Vol.2019-SE-201 No.7
2019/3/7

プログラミング演習における 自動正誤判定の精度向上を目的とするミューテーション手法

中井 舞人[1,a)]　大久保 弘崇[1,b)]　粕谷 英人[1,c)]　山本 晋一郎[1,d)]

概要：e ラーニングシステムを用いたプログラミング演習において，正誤判定システムにソフトウェアテストを利用することができる．ここで，出題者が用意したテストスイートが不完全な場合，受講者の解答プログラムに対して偽陽性や偽陰性を示すことがある．本研究は，偽陽性を持つプログラムの採点ミスを減らすために，ミューテーションテストを利用してあらかじめテストスイートを通過する誤解答を検出し，その結果をテストスイートに反映させることで，自動正誤判定の精度を改善することを目的とする．また，より多くのテストスイートの弱点を指摘するミュータントを生成させるため，使用するツールのミュータント生成アルゴリズムを改良した．ミューテーション解析の結果，本研究の手法によるテストスイートの欠陥検出が可能であり，テストスイートの改善に役立つことが確認された．

1. はじめに

ウォータフォールモデルに代表される多くのソフトウェア開発は，要件定義，システム設計，プログラミング，テスト，検収，保守及び運用のプロセスを経る．テストとはコンピュータのソフトウェアプログラムを実行し，それが設計者の意図したとおりに動くかを観測・評価・検証する作業のことであり，一般的にソフトウェアテストと呼ばれる．ソフトウェアテストを行う目的は，欠陥を検出すること，システムの仕様を満たすか確認することなどが挙げられる．ソフトウェアの品質を保証する上で, ソフトウェアテストは重要な役割を担っている.

ソフトウェアテストの手法は，e ラーニングシステムを用いたプログラミング演習において，ある問題に対して受講者が提出したプログラムが，出題側の意図した動作を行うかを検証・採点する目的で使用されることがある [1]．ここで，出題側の用意したテストスイートが完全でない場合，受講者の解答プログラムが出題者の想定した動作をせずともテストを通過することや，解答プログラムが出題者の想定した動作をしているにも関わらず，テストを通過できないことがある．このような採点ミスを減らすため，プログラミング演習に用いるテストスイートの精度の向上が求められている．

本研究では，テストの分析手法であるミューテーションテスト [2] に着目する．ミューテーションとは，ある動作を行うプログラムに部分的な変更を加え，異なるプログラムへ変異させることである．ミューテーションテストによりテストスイートの不備を検出し，それを基に演習問題の正誤判定を改善できるかを検証・評価する．対象とする言語については，本学の「プログラミング入門」の講義で使用するプログラミング言語 Haskell を選択する．ミューテーションテストツールは MuCheck[3] を使用する．

2. ミューテーションテスト

ソフトウェアテストがプログラムの品質を測る手法であるように，ミューテーションテストはソフトウェアテストの品質を測る手法のひとつである．ミューテーションテストは以下の流れで行われる．

(1) 成果物プログラムに対して意図的に欠陥を埋め込み，欠陥を含むプログラム (これをミュータントと呼ぶ) を生成する．
(2) ミュータントに対してテストスイートを実行する．ミュータントは欠陥を含むため，このテストスイートの実行は失敗することが期待される．失敗は，テストスイートがミュータントを検出できたことを意味する．
(3) 生成されたミュータントに対して検出されたミュータントの割合を計算し，ミューテーションスコアとしてテストスイートの品質を数値化する．

ソフトウェア開発では，ミューテーションテストから得

[1] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
a) nakai@yamamoto.ist.aichi-pu.ac.jp
b) ohkubo@ist.aichi-pu.ac.jp
c) kasuya@ist.aichi-pu.ac.jp
d) yamamoto@ist.aichi-pu.ac.jp

