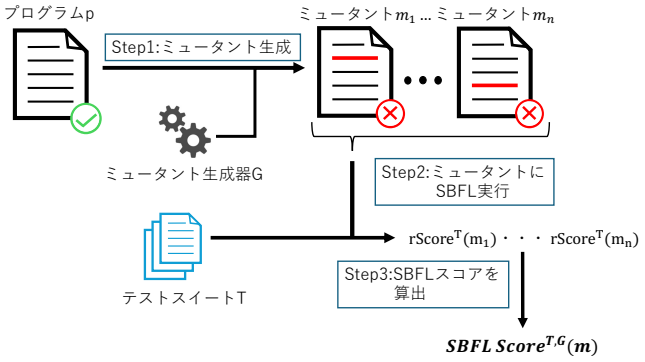

図 1　SBFL スコアの算出方法

すべてのテストを実行し，テストの成否と実行経路情報を記録する．次に，これらの情報を利用して，疑惑値と呼ばれる，欠陥である可能性の高さを示す値を文ごとに算出する．

Ochiai [12] の計算式における疑惑値 $susp(s)$ の算出方法を (3) に示す．ここで，$fail(s)$ は文 s を実行した失敗テストの数，$pass(s)$ は文 s を実行した成功テストの数，$totalFail$ はすべての失敗テストの総数である．

$$susp(s) = \frac{fail(s)}{\sqrt{totalFail \times (fail(s) + pass(s))}} \tag{3}$$

この $susp(s)$ をすべての文に対して算出し，その値が高い文ほど，欠陥の原因箇所である可能性が高いと推測する．

3.3　SBFL スコア

佐々木らは SBFL 適合性を提案した [13]．SBFL 適合性はプログラム自体が SBFL にどの程度適しているかを示す指標である．この SBFL 適合性を示す値を SBFL スコアと定義している．SBFL スコアは 0 以上 1 以下の値をとり，値が高いほど SBFL 適合性が高い．

SBFL スコアの計測方法ついて説明する．プログラム p の SBFL スコアは，テストスイート T，ミュータント生成器 G に依存するとして，$SBFLScore^{T,G}(p)$ と定義する．

SBFL スコアの算出の流れを図 1 に示す．SBFL スコアは次の三つのステップによって算出される．

Step1　プログラム p から複数のミュータントを生成

Step2　各ミュータントに SBFL を実行し，疑似欠陥の疑惑値の順位を算出

Step3　各ミュータントに含まれる疑似欠陥の疑惑値の順位から，SBFL スコアを算出

Step1. ミュータントを生成する

対象プログラムとテストスイートを用意する．このプログラムに対して，ミュータント生成器を用いてミュータントを生成する．このとき，各ミュータントは元のプログラムに対して一箇所だけが変更されるように生成する．プログラムには変更可能な箇所が複数存在するため，ミュータントは複数生成される．ミュータント生成器 G によって生成されたミュータントの集合を $M^G(P)$ と定義する．

Step2. ミュータントに対して SBFL を実行

$M^G(P)$ に含まれる各ミュータントに対して，テストスイート T を用いて SBFL を実行し，文ごとの疑惑値を算出する．ここで，あるミュータント $m \in M^G(P)$ に含まれる各文 s について，以下を定義する．

- $susp^T(s)$: 文 s の疑惑値
- $rank^T(s)$: 文 s の疑惑値の順位
- $rScore^T(s)$: 文 s の疑惑値の正規化順位

疑惑値の算出には Ochiai の計算式を用いる．疑惑値の順位は，疑惑値の高い順に文を並べた際に，欠陥が存在する文を発見するまでに最大で確認しなければならない文の総数とする．疑惑値 1.0 の文が二つ，0.8 の文が一つ存在する場合は，疑惑値 1.0 の文はどちらも 2 位と扱い，疑惑値 0.8 の文は 3 位とする．疑惑値の順位は，文の総数により異なる価値を持つ．例えば，10 個の文のうちの 10 位と，100 個の文のうちの 10 位とでは，後者の方が順位としての価値が高い．そこで，各文が文全体の中でどの程度上位に位置するかを表すため，順位を 0 以上 1 以下の範囲で線形に正規化する．文 s の正規化順位 $rScore^T(s)$ を以下の (4) の通り算出する．1 が最も価値が高く，0 が最も価値が低いことを表す．(4) において，$totalStatements^T$ はテストスイート T によって実行される文の数である

