ミュータント削減手法の高信頼化に向けて

徳本 晋[1,2,3,a)]　本位田 真一[2,3,b)]

概要：ミューテーションテストの高速化のためにミュータント削減に関する研究が多く行われている．ミュータント削減量の限界を解析することは重要な研究トピックの 1 つである．しかし既存手法は過剰なミュータント削減と削減後のミューテーションスコアの誤差という点で課題を残している．
本研究ではミューテーション解析中に引き起こされるエラーの種類を網羅するようなミュータントを選択することと，選択されたミュータントに重み付けをすることで過剰なミュータント削減を防ぐ高信頼なミューテーションスコアの計測方法を提案する．OSS を用いて評価した結果，既存手法よりも削減量は劣るものの，貪欲法によるミュータント選択方法により約 40%の実行時間の削減効果が得られた．また，過剰なミュータントの削減の影響を評価するために，バグ検出力を人為的に落としたテストに対してミューテーションスコアを計測したところ，既存手法に比べ提案手法のミューテーションスコアの誤差は少なくなった．

1. はじめに

ミューテーションテストはミュータントと呼ばれるバグが埋め込まれたプログラムを網羅的に生成し，それらに対しテストスイートを実行しバグをどれだけ検出できたかによって，そのテストスイートのバグ検出力を測定する方法である．ミュータントにテストを実行したときに失敗すれば，そのテストによってミュータントは殺されたと言う．殺されたミュータントの割合（ミューテーションスコアと呼ぶ）が多い程，バグ検出力の高いテストスイートと考えられる．

ナイーブなミューテーションテストではそれぞれのミュータントごとにテストを実行する必要があるので，実行時間はテスト実行時間とミュータントの数の積によって決まる．ミュータントの数はプログラム規模に比例して多くなるため，ミューテーションテストをスケールさせるためにミュータントの数を減らす方法が多く提案されている．ミュータント数を削減するアプローチで重要なのは，ミュータント削減後もミュータント削減前とほぼ同等のバグ検出力測定ができること，つまりバグ検出力を表すミューテーションスコアがほぼ変わらないことである．例えば，ミュータント削減前はミュータント 10 個中 6 個がテストで殺すことができた（つまりミューテーションスコアは 60%）としたときに，ミュータント削減後にミュータント 5 個中 5 個がテストで殺すことができてしまう（つまりミューテーションスコアは 100%）と，本来のミューテーションスコアである 60%からかけ離れた値を示すことになる．これはミュータント削減により正確なバグ検出力測定ができなくなってしまうことを意味する．正確ではないバグ検出力の結果を知らされた人は，プログラム中に潜在するバグをテストによってつぶし切れてないにもかかわらず，テストが十分されていると誤った認識をしてしまう可能性がある．またはその逆で，潜在バグをテストでつぶし切れているにもかかわらず，テストが不十分と誤認識する可能性もある．

Ammann ら [1] のミュータント削減モデルでは，同じテストケースの組合せによって殺されるミュータント同士はバグ検出力の測定に貢献しないため冗長だと見なし，1 つを残して削除する．しかし，ミュータントの冗長性を測る単位として，テストケースごとでの分類では粒度が粗い場合がある．ユニットテストにおける 1 テストケース相当となるテストメソッドでは，複数のテスト対象メソッドが呼ばれ，複数のアサーションによって結果のチェックを行い，1 テストケース内で多様な性質をテストしているものも多く存在する．このようなテストケースで殺されるミュータントの集合は同質なものとは見なさない方がより自然である．特に 2000 年代前半から提唱されているテストコードへのリファクタリング [2] が近年認知され始めてきてお

[1] 株式会社富士通研究所
FUJITSU LABORATORIES LTD., Kawasaki, Kanagawa 211–8588, Japan
[2] 早稲田大学
Waseda University, Shinjuku-ku, Tokyo 162–0042, Japan
[3] 国立情報学研究所
National Institute of Informatics, Chiyoda-ku, Tokyo 101–8430, Japan
[a)] tokumoto.susumu@fujitsu.com
[b)] honiden@nii.ac.jp

り，1 テストケース内に複数のアサーションを含むアンチパターンはアサーションルーレットと呼ばれ，保守時のプログラム理解に悪い影響を与える [3, 4] ことから，リファクタリングすることが推奨される．さらにテストコード中のアサーションのうち，45.7%のアサーションがバグの検出に貢献しないという実験結果 [5] からも，テストコード内のアサーションの整理の必要性がわかる．アサーションの削除などのテストメソッド内の命令を修正した場合に，その影響を把握できるミュータントを残すことは重要である．

また削減されたミュータント集合は元のミュータント集合と異なる数となるため，ミューテーションスコアを求めようとすると，分母となる総ミュータント数も分子となる殺されるミュータント数も元のミュータント集合と異なることになり，本来のミューテーションスコアとの間に誤差が生じる．

本論文ではミュータントの冗長性を判断する方法として，ミュータントによって引き起こされるエラーの種類に着目し，エラーの種類が重複するミュータントは冗長と見なすことで，ミュータントを削減するモデルを提案する．また，削減されるミュータントの影響を計算し，残すミュータントに重み付けすることで，元のミュータント集合のミューテーションスコアとの誤差を減らした．

その効果として，テストコード中の命令がリファクタリングなどで修正または削除された場合でも，元のミュータント集合と同等なバグ検出力の測定が可能になることが期待できる．また OSS の 53 プロジェクトに対してミュータントの削減を試み，ミュータント数，実行時間について評価を行った．さらにその中の 3 つのプロジェクトに対してテストコード内のアサーションを一定の割合で削除してバグ検出力を下げたときに提案モデルと既存モデルでどれだけ削減前と近いバグ検出力を測定できるかを評価した．