られた結果を元にテストすべきプログラムの性質をテストケースに加えることでスコアを改善できるため，テストスイートの品質の向上に応用することができる．

ミューテーションテストに関する用語を説明する．

ミュータント ミュータントは，分析に用いられる元プログラムをミューテーションのアルゴリズムに従って変異させたプログラムである．通常，元プログラムから複数のミュータントが生成され，テストスイートによって検知されたものを Kill されたミュータント，検知されなかったものを Alive したミュータントと呼ぶ．

ミューテーション操作 ミューテーション操作とは，ミューテーションテストの対象プログラムを元に，ミュータントを作成するためのプログラム書き換え方法のことである．例えば，プログラム内に加算演算子 (+) が使われている箇所を減算 (-) 除算 (/) 乗算 (*) のいずれかに置き換えることや，プログラム内の特定の文を削除すること等は，基本ミューテーション操作 [4] として知られている．

等価ミュータント 生成されたミュータントのうち，一部のミュータントは元のプログラムと動作が変わらないことがある．このようなミュータントを等価ミュータントと呼ぶ．等価ミュータントはミューテーションテストの未解決な問題の一つであり [5]，ミューテーションスコアのノイズとなる．

ミューテーションスコア ミューテーションスコア score は次のように定義される．

$$score = \frac{\text{検出されたミュータントの数}}{\text{生成されたミュータントの数}}$$

スコアはテストスイートの欠陥検出能力を表し，一般的にスコアの高いテストスイートほど質の高いテストスイートとされる．

3. MuCheck

MuCheck は，プログラミング言語 Haskell を対象としたミューテーションテストツールであり，Haskell で書かれたテストスイートに対してミューテーションテストを行うことができる．関数型言語である Haskell を対象としたことにより，一般的なオブジェクト指向型言語を対象としたミューテーションテストツールと異なる以下の特徴を持つ．

- QuickCheck[6] や SmallCheck[7]，HUnit[8] 等の Haskell のテストフレームワークをテストの対象にすることができる．
- Haskell に特有のミューテーション操作を定義している．例えば，パターンマッチングを入れ替えるミューテーションや，特定のリスト操作を別のリスト操作へ置き換えるミューテーションがある．また，Haskell において関数は第一級オブジェクトであるため，呼び出される関数を同じ型をもつ別の関数に置き換えるミューテーションがある．

```
1  -- 変更前のミューテーション条件関数
2  selectIfElseBoolNegOps :: Module_ -> [MuOp]
3  selectIfElseBoolNegOps m
4                  = selectValOps isIf convert m
5    where isIf :: Exp_ -> Bool
6          isIf If{} = True
7          isIf _    = False
8          convert (If l e1 e2 e3)
9                    = [If l e1 e3 e2]
10         convert _ = []
11
12 -- 変更後のミューテーション条件関数
13 selectIfElseBoolNegOps :: Module_ -> [MuOp]
14 selectIfElseBoolNegOps m
15                 = selectValOps isIf convert m
16   where isIf :: Exp_ -> Bool
17         isIf If{} = True
18         isIf _    = False
19         convert (If l e1 e2 e3)
20                   = [If l e1 e3 e3]
21         convert _ = []
```

図 1 ミューテーションの実装例

3.1 ミューテーションの実装

MuCheck はミューテーション操作の定義を実装するためのドメイン固有言語 (DSL) を提供する．DSL は以下の 3 つの多相型演算子からなる．

- 構文 a を構文 b に置き換えさせる演算子 `a ==> b`.
- 構文 a を構文 b1 と構文 b2 の両方に置き換えたリストにする演算子 `a ==>* [b1,b2]`.
- 構文 a1 または構文 a2 のいずれかを構文 b1 と構文 b2 の両方に置き換えたリストにする演算子 `[a1,a2] *==>* [b1,b2]`.

