Statement Deletion Operator を用いたミューテーション解析ツールの作成と実用のための考察

Creating a mutation testing tool with Statement Deletion Operator and consideration about mutation testing

テクマトリックス株式会社
TechMatrix Corporation
○野中亮
○Ryo Nonaka

This report is written about mutation testing. One of major problems about software testing is evaluation of test suites. Test coverage is widely known as a method of evaluating but it cannot measure bug detection ability. So created a mutation testing tool for measuring bug detection ability and consider about mutation testing.

1. はじめに

ソフトウェア開発において，いかに不具合を出さずに高品質なソフトウェアにするかというのは重要な課題になっている．一般的にソフトウェアの品質を担保するためのプロセスとしてソフトウェアテストがある．ソフトウェアテストはバグを見つけるには効果的だが，やろうと思えば無限に行うことができ，どこまでやるかが難しいということがしばしば問題となる．そのため，何かしらの基準で十分と思われる種類のテストケースをテストスイートとしてソフトウェアテストを行うことになる．しかし，テストスイートを評価するための手段が少なく，どの程度のテストを行えば十分かを判断するのはとても困難である．

一般的にテストスイートを評価する方法としてカバレッジ[1]が広く知られている．カバレッジとはソースコードのうちどの範囲がテストされたかを計測する手法である．カバレッジはいくつかの種類が考えられており，どれだけ命令文をカバーしているかを計測するステートメントカバレッジや，条件分岐をどれだけカバーしているかを計測するブランチカバレッジ等がある．なお，本レポートでは以降，カバレッジとはステートメントカバレッジのことを指すこととする．カバレッジを 100%にすることで全てのソースコードがテストされたことを保証し，テストスイートが一定の水準を満たしていることを評価することが可能である．しかし，カバレッジでも不十分な面がある．カバレッジの計測では，テストスイートがどの程度ソースコードをカバーしているかを計測することは可能だが，バグ検出の性能を評価することができない．例えば，図 1 のような足し算を行うメソッドがあるとする．これはバグが含まれており，常に 1 を返してしまう．

†テクマトリックス株式会社 システムエンジニアリング事業部
ソフトウェアエンジニアリング技術部 ソフトウェアエンジニアリング技術 1 課
TechMatrix Corpration SOFTWARE ENGINEERING TECHNICAL SECTION No.1

東京都港区三田 3-11-24 国際興業三田第 2 ビル Tel:03-4405-7856 email:nonaka@techmatrix.co.jp
3-11-24, MITA, MINATO-KU, TOKYO

```
public int add(int a, int b) {
    int res = 1;
    int ans = a + b;
    return res;
}
```

図 1 add メソッド

このメソッドをテストするためのテストスイートとして図 2 のテストスイートがあるとする.

```
public void testadd() {
    int ret = add(1, 0);
    assertEquals(1, ret);
}
```

図 2 testadd メソッド

このテストではカバレッジが 100%になるが，バグを見つけることは出来ない．このようにカバレッジではテストがどの範囲で行われたのかを評価することは可能だが，バグ検出能力の評価を行えないことが分かる.
そこで近年研究されている手法にミューテーション解析[2]という手法がある．ミューテーション解析はテストスイートのバグ検出能力を評価するための手法である．本研究では実際にプロジェクトに適用可能なミューテーション解析を行うツールを作成し，GitHub 上のプロジェクトに対して実行することで，実際のプロジェクトに適用するための検討を行うことを目的とする.

2. ミューテーション解析

ミューテーション解析とは，意図的に改変を加えたソースコードに対してテストを実行し，改変を検出できるかどうかでテストスイートの評価を行う手法である．ミューテーション解析を行うにはテスト実行可能な環境が必要となる．テスト対象のソースコードと実行可能なテストスイートがあればミューテーション解析が可能だ．また，ソースコードに変更を加える必要があるため，テスト対象のソースコードを変更できることも条件となる．ミューテーション解析は以下の手順で行う.