$$rScore^T(s) = 1 - \frac{rank^T(s) - 1}{totalStatements^T - 1} \tag{4}$$

また，ミュータント m に含まれる欠陥の疑惑値の正規化順位を $rScore^T(m)$ と定義する．各ミュータントに含まれる欠陥は一箇所であるため，この値はミュータントごとに一意である．ミュータント m の欠陥を含む文を s^m_{fault} とすると，$rScore^T(m)$ は文 s^m_{fault} の疑惑値の正規化順位である．ミュータントの $rScore$ が高いほど，欠陥箇所を正確に特定できることを意味する．

$$rScore^T(m) = rScore^T(s^m_{fault}) \tag{5}$$

Step3. SBFL スコアを算出

対象プログラムから生成された各ミュータントの $rScore$ の平均をとることにより，SBFL スコアが算出される．なお，$|M^G(P)|$ は，生成されたミュータントの総数を表す．

$$SBFLScore^{T,G}(P) = \frac{1}{|M^G(P)|} \sum_{m \in M} rScore^T(m) \tag{6}$$

4.　実　　験

Java で書かれたテストを対象として網羅率，SBFL スコアを計測し，テストケースの比較を行う．

4.1　実験対象

実験対象として，Java の実際のプロジェクトにおけるバグを収集したデータセットである Defects4J [14] を用いた．Defects4J にはバグのあるコード，修正後のコード，開発者によって作成されたテストが含まれる．Defects4J の Lang に含まれる 65 個のデータセットに対して網羅率，SBFL スコアの計測を行う．

4.2　実験方法

修正後のコードに対して EvoSuite を用いてテストを生成す

る．開発者によって書かれたテストと EvoSuite で生成したテストに対し，網羅率，SBFL スコアの比較を行う．網羅率の測定には JaCoCo(注1) を用い，バグのあるメソッドに対する結果を得る．JaCoCo は命令網羅率，分岐網羅率の算出を行い，網羅率の可視化が可能である．SBFL スコアは 3.3 節の Step1 から Step3 の手順に沿って計測を行う．SBFL の実行には，Gzolter(注2) を用い，Ochiai の計算式で結果を求める．

4.3 実験結果

4.3.1 網羅率の比較結果

Maven のプラグインとして JaCoCo を利用したため，pom.xml ファイルが存在しないメソッドを除いた．また，Java8 以降の環境でビルドに失敗したメソッドを除いた．65 個のうち 36 個のメソッドに対し結果を得られた．

命令網羅の結果を図 2 に，分岐網羅の結果を図 3 に示す．網羅率の平均を表 1 に示す．全体的に見ると網羅率においてどちらが優れているとは一概に言えない結果となった．個々の結果のうち，開発者が作成したテストが高い網羅率を示すメソッド，自動生成テストが高い網羅率を示すメソッドについてテストケースの比較を行った．

自動生成テストの方が網羅率が高くなる場合として，メソッドに多くの分岐が含まれ，それらが単純な条件で構成されている場合が挙げられる．apache-commons-lang3 の math の NumberUtils.java に含まれる crateNumber() メソッドでは，if 文と swith 文による条件分岐が 22 個含まれている．それぞれのテストケース内の assert 文の数を比較したところ，開発者が作成したテストには 75 個，自動生成されたテストには 27 個の assert 文が含まれていた．assert 文が多いにも関わらず，開発者の作成したメソッドが高い網羅率を示さないことから，開発者の作成したテストケースには冗長な assert 文が含まれていることがわかる．コード 1 に例を示す．4，5 行目，7，8 行目，10，11 行目はそれぞれ同じ分岐を通る assert 文であり，どちらか一方を削除しても網羅率は変化しない．このように，分岐が多数存在する場合には，自動生成を用いることで効率的かつ正確にテストを行うことが可能である．

4.3.2 SBFL スコアの比較結果

Java8 以降の環境でビルドに失敗したテストを除いた．また，生成したミュータントに対して全てのテストケースが成功するテストを除いた．65 個のうち 25 個のバグ ID のテストに対して結果を得られた．

SBFL スコアの結果を図 4 に示す．SBFL スコアの平均を比較すると，開発者が作成したテストは 0.81，自動生成されたテストは 0.76 であり，ほとんど差がない結果となった．