テスト対象のプログラムは，`selectValOps` 関数により DSL の演算子を介してミュータントに置き換えられる．その際，`selectValOps` 関数はプログラム内のある構文を別の構文へ置き換える関数を引数にとる．例として，数値や文字列などのリテラルを同型の別の値へ置き換える `selectLiteralOps` 関数や，if 式の then 節と else 節を入れ替える `selectIfElseBoolNegOps` 関数が，ミューテーションを行う構文を決定する条件関数である．

この条件関数を新たに加えるまたは変更することで，新しいミューテーション操作の定義や既存のミューテーション操作の定義を変更することができる．例えば，前述した if 式の then 節と else 節を入れ替える `selectIfElseBoolNegOps` 関数を，then 節を else 節へ置き換えるミューテーション関数へ変更したい場合は，図 1 のように 8 行目の `convert` 補助関数の結果を `[If l e1 e3 e2]` から `[If l e1 e3 e3]` へ変更することで実現することができる．

```
1  module Examples.AssertCheckTest where
2
3  import Test.MuCheck.TestAdapter.AssertCheck
4
5
6  qsort :: [Int] -> [Int]
7  qsort [] = []
8  qsort (x:xs) = qsort l ++ [x] ++ qsort r
9      where l = filter (< x) xs
10           r = filter (>= x) xs
11
12 uncoveredDummy :: Int -> Int
13 uncoveredDummy a = 0 + a
14
15 {-# ANN sortEmpty "Test" #-}
16 sortEmpty = assertCheck $ qsort [] == []
17
18 {-# ANN sortSorted "Test" #-}
19 sortSorted = assertCheck $ qsort [1,2,3,4] ==
       [1,2,3,4]
20
21 {-# ANN sortRev "Test" #-}
22 sortRev = assertCheck $ qsort [4,3,2,1] ==
       [1,2,3,4]
23
24 {-# ANN sortSame "Test" #-}
25 sortSame = assertCheck $ qsort [1,1,1,1] ==
       [1,1,1,1]
26
27 {-# ANN sortNeg "Test" #-}
28 sortNeg = assertCheck $ qsort [-1,-2,3] ==
       [-2,-1,3]
```

図 2 テスト対象プログラムとテストスイート

3.2 ミューテーションの選択

MuCheck が行うミューテーションは設定ファイルにより制御可能であり，以下の設定を変更することができる．

- 生成するミュータントの最大数
- 高階関数に対するミューテーションの非実行
- 適用するミューテーション操作の選択
- 置換する演算子または関数同士の組み合わせの変更

3.3 MuCheck の分析例

テスト対象のプログラム図 2 はリストのソート関数とテストに関係ないダミー関数で構成されており，5 つのテストケースからなるテストスイートの分析を行う．

テストスイートの分析の結果は図 3 のように表示される．この例では，テスト対象のプログラムから生成されたミュータントの総数 (Total mutants) が 19 種あり，そのうちテストスイートによるテストに用いられないダミー関数にミューテーションが適用された結果生成されたミュータント 6 種を除いた数 (Covered，Sampled) は 13 種である．加えて，テストの結果テストスイートが誤りを検知できたミュータント (Killed) が 12 種あり，誤りを検知できなかった (テストスイートの不備を示す可能性がある) ミュータント (Alive) が 1 種あったことが確認できる．またミューテーションスコアは 63%である．

```
1  Total mutants: 19 (basis for %)
2          Covered: 13
3          Sampled: 13
4          Errors: 0  (0%)
5          Alive: 1/19
6          Killed: 12/19 (63%)
```

図 3 テストサマリ

4. 演習問題のテストスイート欠陥検出実験

ミューテーションテストを用いた演習問題テストスイートの欠陥検出の有用性を検証するため，本学の 2018 年のプログラミング入門の期末テスト全 15 問のうち，問題の性質上分析のノイズとなる等価ミュータントを多く生成する 1 問を除いた 14 問を対象とし，MuCheck を使用して問題に使用されるテストスイートの不備を検出する実験を行った．また，テストスイートはいくつかの固定テストとランダムテストから構成されている．