我々の先行研究 [6] でも同じようにエラーに着目したミュータント削減手法を提案しているが，本論文での手法は先行研究で考案した冗長性とは異なる冗長性を定義することと，新たに考案したミュータントへの重み付けによって，先行研究で発生していた削減前のバグ検出力との乖離の問題を解決した．また，先行研究からさらに実験を拡張し，手法の妥当性をより広く確認している．

本論文の構成は，2 章で背景を紹介し，3 章で動機付けの例を示し，4 章で提案手法を説明する．5 章で評価方法と結果について示し，6 章でその内容について議論し，7 章で関連研究を紹介する．

2. 準備

ミューテーションテストのアルゴリズムは，基本的にはミュータントの生成と，それぞれのミュータントにおけるテストの実行の大きく 2 つに分かれる．ミュータントの生成は，例えば比較演算子を反転させるなどといった，ある変換規則（ミューテーションオペレータと呼ぶ）を，元のプログラムの変換対象プログラム要素それぞれに対し適用し，それぞれミュータントとして記録していく．つまりプログラム中の変換対象プログラム要素の数だけミュータントが生成されることになる．どのようなミューテーションオペレータが利用できるかは言語やツールによって異なり，どのミューテーションオペレータを用いるかはミューテーションテストの実行前にユーザが与えるものである．その後，記録したミュータントそれぞれにおいてテストを実行し，テストが失敗すればそのミュータントは殺すことができると見なす．すべてのミュータントのうち殺せたミュータントの割合をミューテーションスコアと呼ぶ．

Offutt ら [7] はミューテーション解析の高速化の戦略を *do fewer*, *do smarter*, *do faster* の三種類に分類した．“do fewer” はミュータントを減らす戦略，“do smarter” は分散させて計算させるなどの戦略，“do faster” はミュータントの生成・テスト実行を早くする戦略になる．今回注目するのは “do fewer” のアプローチである．

Ammann ら [1] によるテストケースベースのミュータント削減モデルでは，ミューテーションスコアを維持できる最小のテスト集合を以下のように定義する．

定義 1. 元のテスト集合 T とミュータント集合 M に対し，テスト集合 $\hat{T}_M$ が最小とは，任意のテスト $test_i \in \hat{T}_M$ において $\hat{T}_M - \{test_i\}$ がミューテーションスコアを維持できないときである．

これはつまりミューテーションスコアの向上に貢献しないテストのことを冗長と見なしていることになる．

その最小テスト集合を用いて冗長なミュータントを以下のように定義する．

定義 2. あるミュータント $m_j \in M$ に対し，$M_j = M - \{m_j\}$ とする．ミュータント集合 M の最小テスト集合 $\hat{T}_M$ と，ミュータント集合 M_j の最小テスト集合 $\hat{T}_{M_j}$ が等しいとき，つまり $\hat{T}_M = \hat{T}_{M_j}$ のとき，m_j はテスト T に対して冗長という．

これはテスト集合の最小化に影響しないミュータントを冗長と見なすことになる．言い換えると，あるミュータントが削除されたときに，対応するテストが最小ではなくなる場合は，そのミュータントは冗長ではないと考える．

さらに最小ミュータント集合を以下のように定義する．

定義 3. ミュータント集合に冗長なミュータントが存在しないとき，そのミュータント集合は最小である

3. 動機付けの例

ミュータントの削減モデルを考慮するにあたり，既存の手法では問題となる例を示し，新たな手法を考案する必要性について論じる．ここでは Apache Commons Lang の isEmpty メソッドに対するテストについて考える．

以下のリスト 1 で示すテスト対象メソッド isEmpty() において生成されるミュータントは比較演算子を反転するミューテーションオペレータによるものが 2 つと，論理和の左辺・右辺をそれぞれ false に置き換えるミューテーションオペレータによるものが 2 つの，計 4 つとする．リスト 2 の m_1 とリスト 3 の m_2 が論理和の左辺に対するミュータントになり，同様にリスト 4 の m_3 とリスト 5 の m_4 が論理和の右辺に対するミュータントになる．

リスト 1: ソースコード例（StringUtils.java）

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || cs.length() == 0;
3 }
```

リスト 2: ミュータント例 m_1

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs != null || cs.length() == 0;
3 }
```

リスト 3: ミュータント例 m_2

```
1 public static boolean isEmpty(CharSequence cs) {
2   return false || cs.length() == 0;
3 }
```

リスト 4: ミュータント例 m_3

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || cs.length() != 0;
3 }
```

リスト 5: ミュータント例 m_4

```
1 public static boolean isEmpty(CharSequence cs) {
2   return cs == null || false;
3 }
```

そのときリスト 6 で示すテストメソッドによって全てのミュータントは殺すことができる．しかし，テスト失敗を引き起こす命令はテストコード中の 2-4 行目になり，仮に 2 行目以外の命令を削除した場合，全てのミュータントを殺すことはできない．

リスト 6: テストコード例（StringUtilsTest.java）

```
1 @Test public void testIsEmpty() {
2   assertTrue(StringUtils.isEmpty(null));
3   assertTrue(StringUtils.isEmpty(""));
4   assertFalse(StringUtils.isEmpty("␣"));
5   assertFalse(StringUtils.isEmpty("foo"));
6   assertFalse(StringUtils.isEmpty("␣␣foo␣␣"));
7 }
```

テストケースベースの削減モデルでは，最小テスト集合がリスト 6 のテストメソッドとなり，4 つすべてのミュータントは殺せるため，4 つのうち 1 つのミュータント以外は冗長と見なし削減される．

ここでリスト 6 のテストメソッドの命令のうち 2 行目のみを残し，バグ検出力を弱めたテストメソッド（リスト 7）に対するミューテーションスコアを考える．

リスト 7: バグ検出力を弱めたテストコード例