1. ソースコードに改変を加えたミュータントを作成する
2. ミュータントに対してテストを実行し，テストが失敗すればミュータントをキル(kill)したと呼ぶ
3. 作成した全てのミュータントに対してテストを行ったら，作成したミュータントの数とキルしたミュータントの数を集計する
4. 作成したミュータントのうちキルできた割合をミューテーションスコアと呼ぶ

つまり，ミューテーションスコアが高ければ高いほど，バグ検出能力の高いテストスイートと言える．バグ検出能力を評価する手法として便利だが，ミューテーション解析の課題としては，必要な工数が大きいことがある．作成したミュータントごとに全てのテストを実行するため，多大な時間が必要となる．そのため，良いミューテーション解析を行うためには，いかに少ないミュータントで，より効果的なバグを埋め込めるかが重要と言える．他にも等価ミュータントという問題がある．これはミュータントが元のソースコードと論理的に等価であることを指す．バグを埋め込んだのにバグにならないので評価するにあたり等価ミュータントをどうするか考える必要がある．等価ミュータントを考慮したミューテーションスコアは以下の式により求められる.

$$score(\%) = \frac{k}{m-e} * 100$$

k ： キルしたミュータントの総数

m : 生成したミュータントの総数

e : 等価ミュータントの総数

一般的にミュータントは機械的に全て作成する．そのため，機械的にミュータントを作成するための様々な手法が考えられている．それらの手法のことをミューテーションオペレーターと呼ぶ．例えばその一つである CONDITIONALS_BOUNDARY は表 1 のように条件文中の不等号を変更するミューテーションオペレーターである．

表 1 CONDITIONALS_BOUNDARY オペレーター

オリジナルコード	ミュータント
<	<=
<=	<
>	>=
>=	>

他にも数学の演算子を他のもの変更するものや，数値を全てゼロにするものなどがある．具体的にミューテーションオペレーターを適用してミュータントを作成する例を考えてみる．図 3 のメソッドに対して CONDITIONALS_BOUNDARY を適用することで，図 4，図 5 などのミュータントが作られる．

```
public string checkScore(int score) {
    string res  res;
    if (score < 50) {
        res = “LOW”;
    } else if(score < 80) {
        res = “MIDDLE”;
    } else {
        res = “HIGH”;
    }
    return res;
}
```

図 3　ミューテーション前のコード

```
public string checkScore(int score) {
    string res  res;
    if (score <= 50) {
        res = “LOW”;
    } else if(score < 80) {
        res = “MIDDLE”;
    } else {
        res = “HIGH”;
    }
    return res;
}
```

図 4　不等号が変更されたミュータント 1

```
public string checkScore(int score) {
    string res  res;
    if (score < 50) {
        res = “LOW”;
    } else if(score <= 80) {
        res = “MIDDLE”;
    } else {
        res = “HIGH”;
    }
    return res;
}
```

図 5 不等号が変更されたミュータント 2

このようにミューテーションオペレーターを機械的に適用して複数のミュータントを作成し，ミュータントごとにテストを実行する．もし 2 つのミュータントに対してテストを実行し，1 つ

だけテストが失敗，つまりミュータントをキルした場合のミューテーションスコアは 50 となる．先述したが，ミューテーション解析の研究では解決する課題として，より生成されるミュータントの数を少なくしつつ，バグ検出能力を落とさないミューテーションオペレーターの作成がある．例えば，全ての条件式と数学演算子を変化させるミューテーションオペレーターを考えると，作成されるミュータントは膨大な数になる．それぞれに全てのテストを実行すると完了までに膨大な時間がかかってしまうというのは実用化のためには大きな障害となる．その課題の解決策の一つとして Deng Lin らは，2013 年に statement deletion operator を適用したミューテーション解析について発表した[3]．Deng Lin によると，ミューテーションオペレーターとして，statement deletion operator(SDL)のみを適用したミューテーション解析で課題の改善が見込めると言う．発表では SDL でのミューテーション解析と，他のミューテーションオペレーターを適用した既存のツールとの比較を行っている．SDL とはプログラムのステートメントを削除するミューテーションオペレーターである．その結果，作成されるミュータントの数が 80%削減されたとのことだ．また，SDL を使用したミューテーション解析のミューテーションスコアを 100 とするテストスイートを使って既存ツールでのミューテーション解析を行ったところ，ミューテーションスコアは 92 となり，バグ検出能力の評価においても大きく劣らないという結果となっている．さらに，等価ミュータントが 41%削減されたという結果も出ている．以上より SDL を使用したミューテーション解析は以下の特徴があることが言える．