結果，全問合わせて 93 のミュータントが生成され，その中で 17 のミュータントが Alive であった．この Alive ミュータントからテストスイートの改善に有益なミュータントを探したところ，その全てが等価ミュータントであったため，自動正誤判定の精度向上に役立てることができなかった．

ここで，テストスイートの不備が検出されなかった原因として以下の 2 点が考えられる．

- 演習問題に使用されたテストスイートが充分に強力である．
- MuCheck が実装しているミューテーション定義が欠陥を検出するには不十分である．

4.1 テストスイートが完璧な場合

実験対象のテストスイートからいくつかのテストケースを除いて意図的に欠陥を持つテストスイートを作り，その欠陥を分析により検出することができるかを調べることで，提案手法のアプローチの有用性を示す．

テストスイートから固定テストを抜いた場合のスコアと，ランダムテストを抜いた場合のスコアを 6.1 節で示す．

4.2 MuCheck に問題がある場合

MuCheck のミューテーションを追加実装することによりテストスイートの欠陥を検出できる可能性がある．既に，MuCheck は内蔵のミューテーションの構文置き換え関数として，3.1 節で紹介したミューテーション条件関数に加えてパターンマッチの順序の入れ替えを行う `selectFnMatches` 関数，ガードの条件部への否定の挿入を行う `selectGuardedBoolNegOps` 関数，関数と演

```
1 -- original f1
2 f1 :: Int -> Int -> Int
3 f1 x y = (succ x) `div` (succ y)
4
5 -- mutant f1
6 f1 :: Int -> Int -> Int
7 f1 x y = (succ y) `div` (succ x)
8
9 test = f1 3 3 == 1 -- True
```

図 4 引数置換

算子の置換を行う selectFunctionOps 関数の構文に基づく 5 つのミューテーション操作を実装している．また具体的な演算子と関数の置換では，succ と pred の置換や四則演算 (+)，(-)，(*)，(/) の置換などのミューテーション操作を定義している．

しかし，演算子の置換においては論理和 (||) と論理積 (&&) の置換が，構文の置換では式の定数置換のミューテーションなど，基本ミューテーション操作のうちいくつかのミューテーションが定義されていない．

そこで，未実装の基本ミューテーションに加え，畳込み関数の置換といった Haskell に特有のミューテーション操作を新たに定義することで，生成されるミュータントの量と質の向上を 5 章でする．ミューテーションを追加実装した MuCheck を便宜的に MuCheck+と呼び，その評価は 6.2 節で示す．

5. ミューテーションの追加実装

MuCheck の改善を目的として新たに実装したミューテーションを，ミューテーション毎にその意図と共に説明する．またミューテーション例の下部に，そのミューテーションで作られたミュータントを検出することができないテストの関数の例を載せる．また，MuCheck にあらかじめ定義している関数に対する修正の有無，ミューテーションの種類 (基本か Haskell 特有のミューテーションか)，MuCheck+で追加したミューテーション関数の一覧を付録 A.1 に添付する．

引数置換のミューテーション

2 引数関数の引数を入れ替えるミューテーションである．図 4 の 3 行目の div の呼び出しの引数を入れ替え，7 行目のように入れ替える．

また (a -> b -> a) など，1 引数目と 2 引数目の型が異なる関数に対してこのミューテーションを行うことで，型の不一致によるコンパイルエラーが発生する可能性がある．しかし，このような場合 MuCheck はエラーミュータントを破棄するだけなので，分析に影響はない．

式置き換えのミューテーション

関数の型シグネチャに応じて式を定数に置換する．図 5

```
1  -- original f2
2  f2 :: Int -> String -> String
3  f2 1 s = "1␣" ++ s
4  f2 2 s = "2␣" ++ s
5  f2 _ s = "other␣num␣" ++ s
6
7  -- mutant f2
8  f2 :: Int -> String -> String
9  f2 1 s = "1␣" ++ s
10 f2 2 s = "dummy␣string"
11 f2 _ s = "other␣num␣" ++ s
12
13 test = f2 1 "hoge" == "1␣hoge" -- True
```