```
1 @Test public void testIsEmpty() {
2   assertTrue(StringUtils.isEmpty(null));
3 }
```

バグ検出力を弱めたテストメソッドでは，4 つのミュータントのうち，m_1, m_2 の 2 つしか殺すことができない．つまり，ミューテーションスコアは 50%に落ちることを意味する．一方，テストケースベースの削減モデルにより 4 つのミュータントのうち m_1 のみを残して削減されたとすると，このミュータントはバグ検出力を弱めたテストメソッドでも殺せるため，ミューテーションスコアは 100%のままとなり，削減前のミューテーションスコアと差異が生じてしまう．

このようにテストケース単位ではなく，テストメソッド内の命令単位での影響まで考慮しないと，バグ検出力の正確な測定はできない場合がある．特にテストコードに対するリファクタリングにより命令単位での修正が発生すると，テストケース単位のミュータント削減ではバグ検出力の低下を検知できないことが考え得る．

4. 提案手法

3 章で示したように，テストメソッド内の命令単位の修正が起きたときにその修正によるバグ検出力，つまりミューテーションスコアの変化を検出できるミュータントの最小量を求めることを本手法の目的とする．

ミュータントによって引き起こされるエラーは，テストメソッド内の命令からテスト対象メソッドを呼び出し，その中の実行時に発生するか，実行後のアサーションによって発生する．エラー内のスタックトレースには，テストメソッド内の命令からエラー発生個所に至るまでの呼び出し階層が記録される．つまり，どのミュータントがテストメソッド内の命令に影響を与えているかは，そのミュータントが引き起こすエラーを観察することでわかる．

本論文で提案する手法は，ミューテーション解析において発生するエラーに対する冗長性を定義し，冗長なミュータントを削減することでミュータント数の最小化を図る．さらに削減されるミュータントと性質が近いミュータントに重み付けをすることにより，本来のミューテーションスコアとの誤差を減らすことを目指す．

注意したいのは，提案手法で扱うミュータントはすべてテスト集合 T によって殺されるものである．テスト集合 T によって殺されないミュータントについては，「どのテストで殺されたか」や「どのようなエラーが発生したか」といった情報が得られず，ミュータントとテストやエラーとの関連を調べることができないため，削減の対象外とす

る．削減モデルの傾向を知るという上では，統計的に十分な量の殺されるミュータントを準備できることが重要であり，テスト集合 T はそのような十分な量のミュータントを殺せるものを準備する．

4.1 定義

新たな手法のために 2 章で示した定義とは異なる “冗長なミュータント” を定義する．まずそのために，エラーについて定義する．

定義 4. エラーは例外の種類 ex とスタックトレース st の組合せで構成される．

例外の種類はエラー発生時にどのような事象が起きたかを判別するための情報で，例えば Java における NullPointerException が相当する．スタックトレースはエラー発生個所のメソッド呼出階層を記録したものである．

“同一のエラー” を次のように例外の種類とスタックトレースが同一のものと定義する．

定義 5. エラー err_i が発生したときの例外の種類 ex_i とスタックトレース st_i とエラー err_j が発生したときの例外の種類 ex_j とスタックトレース st_j が同一のとき，$err_i = err_j$ とする．ここでスタックトレースが同一とは，スタックトレースの各層における呼出メソッド名，ファイル名とその行番号のそれぞれが一致する場合である．

さらに “エラーが区別可能” を以下のようにエラー原因のミュータントとの関係で定義する．

定義 6. エラー err_i とエラー err_j が同一ではなく，かつ，err_i を発生させるミュータントの集合 M と err_j を発生させるミュータントの集合 M' が異なるとき，err_i と err_j は区別可能とする．またエラー集合 E に含まれる任意の異なる 2 つのエラー err_i と err_j が区別可能なとき，そのエラー集合は区別可能という．

区別可能なエラー以外を冗長なエラーと見なし，そのような冗長なエラーを削除する関数を定義する．

定義 7. 区別可能なエラー集合 $\hat{E} \subseteq E$ とその集合に含まれないエラー $err \in E \setminus \hat{E}$ に対し，$\hat{E} \cup \{err\}$ が区別可能ではない場合，err は冗長なエラーという．エラー集合 E の冗長なエラーを削除したものを $\mathcal{D}(E)$ とする．

冗長性を排したエラー集合を，区別可能なことを保ったままミュータントを削除することを考える．エラー集合が区別可能であれば，どのミュータントによってテストメソッド内の命令へ影響が及ぶかを判断できる．

定義 2 の考え方をベースとし，エラー集合 E に対する冗長なミュータントを次のように定義する．

定義 8. ミュータント集合 M に対し，テスト集合 T を実行したときに発生するエラーの集合を E_M とする．ミュータント集合 M のエラー集合 E_M から冗長なエラーを削除した $\mathcal{D}(E_M)$ と，ミュータント集合 M_j のエラー集合 E_{M_j} から冗長なエラーを削除した $\mathcal{D}(E_{M_j})$ が等しいとき，つまり $\mathcal{D}(E_M) = \mathcal{D}(E_{M_j})$ のとき，m_j はエラー集合 E において冗長という．

最小ミュータント集合についての定義である定義 3 において，冗長なミュータントが存在しないミュータント集合を最小としているため，定義 8 で定義した冗長なミュータントを削除することで，エラー集合 E に対する最小ミュータント集合のモデルを得ることができる．

4.2 ミュータント集合最小化アルゴリズム

本論文では，最小ミュータント集合を得る際に，実行時間の短いミュータントを選択することで，ミューテーション解析全体の実行時間の最適化を図る．具体的には，最初に各ミュータント m を実行し，各ミュータント m ごとのテスト実行時間 t_m の測定と発生するエラー集合 E_m の取得を行う．次に合計の実行時間を最小にするべく以下の最適化問題を解くことでエラーをすべて網羅する最短実行時間のミュータント集合 $\hat{M}$ が手に入る．

- 入力：殺されたミュータントの集合 M_{killed}，全エラー集合 $E_{M_{killed}}$
- 出力：ミュータント集合 $\hat{M}$
- 制約条件：$e_i \neq e_j (\forall e_i, e_j \in E_{\hat{M}})$
- 目的関数：$\sum_{m \in \hat{M}} t_m$
- ゴール：最小化

制約条件であるエラーの組合せは $\frac{|E_{\hat{M}}| \cdot (|E_{\hat{M}}|-1)}{2}$ になり，入力のミュータント集合 $\hat{M}$ の組合せは $2^{|\hat{M}|}$ となる．最適解を探索するには指数オーダーの計算量がかかることになるため，今回，発生させるエラーが少ないミュータントのうち一番実行時間が短いものを選択するような貪欲法で解くことで最小ミュータント集合を求めた．詳細をアルゴリズム 1 に示す．

アルゴリズム 1 貪欲法によるミュータント最小化

Input: M: ミュータント集合, E: エラー集合
Output: $\hat{M}$: 最小化ミュータント集合