表 1　網羅率の平均値の比較

	開発者が作成したテスト	自動生成テスト
命令網羅率 (%)	94.6	93.4
分岐網羅率 (%)	88.0	90.4

（注1）：https://www.jacoco.org/jacoco/

（注2）：https://github.com/GZoltar/gzoltar

まず，開発者が作成したテストの方が SBFL スコアで優れていた場合として，自動生成テストでは発見できなかったミューテーションが存在していたケースが挙げられる．これらのミューテーションが存在するコードは自動生成テストで網羅できていない部分であった．

次に，自動生成テストの方が SBFL スコアで優れていた場合について見る．自動生成テストと開発者が作成したテストで差が 0.17 で最大であった，バグ ID が 28 のテストケースを比較したところ，開発者が作成したテストケースの数が自動生成テストと比較して少ないことがわかった．対象となるクラスファイルに含まれるメソッドは 1 個であった．それに対して開発者が作成したテストケースは 1 個，自動生成されたテストケースは 5 個であった．テストケースが不足していたため，ミューテーションの多様な挿入位置に対応しきれず SBFL スコアが下がったと考える．対象のクラスに含まれるメソッドには，if 文が三つ含まれており，少なくとも三つの assert 文が必要であることは明らかであるが．開発者が作成したテストには assert 文が一つしか含まれておらず，不十分なテストである．

さらに，開発者が作成したテストと自動生成テストのどちらも SBFL スコアが 0.5 に満たない場合も存在した．この場合の共通点として，テストケースの数が全体的に少なかったことが挙げられる．表 2 に SBFL スコアが 0.5 に満たない場合のそれぞれのテストケース数を示す．少ないテストケースでは，多様なミューテーションに十分に対応できず，結果としてバグの検出率が低下する傾向が確認された．

4.4 考　　察

結果から，人間が書くテストと自動生成ツールによるテストの得意分野が異なることが明らかになった．自動生成ツールを活用する場合は，単純な条件分岐やパターン網羅が必要なテストを任せるのが効果的である．このような網羅的なテストは，パターン数が増えるほど生成されるテストケースの数も増加し，テストの作成にかかる時間が膨大になる傾向がある．しかし，テスト自動生成ツールを用いることでテスト設計に要する時間や工数を大幅に削減できる可能性がある．

これらの結果は，人間によるテスト設計と自動生成ツールを組み合わせることで，それぞれの長所を最大限活かし，効率的かつ網羅的なテストスイートを構築できる可能性を示唆している．

5. おわりに

本研究では，Java プログラムに対して自動テスト生成ツールを用いてテストを生成し，開発者が作成したテストと自動生成されたテストの網羅率および SBFL スコアの結果を比較