図 5 式置き換え

```
1  -- original f3
2  f3 :: Int -> Int
3  f3 x
4    | x < 1     = x
5    | otherwise = x + 10
6
7  -- mutant f3
8  f3 :: Int -> Int
9  f3 x
10   | x < -1    = x
11   | otherwise = x + 10
12
13 test = f3 1 == 11 -- True
```

図 6 符号反転

```
1  -- original f4
2  f4 :: [Int] -> Int
3  f4 x = foldl (-) 0 x
4
5  -- mutant f4
6  f4 :: [Int] -> Int
7  f4 x = foldr (-) 0 x
8
9
10 test = f4 [0,1,0] == -1 -- True
```

図 7 畳み込み関数置換

の 4 行目の右辺を，10 行目のようにダミー文字列へ置き換える．

符号反転のミューテーション

Int, Float, Double 型の定数にマイナスを付ける．図 6 の 4 行目に表れる定数 1 を 10 行目のように-1 を掛けて置き換える．

このミューテーションには，テストプログラム内に単項演算子 (-) を引き算の演算子と区別する NegativeLiterals 拡張が必要である．

畳み込み関数置換のミューテーション

foldl と foldr の置換を行う．図 7 の 3 行目の foldl を 7 行目の foldr へ置き換える．

```
1 -- original f5
2 f5 :: Bool -> Bool -> Bool
3 f5 x y = not x && y
4
5 -- mutant f5
6 f5 :: Bool -> Bool -> Bool
7 f5 x y = not x || y
8
9 test = f5 False True == True -- True
```

図 8　論理演算子置換

```
1 elem t (x : xs) = t < x || elem t xs
2 elem _ [] = False
```

図 9　elem 関数の固定テスト通過ミュータント 1

```
1 elem t (x : xs) = t <= x || elem t xs
2 elem _ [] = False
```

図 10　elem 関数の固定テスト通過ミュータント 2

```
1 zip [] _bs = []
2 zip (a : as) (b : bs) = (a, b) : zip as bs
```

図 11　zip 関数の固定テスト通過ミュータント

論理演算子置換のミューテーション

論理演算子（&&）と（||）の置換を行う．図 8 の 3 行目に表れる（&&）を 7 行目の（||）へ置き換える．

6. 評価

4 章の実験から得られた結果を基に，テストスイートと MuCheck の条件を変えた状態で再度実験を行うことで，得られるテストスイートの欠陥にどのような変化があるかを評価する．

6.1 実験 1

MuCheck が不完全なテストスイートの欠陥を検出することができるか調べるため，意図的にテストケースを除いたテストスイートを用いて 4 章の実験と同様の実験を行う．

6.1.1 固定テストのみ

4 章の実験に使用されたテストスイートからランダムテストを除き，固定テストのみを分析対象としてミューテーションテストを行った．固定テストのみを対象とすることで新たに検出されたテストスイートの欠陥は 3 種であった．

図 9 と図 10 は関数 elem の再実装問題の，図 11 は GHC のプレリュード関数 zip の再実装問題の固定テストを通過したミュータントである．具体的な問題本文とテストスイートの内容は本稿の付録 A.2 に添付する．

6.1.2 ランダムテストのみ

4 章の実験に使用されたテストスイートから固定テストを除き，ランダムテストのみを分析対象としてミューテーションテストを行った．ランダムテストの性質上，分析

```
1  module Examples.QuickCheckTest where
2
3  import Test.QuickCheck
4
5  pencil :: Int -> (Int,Int) -> (Int,Int) ->
       Int
6  pencil n (a,x) (b,y) = min (x * sub a) (y *
       sub b) where
7    sub k = (n+k-0) `div` k
8
9  {-# ANN test1 "Test" #-}
10 test1 n (a,x) (b,y) = pensil n (a,x) (b,y) ==
        min (x * sub a) (y * sub b) where
11   sub k = (n+k-1) `div` k
```