```
M̂ ← φ
E_cover ← φ                                   ▷ 選択済みのエラーの集合
while E ≠ E_cover do
  M_minerr ← arg min_{m∈M\M̂} |E_m \ E_cover|   ▷ 未選択のエラーが最も少ないミュータント集合
  m' ← arg min_{m∈M_minerr} t_m               ▷ 最も実行時間が短いミュータントを選択
  M ← M \ {m'}
  if E_m' \ E_cover ≠ φ then
    M̂ ← M̂ ∪ {m'}
    E_cover ← E_cover ∪ E_m'
  end if
end while
return M̂
```

4.3 ミュータントの重み付け

前節で示した手法などで最小化したミュータント集合は

元のミュータント集合より数が減るため，そのままミューテーションスコアを算出しようとすると，母数の差異によりミューテーションスコアに誤差が発生する．そこで，削減されたミュータントが与える影響を残されたミュータントに重み付けしその値を基にミューテーションスコアを計算することで，疑似的に元のミュータント集合と母数を同じにしてミューテーションスコアの誤差を減らす．その重み付けの方法をアルゴリズム 2 に示す．

アルゴリズム 2 ミュータント重み付け

Input: M: ミュータント集合, $\hat{M}$: 最小化ミュータント集合
Output: $w_{m_1},...,w_{m_n}$: ミュータントの重みの集合
  $w_{m_1},...,w_{m_n} \leftarrow \{1,...,1\}$ ▷ 重みの初期化
  for $m_{removed} \in M \setminus \hat{M}$ **do** ▷ $m_{removed}$: 削除対象ミュータント
    $m_{nearest} \leftarrow \arg\min_{\tilde{m} \in \hat{M}} |E_{m_{removed}} \oplus E_{\tilde{m}}|$ ▷ エラー集合の対象差が最小となるミュータント
    $w_{m_{nearest}} \leftarrow w_{m_{nearest}} + w_{m_{removed}}$
    $w_{m_{removed}} \leftarrow 0$
  end for
  return $w_{m_1},...,w_{m_n}$

このアルゴリズムでは，残されたミュータントのうち，削減されるミュータントと発生するエラー集合の対象差が最小となるものをテストメソッド内の命令に与える影響が近いものと見なし重みを増すように設計されている．出力される重みの総計は元のミュータント集合のミュータント数と同じであるため，母数の違いによる誤差を減らしてミューテーションスコアを計算できる．ここで削減されたミュータント集合 $\hat{M}$ のうち，新たなテストによって殺されたミュータント集合を $\hat{M}_{killed}$ とすると，ミューテーションスコアは以下の式で表せる．

$$MutationScore = \frac{\sum_{m \in \hat{M}_{killed}} w_m}{\sum_{m \in \hat{M}} w_m} \quad (1)$$

4.4 ミュータント集合最小化の例

表 1 に示すようなミュータントについて最小化することを考える．表 1 のミュータント $m_1,...,m_4$ はすべて殺されるもので，発生するエラー $err_1,...,err_4$ と各ミュータントの関係を t で表現している．また，表 1 の最下段は各ミュータントについての全テストの実行時間になる．

	m_1	m_2	m_3	m_4
err_1	t	t		
err_2	t		t	t
err_3	t		t	
err_4	t		t	
実行時間	1	3	2	2

表 1: ミュータントの例

最初に区別可能なエラー集合を得るため，冗長なエラーを取り除くことを考える．err_3 と err_4 は同じミュータントによって発生するものであるため，区別可能ではないと見なせる．よって err_3 と err_4 のどちらかを削除するが，ここでは err_4 を冗長と見なし削除する．

次にミュータントの選択を行う．アルゴリズム 1 で示した貪欲法では，発生するエラーが一番少ないミュータントから選ぶため，ここでは m_2 と m_4 が候補となる．m_2 と m_4 の実行時間を比較すると，m_4 の方が小さいため，最初に m_4 が選ばれる．次に，発生するエラー数の順番で m_2 が選ばれ，その次に m_3 が選ばれる．この時点でエラーが全て網羅されるため，アルゴリズムは終了となり，選択された m_2, m_3, m_4 を出力する．

最適化の観点で考えると，m_1 を削除するよりも m_2 を削除する方が実行時間は少なくなるため，このアルゴリズムで出力する解は最適解ではないことが分かる．

4.5 ミュータント重み付けの例

前節で使った例を用いてミュータントに重み付けをすることを考える．まず初期状態として各ミュータントの重みとして $w_{m_1}, w_{m_2}, w_{m_3}, w_{m_4} = 1,1,1,1$ を与える．次に削減されるミュータントのエラー集合と対象差が最小となるエラー集合を持つミュータントに対して重みを加えることを考える．削減されるミュータントである m_1 は m_3 のエラー集合と発生するエラーが同じものが多く，実際に E_{m_1} と E_{m_3} の対象差の大きさを求めると，$|E_{m_1} \oplus E_{m_3}| = |err_1| = 1$ となり，他のミュータントのエラー集合との対象差よりも小さいことがわかる．よって m_1 の重みを m_3 の重みに加えることで，$w_{m_1}, w_{m_2}, w_{m_3}, w_{m_4} = 0,1,2,1$ となる．仮に新たなテストによって m_1,m_3 が殺されない場合，重み付けを用いたミューテーションスコアは $\frac{w_{m_2}+w_{m_4}}{w_{m_2}+w_{m_3}+w_{m_4}} = 0.5$ となり，m_1 も m_3 とエラー集合が近いため殺されないと見なされる．重み付けを使わない場合のミューテーションスコアは $\frac{|(m_2,m_4)|}{|(m_2,m_3,m_4)|} = 0.66...$ となり，最小化する前のミューテーションスコア $\frac{|(m_2,m_4)|}{|(m_1,m_2,m_3,m_4)|} = 0.5$ との誤差が生じてしまっているが，重み付けを使うことにより誤差をなくすことができていることがわかる．