- 生成されるミュータントの数が少ない
- バグ検出能力の評価は既存のツールに引けをとらない
- 等価ミュータントが生成されにくい

これらの SDL の特徴はミューテーション解析の問題を軽減していることを表している．それを踏まえ，本研究では SDL を使用したミューテーション解析ツールを作成し，実際のテストスイートの評価に適用することにした．

3. 作成したミューテーション解析ツール

ツールの対象言語として，今回は JAVA を選択した．また，JAVA の中でも Maven プロジェクトに限定している．以下の仕様のツールを C#で作成した．

- プログラムの実行は引数にテスト対象の Maven プロジェクトのルートディレクトリのパスを渡して実行する
- Maven プロジェクトのソースコードに対して，SDL を適用したミュータントを作成する
- 各ミュータントに対して mvn test コマンドでテストを実行する
- プログラムを実行すると，作成されたミュータントの数，キルしたミュータントの数，ミューテーションスコアを表示する
- オプションとして-c を指定すると CSV 形式で結果を出力する
- オプションとして-r <0-100>を指定するとミュータントを作成する割合を指定できる
  例えば 50 を指定すると本来生成されるミュータントのうちランダムに選択した半分となる
- 等価ミュータントについてはノイズとして無視することとする

作成したツールを実際に maven プロジェクトに適用し，問題なく SDL を適用したミュータントが作成されていることを確認した．以下に作成されるミュータントの例を示す．図 6 のコードに適用すると，図 7-9 のミュータントが生成される．

```
public class Main {
    public  static  void  main(String[]
args) {
        int i=0, j=0;
        i=2;
        j=3+i;
```

```
        System.out.println(j);
    }
}
```

図 6 ミューテーション前のコード

```
public class Main {
    public static void main(String[]
args) {
        int i=0, j=0;
        {}
        j=3+i;
        System.out.println(j);
    }
}
```

図 7 SDL が適用されたミュータント 1

```
public class Main {
    public static void main(String[]
args) {
        int i=0, j=0;
        i=2;
        {}
        System.out.println(j);
    }
}
```

図 8 SDL が適用されたミュータント 2

```
public class Main {
    public static void main(String[]
args) {
        int i=0, j=0;
        i=2;
        j=3+i;
        {}
    }
}
```

図 9 SDL が適用されたミュータント 3

このように，命令文を意味のないブロックに置き換えたミュータントが作成される．ただし，変数宣言などのコンパイルが通らなくなるような箇所は削除しない．対象のステートメントが制御ブロックの場合は，ブロック中の各ステートメントを削除するミュータントとブロック全体を削除したミュータントを生成する．また，Maven プロジェクトに実行した結果，図 10 のようにミューテーションスコアが出力され，ツールが動作していることが分かる．

```
Statements: 14
Minutes (max: 60): 1 (4384[ms] * 14) /
FizzBuzz.java:[10/10]
Main.java:[4/4]
C:¥work¥topse¥MavenSample¥FizzBuzz: 10 /
14: 71%
```

図 10 ツール実行結果

4. プロジェクトへの適用

作成したツールを使用し，実際のプロジェクトに適用して結果を計測した．今回は GitHub からテスト実行可能な Maven プロジェクトを 41 プロジェクト取得し，ミューテーション解析を実施した．その中でミューテーションスコアが 0 ではなかった 17 プロジェクトの結果を表 2 に示す．

