# テストケースの実行パス分析に基づくテスト効率の評価指標の提案

# A Method for Evaluating Software Testing Efficiency on the Basis of Program Execution Path Analysis

八巻奈々恵† 橋浦弘明‡ 古宮誠一‡
Nanae Yamaki† Hiroaki Hashiura‡ Seiichi Komiya‡
†芝浦工業大学　工学部
‡芝浦工業大学大学院
† Faculty of Engineering, Shibaura Institute of Technology.
‡ Graduate School of Engineering, Shibaura Institute of Technology.

**要旨**
テストカバレッジに関する現行の基準では、個々のテストケースの重複を測ることは出来ない。入出力が異なるテストケース同士であっても、テストケースの働きとしては同値になる（重複している）場合がある。重複した単体テストケースが多くある場合、たといカバレッジが高くても同値分割などがうまく実施されておらず、テストケースの効率評価としては適切ではない。本研究では、テストケースの実行パス分析を行い、実行パス数をテストケース設計作業の効率の評価指標として用いることを提案する。

## 1. はじめに

ソフトウェア開発において、テストは品質を保証するための重要な工程である。

テストは、図 1 のような流れで行う場合がある。開発チームが作成したプログラムを品質チェックチームがチェックし、その品質が悪い場合はもう一度テストするように指示を出す。そのとき、積み上げるテストケースの件数の最低基準が提示されるが、そのあと積み上げたテストケースの大部分が重複していてはテストの意味があまりない。

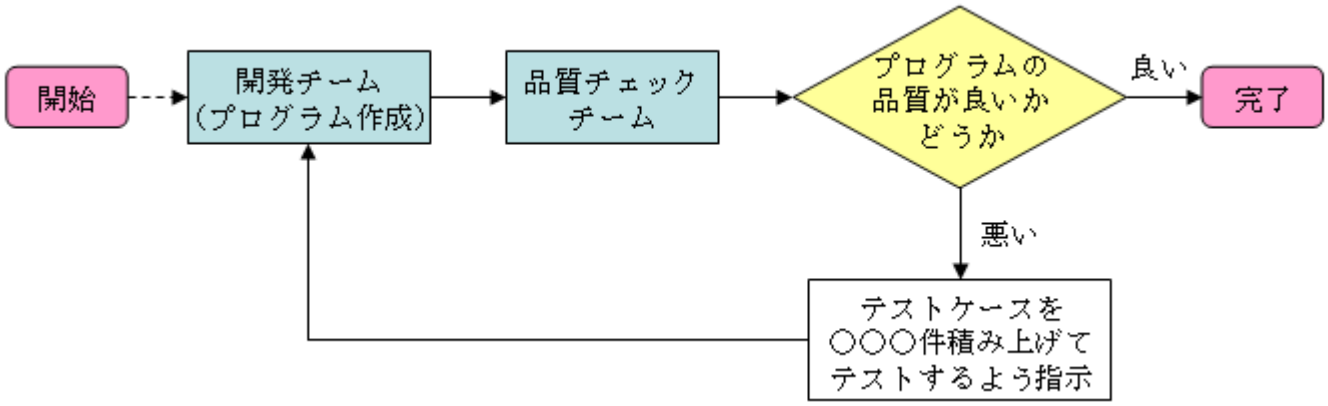


**図 1　テストの流れのフローチャート図**

テストでは、複数の入力と出力に対するテストケースを用意し、それらのテストカバレッジを一括して計測し、その計測値をテストの進捗評価に用いるのが一般的である。カバレッジとは、プログラムのテストを実施し、どの程度期待通りの動作をしたかを表す指標である。

現状のカバレッジ基準はテストケース全体のカバレッジを測ることは可能だが、個々のテストケースがどれだけ重複しているかを測ることは出来ない。入出力が異なるテストケース同士であっても、テストする部分の実行パスが同じであれば、テストケースの働きとしては同値になるので、それらのテストケースは重複しているといえる。同値分割がうまく実施されないと、カバレッジが高くても重複した単体テストケース多くなるので、テスト工程としての効率は低いと言える。そのため、カバレッジはテストケース設計作業の効率評価には利用できない。本稿では、テストケースの実行パス分析を行い、同値

分割を行うことにより、実行パス数をテストケース設計作業の効率の評価指標として用いることを提案する。

以下、2 章では問題提起、3 章では問題解決のための提案手法、4 章では実装法、5 章ではまとめと今後の予定について述べる。

## 2. 問題提起

この節では、カバレッジだけではテストの効率を測ることが難しいことを示す。まず 2.1 節ではテストの効率を測る際にカバレッジを用いる現状について説明し、2.2 節ではその問題点について述べる。