5. 評価

5.1 評価方法

4 章で示した提案手法を実現するため，ミューテーション解析ツール PIT [8] に以下の機能を実装した．

- ミュータントに対するテスト実行でテスト失敗した場合に例外とスタックトレースをミュータントに紐づけて記録する
- ミュータントに対するテスト実行でテスト失敗しても引き続き他のテストを実行する
- 各ミュータントの実行時間を記録する
- 記録したミュータント集合を再生実行する

また PIT で用いたミューテーションオペレータを表 A·1 に示す．

これらの機能により記録した各ミュータントの例外，スタックトレース，実行時間を用いて，殺されたミュータントの集合から最小ミュータント集合を求める．

比較として，テストケースベースの最小ミュータント集合についても求め，それぞれの最小ミュータント集合によるミューテーション解析を実行したときの実行時間と，ミューテーションスコアを測定した．ミューテーションスコアの測定は，テストコードのバグ検出力を落としたときに最小ミュータント集合でも元のミュータント集合と同等の値になるかを確認するため，テストコード中のアサーションを一定の割合を削除した上でミューテーションスコアの計測を行った．その際にアサーション中の引数となっているメソッド呼出については，副作用を考慮し，アサーション内から取り出し削除対象としないことで，アサーション以外は元のテストコードと同等の振る舞いがされるようにしている．

5.2 評価対象

今回の提案手法を評価するにあたり，Travis CI のビルド時データを集めた TravisTorrentcite [9] のデータを用いて対象となるコードを以下のように抽出した．

(1) TravisTorrent のデータより，言語が Java で，かつ，ビルドツールが Maven で，かつ，成功するテストコードがあり，かつ，ソースコード行数が 1,000 行以上のリポジトリの master ブランチのコードを選択
(2) 上記コードのうち，PIT 実行時に正常終了するものを残す
(3) 上記コードのうち，記録したミュータントの実行が正常終了するものを残す

このようにして抽出された 53 件のリポジトリを表 A·2 に示す．

これらのコードに対し最小ミュータント集合を求めた．さらに，そのうち jsoup, zt-zip, jInstagram の 3 件のリポジトリについて，アサーション削減時のミューテーションスコアを測定した．

その際の比較対象として，Gopinath ら [10] のテストケースを基準としたミュータント削減手法を用いて最小ミュータント集合を求め，同様にアサーション削減時のミューテーションスコアを測定した．そのときの重み付けについては，エラー集合の代わりにテストケース集合に対しての対象差が最小になるミュータントに重みを足すようにした．

5.3 評価結果

5.3.1 RQ1: どれだけ実行時間が削減されるか

まず提案手法（エラー指向最適化）と既存手法（テスト指向最適化）と最適化なしのそれぞれの場合におけるミューテーション解析の全プロジェクト合計の実行時間を表 2 に示す．また各プロジェクト個別の最適化なしとの実行時間比率の分布について図 1 に示す．図 1 の左がエラー指向最適化，右がテスト指向最適化の分布になる．

表 2: ミューテーション解析の実行時間

	合計実行時間 (msec)	最適化なしとの比率
最適化なし	814,090,026	100%
エラー指向最適化	333,001,232	54.8%
テスト指向最適化	182,450,441	22.4%

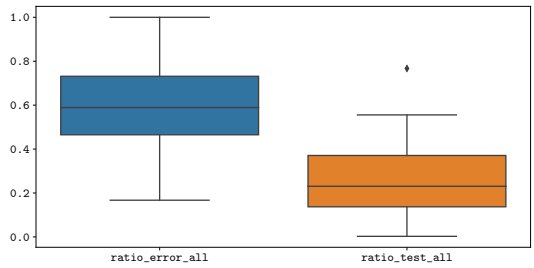

図 1: ミューテーション解析の実行時間の分布

これらの結果を見ると，提案手法のエラー指向最適化により最適化なしに比べて実行時間が 6 割程度に抑えられていることがわかるが，テスト指向最適化ではさらに少ない 1/4 程度の実行時間となっている．つまり提案手法は既存手法であるテスト指向最適化ほど多くの実行時間の削減はできないが，何も最適化をしないよりかは短時間でミューテーションテストが実行できることを意味する．

5.3.2 RQ2: どれだけミューテーションスコアの誤差を少なく測定できたか

5.2 で選んだ 53 件のリポジトリのうち，jsoup, zt-zip, jInstagram の 3 件のリポジトリについて，テストコード中の命令を一定の割合 (80%) で削減した場合のミューテーションスコアを測定し，どれだけ本来のミューテーションスコアとの誤差なく測定できたかを調べた．

表 3 に殺されたミュータント数を，図 2 に最適化なしと各削減方法とのミューテーションスコアの誤差の絶対値を示す．

表 3: アサーション削減時の殺されたミュータント数

		全ミュータント	重み付けなし	重み付けあり
jsoup	最適化なし	9,096	4,419	4,419
	エラー指向最適化	5,915	3,169	4,434
	テスト指向最適化	2,316	1,433	4,537
zt-zip	最適化なし	2,173	1,696	1,696
	エラー指向最適化	1,196	1,110	1,700
	テスト指向最適化	296	258	1,845
jInstagram	最適化なし	1,080	92	92
	エラー指向最適化	795	56	92
	テスト指向最適化	446	48	90

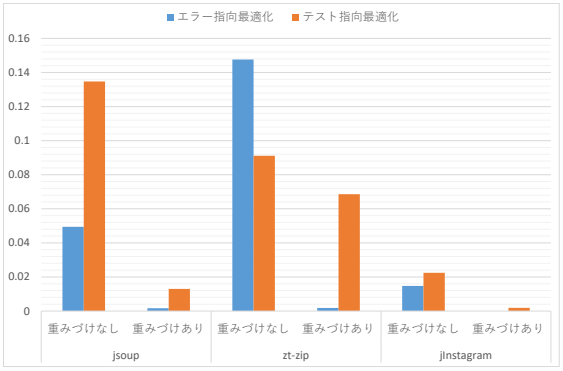

図 2: アサーション削減時のミューテーションスコアの誤差の絶対値

図 2 から分かるように，重み付けを行うことにより，3 件とも誤差の絶対値が大きく減少した．特にエラー指向最適化では重み付けを行うことにより，いずれも誤差の絶対値が 0.002 以下となり，既存手法のテスト指向最適化より優れた結果となった．