表 2　17 プロジェクトの実行結果

No.	url	ミュータント生成数	キル数	ミューテーションスコア
1	https://github.com/spotify/tri	64	56	87
2	https://github.com/square/pol	261	220	84
3	https://github.com/lviggiano/o	380	317	83
4	https://github.com/JakeWhart	96	76	79
5	https://github.com/myabc/ma	260	200	76
6	https://github.com/kungfoo/g	268	196	73
7	https://github.com/square/mi	146	103	70
8	https://github.com/spring-projects/spring-petclinic	210	114	54
9	https://github.com/VerbalExpressions/JavaVerbalExpression	87	45	51
10	https://github.com/LMAX-Exchange/LMAXCollections	52	26	50
11	https://github.com/peter-lawrey/Java-Thread-Affinity	316	86	27
12	https://github.com/octo-technology/sonar-objective-c	207	57	27
13	https://github.com/marcelove	186	41	22
14	https://github.com/WhisperSy	151	29	19
15	https://github.com/fearofcode	861	133	15
16	https://github.com/rtyley/android-screenshot-lib	61	7	11
17	https://github.com/Vektah/CodeGlance	466	41	8

また，SDL の効果を確認するために，フリーのミューテーション解析ツール[4]の実行結果とバグ検出能力と他のメトリクスとの関係性の考察を行うためにサイクロマチック複雑度や行カバレッジなどのメトリクスを計測した．結果を表 3 に示す．

表 3 SDL ではないミューテーション解析の結果とメトリクスの計測

No.	ミュータント生成	キル数	ミューテーションスコア	行数	行カバレッジ	複雑度
1	455	424	93	1494	95.4	1.1
2	866	746	86	1043	88.6	3.6
3	1547	1363	88	3766	95.6	1.7
4	321	294	92	216	93.4	5.9
5	1166	930	80	1487	88.1	2.3
6	1607	1219	76	1282	90.1	2.1
7	290	226	78	527	83.1	2
8	716	359	50	2950	61.4	1.4
9	354	282	80	734	81.1	1.6
10	177	128	72	521	50.7	1
11	N/A	N/A	N/A	2031	48.1	2.2
12	813	257	32	2368	37.2	1.5
13	N/A	N/A	N/A	1154	4.4	1.9
14	832	166	20	2411	48.9	2
15	N/A	N/A	N/A	2855	29.2	3.5
16	246	25	10	425	8.7	1.6
17	1697	182	11	2047	14.7	2.8

5. 考察

プロジェクトの適用結果から，公開されているプロジェクトにおいても，ミューテーションスコアは 4 から 83 とかなり大きな差があることが分かった．ミューテーションスコアがいくつ以上であれば良いテストスイートである，との絶対的な数値は現段階では判定できないが，比較してこれだけ差があるということは，ミューテーション解析を利用してテストスイートを改善できる余地が大きいと言える．これまでテストスイートのバグ検出能力を計測する手法がなかったことからこれだけ大きな差が存在すると考えられる．ミューテーションスコアを品質の指標として使用することで，これまで考えられなかったテストのバグ検出能力の改善を考えられるようになり，回帰テストが重要な派生開発などにおいては，よりソフトウェアの品質向上に繋がると考えることができる．次に SDL を使わないミューテーション解析の結果と比較を行ったところ，表 4 のような結果となった．

表 4 SDL を使わないミューテーション解析との比較

	SDL	既存ツール	備考
平均ミュータント生成数	193.5	791.9	76%削減
平均ミューテーションスコア	55.1	62.0	88.9%

表 4 から，SDL オペレーターを使用することで，生成されるミュータントの数が 76%の削減となっているが，ミューテーションスコアの減少は 11%程度にとどまっていることが分かる．この結果は参考にした Deng Lin らの論文とほぼ同様の結果となっている．実行時間はミュータントの数に依存するため，ミューテーション解析の実行時間が 76%削減されることになる．ミューテーションスコアが減っていることから，バグ検出能力の評価は少し劣っているが，削減数に比べると小さくなっている．これにより，実際のプロジェクトへ適用しやすさが既存の手法に比べ大きく向上していると考えられる．ただし，評価能力は既存ツールの方が高くなるため，実際には状況によって使い分けるか，ミューテーションオペレーターを組み合わせてバランスをとるような方法を考える必要がある．また，行カバレッジとの比較をすると，カバレッジが高くなるプロジェクトはミューテーションスコアも高くなることが分かる．これにより，広く使われているカバレッジと同様に，ミューテーション解析もテストスイートの評価方法として使用できると言える．今回はカバレッジとの組み合わせを検討していないが，今後研究を進める際には，例えばカバレッジ 100%とミューテーションスコア 100 になるテストスイートをそれぞれ準備して，とれるバグにどのような違いがあるかの検討なども行いたい．