### 2.1. カバレッジ

カバレッジとは、「プログラムのテストを実施し、どの程度期待通りの動作をしたかを確認できた割合」を示す指標である。カバレッジツールはテストケース全体のカバレッジを測ることが出来る。カバレッジの例として分岐網羅基準を用いたテストでは、テストケースを走らせてパスした分岐の数を計測し、以下の基準によってカバレッジを算出する。

$$\text{分岐網羅率} = \frac{\text{通過した分岐数}}{\text{全分岐数}} \times 100$$

### 2.2. カバレッジを用いる際の問題点

既存のカバレッジツールでは、テストケース全体の網羅率を測ることは出来るが、個々のテストケースがどれだけ重複しているかを測ることは出来ない。大量にあるテストケースの中から同値類を判別することは、カバレッジツールでは不可能である。つまり、カバレッジが 100%になったとしても、使用したテストケースが重複している場合がある。例として、図 2 の引数の値によって出力を変えるプログラムの単体テストを考える。

```
0: Public static void m (int a) {
1:     if (a>0) {
2:         System.out.println ("aは正の数です.");
3:     } else if (a<0) {
4:         System.out.println ("aは負の数です.");
5:     } else {
6:         System.out.println ("aは0です.");
7:     }
8: }
```

**図 2　数値判定プログラム**

図 2 のプログラムの場合、例えば 3 を入力として与えるテストケースは 1 行目の条件に合い 2 行目が実行される。これは 2 と 1 を与えるテストケースと同じ条件となる。つまり、これらのテストケースは、テスト対象となるプログラムの実行パスが同一になるため、テストケースの働きが重複する。しかし、同値分割の視点で考えればテストケースとしての働きが異なる場合もある。このため、実行パスが同一なテストケース同士は、常にテストケースの働きが重複しているとまでは言い切れない。従って、テストケースの重複が、即、テスト効率を下げるとは言えないが、重複したテストケースが大量に書かれる状況は、多くの場合、テスト工程の効率の面から望ましくない。過度に重複したテストケースの実行は

同じ部分を繰り返しテストすることであり、ホワイトボックステスト[1]の技術である同値分割や境界値分析の視点では効率が悪いといえる。このため、カバレッジを計測するほかに、境界値分析の視点でテストケースの重複を判断することにより、テストの効率を計測するような指標が必要である。

## 3. 提案手法

本稿では 2 章で述べた問題点を解決するために、個々のテストケースの実行パスを取得し、そのパスを比較することによってテストの効率を測る新たな指標を提案する。ここで実行パスとは、プログラムのどこを実行したかを示す情報であり、本稿では実行されたメソッド中の基本ブロックの集合とする。基本ブロックとは分岐を持たないコードのかたまりである。実行パスを取得するためには、コントロールフローグラフを用いる。例として、2 章であげた例題プログラムのコントロールフローグラフを図 3 に示す。

図 3 でのノードはメソッド中の基本ブロックを表す。図中の数値は基本ブロックが対応するソースコードの行番号である。エッジはプログラムの制御の流れを表す。例えば 3 を入力したテストケースの実行パスは{1,2,7}であり、2、1 を与えたテストケースと同じパスになる。それぞれのテストケースで実行パスを計測し、それを比較することでテストケースの重複を調べることができる。

本研究では、各ブロックの先頭に実行パス取得のためのコードを埋め込み、テストを実行した際に実行パスを記録するツールを開発する。

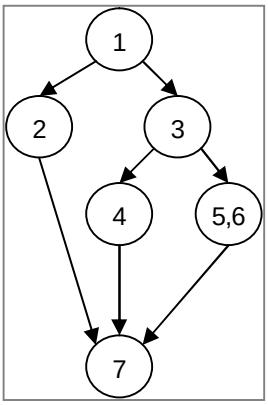


**図 3　図 2 の数値判定プログラムのコントロールフローグラフ**

## 4. 実装

本研究では Java を対象とし、実行パス分析を行うツールを Soot[2]を用いて実装する。Soot は、Java のバイトコードを分析して中間表現に変換、解析、最適化するためのフレームワークである。Soot のブロックグラフを用いると、プログラムを分岐でブロックにわけることが出来る。各ブロックの先頭に実行パス取得のためのコードを埋め込むことで、テストを実行した際に実行パスが記録される。実際に図 2 のプログラムで各ブロックの先頭にブロック番号を取得するコードを埋め込むと図 4 のようになる。

```
 0: Public static void m (int a) {
 1:    print(“ブロック番号：0”);
 2:    if (a>0) {
 3:       print(“ブロック番号：1”);
 4:       System.out.println (“aは 正の 数です.”);
 5:    } else {
 6:       print(“ブロック番号：2”);
 7:       if (a<0) {
 8:          print(“ブロック番号：3”);
 9:          System.out.println (“aは 負の 数です.”);
10:       } else {
11:          print(“ブロック番号：4”);
12:          System.out.println (“aは 0です.”);
13:       }
14:    }
15:    print(“ブロック番号：5”);
16: }
```

**図 4　コードを埋め込んだ図 2 の数値判定プログラム**

**このプログラムによって得られた実行パスをテスト効率の評価指標として用いる。**

## 5. おわりに

**本稿では、カバレッジではテスト効率の評価指標としてテストケースの実行パスを利用する手法を提案した。現在、実装はブロックグラフを作成しブロックの先頭にクラス名、メソッドシグニチャ、ブロック番号を表示することが出来ている。今後は、実行パスの重複を調べる部分を作成する。**

## 参考文献

[1] Glenford. J. Myers, **松尾正信[訳], “ソフトウェアテストの技法”, 近代科学社,**
[2] Vallée-Rai, et. al. “Soot - a Java Optimization Framework” in CASCON99