図 12　買い物選択問題のランダムテスト通過ミュータント

```
1 Total mutants: 169 (basis for %)
2         Covered: 169
3         Sampled: 169
4         Errors: 25  (15%)
5         Alive: 25/169
6         Killed: 119/169 (70%)
```

図 13　elem 関数の固定テスト通過ミュータント 1

を複数回行えばテストスイートをすり抜けるようになるミュータントが生成されるケースがあるため，テストスイートをすり抜ける確率が高い，およそ分析の 4 回に 1 回はテストスイートをすり抜けるミュータントをピックアップして 1 種載せる．

図 12 は n 本の鉛筆を 2 種の商品から適切な数を最低金額で購入する問題である．補助関数の境界条件に 1 を足すことにより，場合によってランダムテストをすり抜けることができるようになっている．具体的な問題本文とテストスイートの内容は本稿の付録 A.2 に添付する．

6.1.3 実験 1 の結果

実験より，元のテストスイートに意図的な欠陥を加えることにより，MuCheck がその欠陥を検出できていることが確認できた．これにより，本手法によるテストスイートの欠陥検出が可能であることを示した．

6.2 実験 2

欠陥検出実験と同じ問題を対象とし，ミューテーションを拡張した MuCheck+により問題に使われるテストスイート共通の欠陥が検出できるかを評価する．

図 13 は MuCheck+による 14 問の分析結果の合計である．ここで生成された 25 の Alive ミュータントのうち，等価ミュータントでないミュータントが 1 種見つかった．

6.2.1 MuCheck+により検出されたテストスイートの欠陥

関数 elem の定義問題にて，模範解答プログラムと異なる動作でありながらテストスイートを通過する図 14 のミュータントが検出された．オリジナルの elem は，探索したい

```
1 elem t (x : xs) = elem t xs || t == x
2 elem _ [] = False
```

図 14 関数のテスト通過ミュータント

```
1 test = elem 1 [1..] == True
```

図 15 test elem

要素を見つけると以降のリストの走査をせずに結果を返すのに対して，間違った elem は，リストの中の探索したい要素を見つけた後もリストの末尾まで要素を探索する．これにより，無限リストを引数に与えた場合は，探索したい要素が含まれていても計算が止まらず，模範解答プログラムと異なる動作をする．

このような性質をテストしたい場合は，テストスイートに図 15 のテストケースを加えることで元のプログラムと異なる動作をするため，検出可能になる．

6.2.2 実験 2 の結果

提案ミューテーションにより，生成されたミュータントの数が MuCheck より 76 種増え，そのうち 1 つがテストスイートの欠陥を検出することができた．ここから，検出された欠陥を検出することができるテストケースを加えることで，テストスイートの正誤判定の精度を向上させることができたといえる．

6.3 考察

MuCheck へのミューテーション追加実装により，無限リストを想定した限定的な条件下のみだがテストスイートの欠陥を検出することができた．しかし，ランダムテストをテストスイートから除いた状況下での MuCheck+の実験で検出されたテストスイートの欠陥は，実験 2 で検出したミュータント以外は MuCheck が検出したものと同じであった．すなわち，実験に用いられたテストスイートの質が下がったにも関わらず，MuCheck+が検出したテストスイートの欠陥の数は生成したミュータントの数に比べて低いことを意味する．ここから，提案ミューテーションは生成するミュータントを増加させたが，そのほとんどがエラーかテストスイートに検出されたミュータントであることがわかる．