さらに jsoup においてテストコード中の命令を削減する割合を変化させたときに重み付けをしたミューテーションスコアがどう変わるか測定した．

表 4 に殺されたミュータント数と図 3 に最適化なしと各削減方法とのミューテーションスコアの誤差の絶対値を示す．

表 4: jsoup におけるテストコード命令削減時の殺されたミュータント数

	全ミュータント	20%	40%	60%	80%
最適化なし	9,096	8,807	7,747	6,274	4,419
エラー指向最適化	5,915	8,819	7,745	6,262	4,434
テスト指向最適化	2,316	8,865	7,808	6,327	4,537

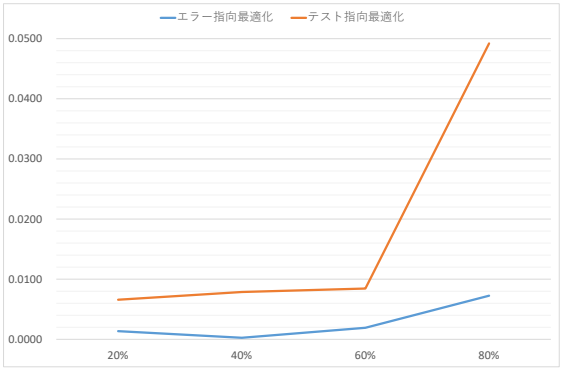

図 3: jsoup におけるテストコード命令削減時のミューテーションスコアの誤差の絶対値

図 3 からいずれのテストコード命令削減率においてもエラー指向最適化の方がテスト指向最適化に比べて誤差が少ないことがわかる．特にテストコード命令削減率が 80%のときにその傾向が顕著に表れている．

5.3.3 RQ3: どのミューテーションオペレータが多く削減されたか

提案手法によって削減されたミューテーションオペレータの傾向を知ることは未知のミュータント集合に対して削減を行うときの指針となり得るため，重要である．提案手法によってどのミューテーションオペレータが多く削減されているかについて調査した結果を図 4 に示す．図 4 中の青い棒グラフが元のミュータント集合，赤い棒グラフが削減されたミュータント集合のミュータント数を表す．さらに各ミューテーションオペレータにおける削減率（削減対象のミュータント数 / 総ミュータント数）を図 5 に示す．

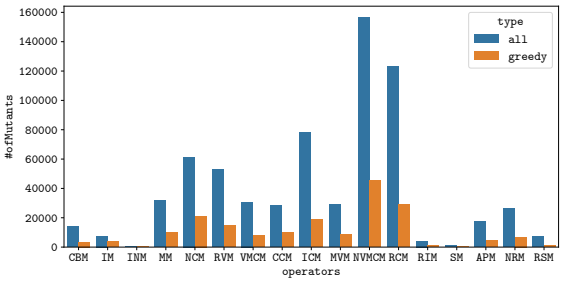

図 4: ミューテーションオペレータごとのミュータント数

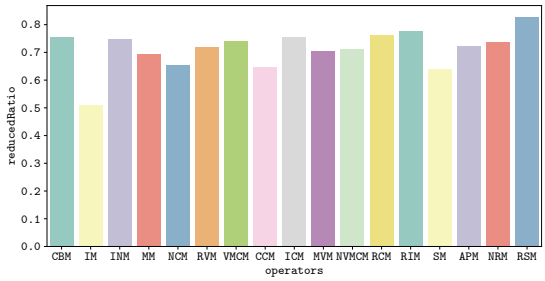

図 5: ミューテーションオペレータごとのミュータント削減率

ミューテーションオペレータごとのミュータント数は，Non Void Method Calls Mutator が削減前後で変わらず一番多いことが分かる．一方，ミューテーションオペレータごとのミュータント削減率は Increments Mutator が一番少なく，Remove Switch Mutator が一番多いが，全体的に削減率のばらつきがそれほど多くなく，特徴的と言えるような削減率のものはなかった．

6. 議論

6.1 実行時間の最適化について

提案手法の評価では実行時間を約 6 割に削減できることが分かった．つまり，未知のミュータント集合に対する選

択をする場合に，信頼性に重きを置く場合はその程度の量の選択をすることが過剰な削減を防ぐ上でも大切だと考える．一方でミュータント選択のアルゴリズムを貪欲法により設計したため，実行時間を最小とする最適解とはなっていない．エラー集合を区別可能とする制約条件を記述できるようなソルバを用いることで，指数オーダーとなる計算量の問題を軽減しながら最適解を求めることを検討する必要がある．

6.2 ミューテーションスコアの誤差の削減について

提案手法では既存手法と比べミューテーションスコアの誤差を減らせることが分かった．また誤差を減らすための重み付けについては，エラー指向最適化だけではなく，既存手法であるテスト指向最適化においても効果のある手法だということが今回の実験で明らかになった．

実行時間の評価で使用した 53 件のリポジトリは系統的に選択されているが，その後の誤差の評価で用いた 3 件のリポジトリについては，テストコード命令削減用に作成したツールが適用でき，かつ，命令削減後ビルドを通すまでの修正が多くないものから 3 件を選択したため，偏りが含まれている可能性が考えられる．また，命令削減率を変化させての評価は jsoup のみに留まるため，今後より多くのリポジトリに対して誤差の評価を行う必要がある．

6.3 ミューテーションオペレータごとの削減量について

ミューテーションオペレータごとのミュータント数を実験で確認したが，ミュータント削減前と削減後のもので分布に大きな差異はなかった．また，ミューテーションオペレータごとの削減率についても調べたが，特徴的と言えるような削減率のものはなかった．つまり，今回の削減モデルでは特定のミューテーションオペレータのミュータントを大きく削減するものではなく，もし未知のミュータント集合に対して削減することを考える場合は，ミューテーションオペレータの傾向から削減することは難しいことを示していると考える．今回の削減モデルのミュータントの他の傾向を調べることで未知のミュータント集合の削減に寄与するものがある可能性もあり，それについてはさらなる調査が必要と考える．

7. 関連研究

これまでミュータント削減手法は多く提案されてきている．

いくつかの論文ではミューテーションオペレータを選択することによる削減とランダムサンプリングによる削減を比較している．Budd [11] と Acree [12] は 10%のミュータントをサンプリングすることで元のミューテーションスコアを 99%の精度で近似できることを示した．Wong ら [13] は各ミューテーションオペレータのミュータントを x% ずつ残すのと，2 つのミューテーションオペレータのみを使う場合を比較し，両方において同等のミューテーションスコアと同等の精度を実現した．Zhang ら [14] はオペレータベースの削減方法とランダムサンプリングを比較し，ランダムサンプリングの方が優れていることを示した．

十分なミューテーションオペレータの集合を調べる研究も多く存在する．Offutt ら [15] は 6 種類のミューテーションオペレータで元の 99.5%の精度でミューテーションスコアを計測できることを示した．Barbosa ら [16] はミューテーションオペレータ選択の指針を示し，65.02%に削減されたミュータントによって元の 99.6%の精度のミューテーションスコアを実現した．

これらのミュータント削減手法については，すべて未知のミュータントに対するものであり，ミュータント削減量の限界を示すものではない．