表 2　テストケース数

バグ ID	開発者が作成したテスト	自動生成テスト
4	2	4
29	15	25

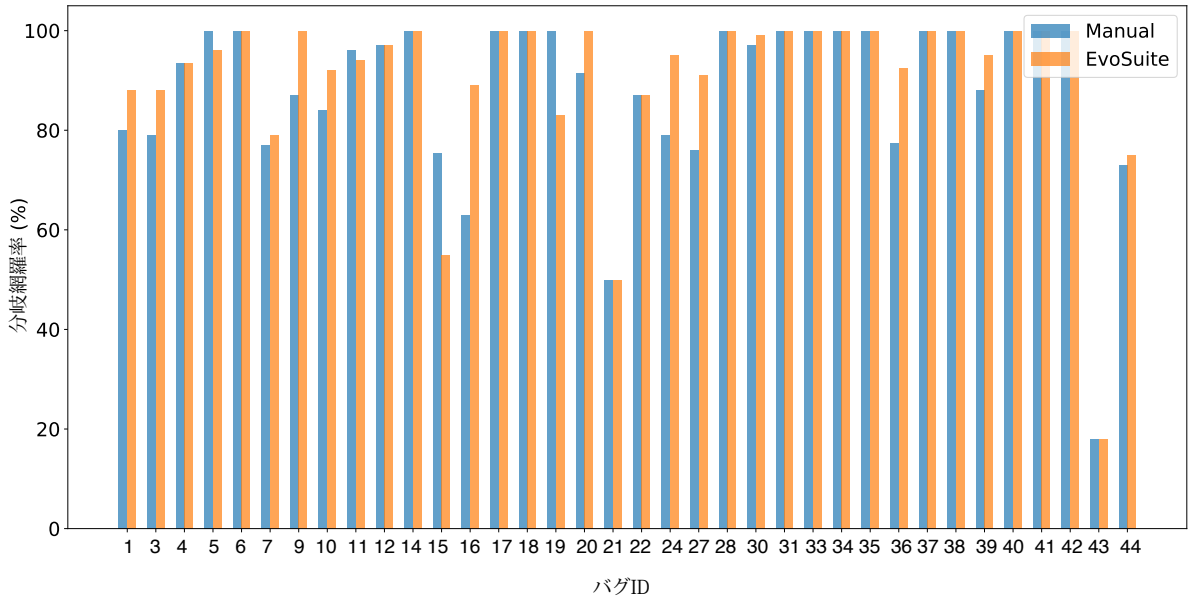

図 2　命令網羅率の比較

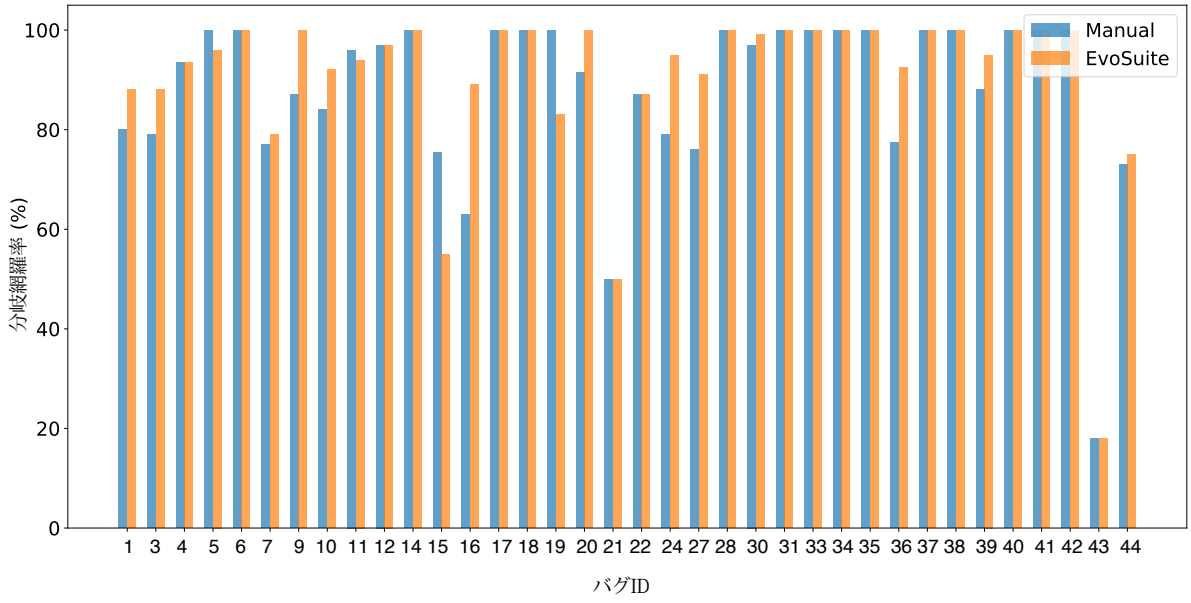

図 3　分岐網羅率の比較

コード 1　冗長な assert 文の例

```
@Test
   public void testCreateNumber() {
   ...
   assertEquals("createNumber(String)␣3␣failed", Double.
        valueOf("1234.5"), NumberUtils.createNumber("
        1234.5D"));
   assertEquals("createNumber(String)␣3␣failed", Double.
        valueOf("1234.5"), NumberUtils.createNumber("
        1234.5d"));
   ...
   assertEquals("createNumber(String)␣4␣failed", Float.
        valueOf("1234.5"), NumberUtils.createNumber("
        1234.5F"));
   assertEquals("createNumber(String)␣4␣failed", Float.
        valueOf("1234.5"), NumberUtils.createNumber("
        1234.5f"));
   ...
   assertEquals("createNumber(String)␣6␣failed", Long.
        valueOf(12345), NumberUtils.createNumber("12345L
        "));
   assertEquals("createNumber(String)␣6␣failed", Long.
        valueOf(12345), NumberUtils.createNumber("12345l
        "));
   ...
   }
```