一般的なミューテーションテストでは，大規模なプログラムを対象として分析した場合は生成されるミュータントが莫大に増え，計算時間が長くなることが多いため，少ないミュータントで多くのテストスイートの欠陥を検出できるミューテーションが良いとされる．しかしながら，本研究が対象とするプログラミング演習の模範解答のような短いプログラムでは分析時間が短いため，分析時間の長大化を考慮する必要は少ない．よって，ミューテーションの効率を無視して様々なミューテーションを定義することはテストスイートの欠陥検出に必要といえる．

7. まとめ

本研究では，提案ミューテーション手法により拡張した Haskell のミューテーションテストツール MuCheck を用いて，本学の講義「プログラミング入門」の期末テスト問題を対象に，自動正誤判定の精度向上に有益なミュータントが生成されたことを確認した．また，テストスイートからランダムテストや固定テストを除いた状態で同様の実験を行ったところ，検出できたテストスイートの不備がより多くなることも確認できた．これにより，提案ミューテーション手法の有用性と複数のテスト手法を組み合わせてテストすることの重要性を示した．

8. おわりに

6.3 節で述べたように，本研究が対象とするプログラミング演習の模範解答のような短いプログラムでは分析時間が短いため，ミューテーションの効率を考慮する必要性は低い．しかし，テストスイートの欠陥の検出に無意味なミューテーションを加えることで，多量の等価ミュータントを生成させてしまう可能性がある．等価ミュータントの生成は分析のノイズになるため，そのようなミューテーションを実装することは好ましくない．そのため，分析にとって意味のある効率的なミューテーションについて考察する必要がある．

また今後の課題として，ミューテーションテストの結果を基に，テストスイートに追加すべきテスト項目の候補を推薦する方法などが挙げられる．具体的な手法として，ランダムテストなどの自動テストから欠陥を Kill できるテストケースを探し，利用者へ知らせる手法などが考えられる．

謝辞 本研究は JSPS 科研費 15K00488 の助成を受けたものである．

参考文献

[1] 大久保弘崇, 山本晋一郎, 『ランダムテストを応用したプログラミング演習における答案の自動正誤判定機構』, 情報処理学会第 77 回全国大会講演論文集, volume 4, pages 529–530 2015 年 3 月.
[2] Y Jia, M Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering (Volume:37, Issue:5,Sept.–Oct.2011), Page(s): 649–678.
[3] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce, MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs, ISSTA 2014 Proceedings of the 2014 International Symposium on Software Testing and Analysis, Pages 429–432.
[4] 梅村 葵，大久保 弘崇，粕谷 英人，山本 晋一郎,『ミューテーション法の拡張によるテスト結果の変化に着目したソフトウェア修正法提示』, 情報処理学会研究報告書, Vol.2015-SE-187, No.25, 2015/3/12.
[5] K. Adamopoulos, M. Harman, and R. M. Hierons, “How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-

evolution," in GECCO (2). Springer, 2004, pp. 1338–1349.
[6] K Claessen, J Hughes - Acm sigplan notices, QuickCheck: a lightweight tool for random testing of Haskell programs, ICFP '00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, Pages 268–279.
[7] C Runciman, M Naylor, F Lindblad - Acm sigplan notices, Smallcheck and lazy smallcheck: automatic exhaustive testing for small values, Haskell '08 Proceedings of the first ACM SIGPLAN symposium on Haskell, Pages 37–48.
[8] D. Herington. HUnit: A unit testing framework for haskell, http://hackage.haskell.org/package/HUnit, January 2019.

付　　録

A.1 ミューテーション一覧

ミューテーション関数名	種類	修正
MuCheck 組み込みミューテーション		
selectFunctionOps	基本ミューテーション	有り
selectFnMatches	特有ミューテーション	無し
selectGuardedBoolNegOps	特有ミューテーション	有り
selectIfElseBoolNegOps	基本ミューテーション	無し
selectLiteralOps	基本ミューテーション	有り
MuCheck+に追加したミューテーション		
selectReplaceValOps	基本ミューテーション	-
selectValBoolOps	基本ミューテーション	-