ミュータント削減量の限界については Gopinath ら [10] のテストケースベースの削減モデルが存在する．Gopinath らは削減量の理論的上限と実験的に求めた上限を示した．削減量の理論的上限は $1/(e-1) \approx 58.2\%$ となるが，一方で実験的上限は平均で 13.078%となった．我々は単純にミュータントを削減するのではなく，削減するミュータントの実行時間を考慮し，実行時間を最小化するミュータントを選択することを考えた．

削減後のミュータント集合に対するミューテーションスコアについては，我々の調査の範囲では既存研究において言及されてきておらず，本研究はその観点について指摘し解決手法を提案した初めての研究と考える．

8. おわりに

本論文ではテストコード中の命令の変更があるような場合でも元のミュータント集合と同等なバグ検出力の測定ができるようなミュータント削減モデルの必要性についての課題を述べた．そして，新たなミュータントの削減モデルとして，テスト実行時に発生するエラーが区別可能であることを保つミュータントを選択し，それ以外のミュータントを冗長と見なす方法を提案した．さらにミュータント削減によるミューテーションスコア計算時の誤差を減らすため，削減されるミュータントの影響を重みとして選択されたミュータントに付け加え，重みによってミューテーションスコアを計算する方法を提案した．OSS の 53 プロジェクトに対する速度評価では約 40%に実行時間を削減でき 3 プロジェクトに対してバグ検出力の不変性について評価を行ったところ，既存のテストケースベースのミュータント削減モデルより誤差の少ない結果となった．

今後はさらなるプロジェクトでの調査や，今回の計測情報などを用いた未知のミュータント集合の削減について取り組む予定である．

参考文献

[1] Ammann, P., Delamaro, M. E. and Offutt, J.: Establishing Theoretical Minimal Sets of Mutants, *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 21–30 (online), DOI: 10.1109/ICST.2014.13 (2014).

[2] Deursen, A., Moonen, L. M., Bergh, A. and Kok, G.: Refactoring Test Code, Technical report, Amsterdam, The Netherlands, The Netherlands (2001).

[3] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. and Binkley, D.: An empirical analysis of the distribution of unit test smells and their impact on software maintenance, *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 56–65 (online), DOI: 10.1109/ICSM.2012.6405253 (2012).

[4] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. and Binkley, D.: Are test smells really harmful? An empirical study, *Empirical Software Engineering*, Vol. 20, No. 4, pp. 1052–1094 (online), DOI: 10.1007/s10664-014-9313-0 (2015).

[5] 徳本　晋，石井康嗣：ミューテーション解析における非利用アサーションの実証評価，ウィンターワークショップ 2018・イン・宮島論文集，Vol. 2018, pp. 20–21 (2018).

[6] 徳本　晋：効果的なミューテーションテストのためのエラー指向のミュータント削減，電子情報通信学会技術研究報告，Vol. 118, No. 69, 電子情報通信学会，pp. 49–53 (2018).

[7] Offutt, A. J. and Untch, R. H.: Mutation Testing for the New Century, Kluwer Academic Publishers, Norwell, MA, USA, chapter Mutation 2000: Uniting the Orthogonal, pp. 34–44 (online), available from ⟨http://dl.acm.org/citation.cfm?id=571305.571314⟩ (2001).

[8] Coles, H.: PIT, `http://pitest.org`.

[9] Beller, M., Gousios, G. and Zaidman, A.: TravisTorrent: Synthesizing Travis CI and GitHub for Full-stack Research on Continuous Integration, *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, Piscataway, NJ, USA, IEEE Press, pp. 447–450 (online), DOI: 10.1109/MSR.2017.24 (2017).

[10] Gopinath, R., Alipour, M. A., Ahmed, I., Jensen, C. and Groce, A.: On The Limits of Mutation Reduction Strategies, *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 511–522 (2016).

[11] Budd, T. A.: Mutation Analysis of Program Test Data, PhD Thesis, New Haven, CT, USA (1980).

[12] Acree, Jr., A. T.: On Mutation, PhD Thesis, Atlanta, GA, USA (1980).