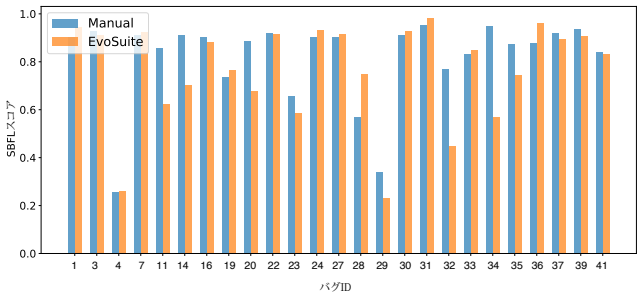

図 4　SBFLscore の比較

した．オープンソースリポジトリ Defects4J の Java プロジェクトを利用し，手動テストとして開発者が作成したテストケースを使用し，自動テスト生成には EvoSuite を利用した．本研究の目的は，手動で作成されたテストと自動生成されたテストを網羅率．SBFL スコアの二つの観点から比較し，それぞれの強みと弱点を明らかにすることである．この比較を通じて，手動テストと自動生成テストをどのように組み合わせるべきか，またどのような場面でそれぞれを活用すべきかについての知見を得ることを目指した．結果から，自動生成ツールを活用する場合は，単純な条件分岐やパターン網羅が必要なテストを任せるのが効果的であることがわかった．このような網羅的なテストは，パターン数が増えるほど生成されるテストケースの数も増加し，テストの作成や実行にかかる時間が膨大になる傾向がある．しかし，テスト自動生成ツールを用いることでテスト設計に要する時間や工数を大幅に削減できる可能性がある

謝辞　本研究は JSPS 科研費 24H00692, 21K18302, 21H04877, 23K24823，22K11985 の助成を受けた．

文　献

[1] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp.416–419, 09 2011.

[2] C. Pacheco and M.D. Ernst, “Randoop: feedback-directed random testing for java,” Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, p.815 – 816, OOPSLA ’07, Association for Computing Machinery, New York, NY, USA, 2007. https://doi.org/10.1145/1297846.1297902

[3] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated unit test generation really help software testers? a controlled empirical study,” ACM Transactions on Software Engineering and Methodology, vol.24, pp.1–49, 09 2015.

[4] “Agitar one,” http://www.agitar.com/solutions/products/automated_junit_generation.html, 2014. (Accessed on 08/01/2024).

[5] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H.C. Gall, and A. Bacchelli, “On the effectiveness of manual and automatic unit test generation: Ten years later,” 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp.121–125, 2019.

[6] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, “Instance generator and problem representation to improve object oriented code coverage,” IEEE Transactions on Software Engineering, vol.41, no.3, pp.294–313, 2015.

[7] S. Lukasczyk and G. Fraser, “Pynguin: automated unit test generation for python,” Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, p.168 – 172, ICSE ’22, Association for Computing Machinery, New York, NY, USA, 2022. https://doi.org/10.1145/3510454.3516829

[8] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, “Unit test generation using generative ai: A comparative performance analysis of autogeneration tools,” Proceedings of the 1st International Workshop on Large Language Models for Code, p.54 – 61, LLM4Code ’24, Association for Computing Machinery, New York, NY, USA, 2024. https://doi.org/10.1145/3643795.3648396

[9] 松田直也，丸山勝久，“テストケースが自動バグ修正に与える影響の調査,” コンピュータ ソフトウェア，vol.37，no.4，pp.31–37，2020.

[10] R. Watanabe, Y. Higo, and S. Kusumoto, “Impacts of program structures on code coverage of generated test suites,” Product-Focused Software Process Improvement: 24th International Conferenc (PROFES), pp.355–362, 2023.

[11] H. Souza, M. Chaim, and F. Kon, “Spectrum-based software fault localization: A survey of techniques, advances, and challenges,” 07 2016.

[12] 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, 2006.

[13] 佐々木唯，肥後芳樹，柗本真佑，楠本真二，“プログラムに対する欠陥限局の適合性計測,” 情報処理学会論文誌，vol.62，no.4，pp.1029–1038，04 2021.

[14] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: a database of existing faults to enable controlled testing studies for java programs,” ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, p.437 – 440, ISSTA 2014, Association for Computing Machinery, New York, NY, USA, 2014. https://doi.org/10.1145/2610384.2628055