実際にプロジェクトへ適用してみたところ，ミューテーション解析を実用的に使用するためには以下のような使い方が考えられる．

- ミューテーションスコアをカバレッジと共にテストのエビデンスとして使用する
- ミューテーションスコアが減らないようにテストケースを減らすことで，テストケース削減を効率的に行う
- ミューテーションスコアが増えるパターンのテストケースを追加することでテストスイートの品質を向上する

ミューテーションスコアはこれまで数値化できなかった部分を，目に見えるメトリクスとして提供できるようになったことで，これまで論じることのできなかった様々な部分へ使用することが考えられる．まずはテストの品質の担保として使用することができる．これまで網羅率でのみ担保していたテストの品質をバグ検出能力も含めた結果として提示できる．今回作成したステートメント削除のミューテーション解析においては，ミューテーションスコアを 100 にするためには全てのステートメントをテストで通過する必要があり，自ずと行カバレッジ 100%も達成すること

が可能だ．また，テストスイートのバグ検出能力数値化することが出来るようになったため，バグ検出能力を下げることなくテストケース数を削減することも可能となる．テストケースを削減することはテスト時間の削減につながり，回帰テストにおいて工数の削減を見込むことが可能となる．効率化以外にも，テストスイートのバグ検出能力を向上させるためにどんなテストケースを追加すればよいかが論理的に判断できるようになる．これまで一部の熟練者の知識によっていた部分を，どんな人が行っても一定の品質まで向上することが可能となる．

一方，実際にツールを作成して使用した結果，課題も見えてきた．まず大きな課題は解析の実行時間である．今回は GitHub よりプロジェクトを選択する際に，小さなプロジェクトのみを選んだため，長くても SDL では 10 数分，SDL ではないミューテーション解析でも 1 時間弱で完了した．ただし，生成されるミュータントが少ない SDL を使用した場合でも，ステートメントの数だけのミュータントが生成されるため，仮にテストに 2 分かかる 1000 ステートメント程度のプロジェクトに適用した場合でも，解析時間は 2000 分必要となり，1 日以上かかってしまう．実際の現場では何万ステートメントものプロジェクトも存在するため，解析時間の短縮は実用のためには必要となる．

6. おわりに

最後に今後の展望と本論文のまとめを行う．今後の展望としては以下のようなことが考えられる．

- 対応言語の拡張
- 解析時間を減らすために並列実行処理
- 実際の開発プロジェクトへ適用し，適用効果の測定

本論文では完了した GitHub 上のプロジェクトへの適用しか行っていないが，今後の実用の検討として，開発中のプロジェクトへ適用し，例えば適用した場合としなかった場合の品質の差を測定することでより実用への検討が行えると考える．また，テストスイートの評価手法だけではなく，例えば他にも，機械的にミューテーションスコアを 100 にするテストセットを作成することで実用的なテストケースの自動生成や，ミューテーションスコアが減少しないテストケースを除外することで，回帰テストの実行時間の短縮などにも応用が可能かも検討の余地がある．

本論文ではミューテーション解析を実際に行い，実プロジェクトへ適用できることを確認し，実用への課題や有用そうな効果についての検討を行った．まだ実用までの課題や効果的な適用方法は検討が必要であるが，これまで計測できていなかったテストスイートのバグ検出能力を数値化できる手法は，様々な用途へ適用が可能だと考えられる．実際のプロジェクトへ適用結果を見てもミューテーションスコアにばらつきがあるように，この数値を使うことでソフトウェアの品質を向上できると考えられる．

参考文献

[1] 大塚俊章, and 荻野富二夫. "ソフトウェアテスト技術." *ユニシス技報, 日本ユニシス*27.2 (2007): 70-88.

[2] DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on test data selection: Help for the practicing programmer." Computer 11.4 (1978): 34-41.

[3] Deng, Lin, Jeff Offutt, and Nan Li. "Empirical evaluation of the statement deletion mutation operator." In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 84-93. IEEE, 2013.

[4] http://pitest.org/