A.2 期末試験の問題文と使用したテストスイート及び模範解答一覧

```
2018期末試験 Q3
問題文

標準関数`elem`を再定義せよ。

import Test.QuickCheck
import Prelude hiding (elem)
import qualified Prelude as DONTUSE (elem)

prop = do
  NonNegative v
    <- arbitrary :: Gen (NonNegative Int)
  xs <- arbitrary :: Gen [Int]
  let real = DONTUSE.elem v xs
  let ans = elem v xs
-- エラー時のメッセージ
  let errmsg = unwords ["elem", show v
  , show xs, "is", show real, "
␣␣,␣but␣returned", show ans]
-- 比較実行
  return $ whenFail (putStrLn errmsg)
    $ ans == real

模範解答

elem t (x : xs) = t == x || elem t xs
elem _ [] = False

テストケース
> print $ elem 'k' "Haskell"
< True
> print $ elem 'E' "Haskell"
< False
> print $ elem False [True,True,True]
< False
> check prop
< OK
```

図 A·1　elem

```
2018期末試験 Q5
問題文

標準関数`zip`を再定義せよ。
(この問題では`zipWith`も使用できない。)

import Test.QuickCheck
import Prelude hiding (zip, zipWith)
import qualified Prelude as DONTUSE (zip)

prop = do
  xs <- arbitrary :: Gen [Int]
  ys <- arbitrary :: Gen [Char]
  let real = DONTUSE.zip xs ys
  let ans = zip xs ys
-- エラー時のメッセージ
  let errmsg = unwords ["zip", show xs
  , show ys, "is", show real, "
␣␣,␣but␣returned", show ans]
-- 比較実行
  return $ whenFail (putStrLn errmsg)
    $ ans == real

模範解答

zip :: [Int] -> [Int] -> [(Int,Int)]
zip []     _bs    = []
zip _as    []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

テストケース
> print $ zip "ABC" [1..5]
< [('A',1),('B',2),('C',3)]
> check prop
< OK
```

図 A·2 zip

```
2018期末試験 Q7
問題文

あなたは鉛筆がn本必要になったので
文房具店に買いに行った。

鉛筆はバラ売りはしておらず、
四菱ハイユニはa本セットでx円、
スッテのルモグラはb本セットでy円である。

鉛筆の種類が混ざるのは嫌なので、
いずれかのセットを選び
n本以上になるようにそれを
複数購入することにする。
(n本を超えて余分に購入する
かもしれないが仕方ない。)

安く済ませたとき、
代金はいくらになるかを求める関数
`pencils :: Int -> (Int, Int) ->
            (Int, Int) -> Int`
を作れ。
引数は`pencils n (a,x) (b,y)`のように与える。

<!-- 日本情報オリンピック 2017 予選問題 1 -->

○ テンプレート

import Test.QuickCheck

fnalnasdlkfjewkl n (a,x) (b,y)
  = min (x * sub a) (y * sub b) where
    sub k = (n+k-1) `div` k

prop = do
  n <- choose (1,1000 :: Int)
  a <- choose (1,1000 :: Int)
  b <- choose (1,1000 :: Int)
  x <- choose (1,1000 :: Int)
  y <- choose (1,1000 :: Int)
  let real = fnalnasdlkfjewkl n (a,x) (b,y)
-- 答案作成
  let ans = pencils n (a,x) (b,y)
-- エラー時のメッセージ
  let errmsg = unwords ["pencils", show n
  , show (a,x), show (b,y), "is"
  , show real, ",␣but␣returned", show ans]
-- 比較実行
  return $ whenFail (putStrLn errmsg)
    $ ans == real

模範解答

pencils n (a,x) (b,y)
  = min (x * sub a) (y * sub b) where
    sub k = (n+k-1) `div` k

テストケース
> print $ pencils 10 (3,100) (5,180)
< 360
> print $ pencils 6 (2,200) (3,300)
< 600
> check prop
< OK
```

図 A·3 pencils