[13] Wong, W. E. and Mathur, A. P.: Reducing the Cost of Mutation Testing: An Empirical Study, *J. Syst. Softw.*, Vol. 31, No. 3, pp. 185–196 (online), DOI: 10.1016/0164-1212(94)00098-0 (1995).

[14] Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T. and Mei, H.: Is Operator-based Mutant Selection Superior to Random Mutant Selection?, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, New York, NY, USA, ACM, pp. 435–444 (online), DOI: 10.1145/1806799.1806863 (2010).

[15] Offutt, A. J., Rothermel, G. and Zapf, C.: An Experimental Evaluation of Selective Mutation, *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 100–107 (online), available from ⟨http://dl.acm.org/citation.cfm?id=257572.257597⟩ (1993).

[16] Barbosa, E. F., Maldonado, J. C. and Vincenzi, A. M. R.: Toward the determination of sufficient mutant operators for C ††, *Software Testing, Verification and Reliability*, Vol. 11, No. 2, pp. 113–136（オンライン），DOI: 10.1002/stvr.226 (2001).

付　　録

表 A·1: 使用したミューテーションオペレータ

タイプ	ミューテーションオペレータ名	略称
Default	Conditionals Boundary Mutator	CBM
	Increments Mutator	IM
	Invert Negatives Mutator	INM
	Math Mutator	MM
	Negate Conditionals Mutator	NCM
	Return Values Mutator	RVM
	Void Method Calls Mutator	VMCM
Non-Default	Constructor Calls Mutator	CCM
	Inline Constant Mutator	ICM
	Member Variable Mutator	MVM
	Non Void Method Calls Mutator	NVMCM
	Remove Conditionals Mutator	RCM
	Remove Increments Mutator	RIM
	Switch Mutator	SM
	Argument Propagation Mutator	APM
	Naked Receiver Mutator	NRM
	Remove Switch Mutator	RSM

表 A·2: 対象のリポジトリとそのコード規模

リポジトリ名	ソースコード行数	テストコード行数
msgpack-java	13,598	28,243
vertx-jersey	1,936	1,854
samoa	16,841	249
owner	2,866	5,520
redline-smalltalk	5,648	451
geometry-api-java	57,497	18,619
vectorz	39,793	8,439
webcam-capture	13,659	745
scribe-java	2,794	2,549
gson-fire	1,535	1,312
jsr354-api	2,319	3,487
p2-maven-plugin	1,690	153
jackson-annotations	1,471	331
auto	8,710	11,808
rxjava-jdbc	3,735	3,081
okio	3,998	4,499
jInstagram	4,015	6,748
javapoet	2,994	4,450
gwtbootstrap3	13,560	406
jphp	43,717	4,441
javaparser	13,588	3,379
JsonPath	3,765	2,714
minimal-json	1,715	4,589
jackson-core	21,451	11,131
rest-driver	3,194	5,108
retrofit	5,046	8,361
zt-zip	4,064	2,220
maven-git-commit-id-plugin	2,883	2,015
linq4j	14,307	3,979
RoaringBitmap	9,267	7,092
Ektorp	11,079	5,876
jsoup	10,696	5,188
moshi	4,136	5,515
http-request	1,391	2,721
slf4j	8,184	4,288
twilio-java	12,835	4,230
graphhopper	22,544	7,549
cassandra-reaper	5,663	1,711
LittleProxy	4,094	4,550
hbc	3,588	1,829
jsprit	20,486	16,607
wire	8,462	27,743
HikariCP	4,135	5,155
java-object-diff	5,817	1,913
docker-maven-plugin	6,085	1,924
stream-lib	4,689	3,619
jsondoc	3,841	3,543
metadata-extractor	18,731	2,344
jOOQ	130,053	1,464
pebble	6,324	4,725
alf.io	7,226	499
zxing	35,164	7,582
Algorithms	1,016	1,356