Vol.2015-SE-187 No.25
2015/3/12

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

梅村 葵[1,a)]　大久保 弘崇[2,b)]　粕谷 英人[2,c)]　山本 晋一郎[2,d)]

概要：ミューテーション法によるテスト結果の変化に着目した，ソフトウェアの修正方法を提示する手法を提案する．修正方法の提示により，欠陥の探索に要する負担を軽減し，修正作業に伴うソフトウェアの理解を支援する．その結果としてソフトウェアの修正作業にかかる負担を軽減する．提案手法は，ミューテーション法へミュータントの分類を追加し，その分類に基づきミュータントに対する評価値を定義する．この評価値は，ミュータントのソフトウェアに対する修正としての適性を数値化したものである．実験により，提案手法の欠陥局所化能力と提示した修正方法の正しさを評価し，修正の適性を表す評価値の性質と提案手法の計算時間について考察した．

1. はじめに

1.1 背景

ソフトウェアの保守はソフトウェアライフサイクルにおいて財政的資源のほとんどの部分を消費する重要な工程である [1]．ソフトウェアの保守は一般的に適応保守，是正保守，緊急保守，完全化保守，予防保守に区分されるが，これらすべてに共通して保守対象であるソフトウェアの修正が伴う．修正を行うためには，対象に対する十分な理解が必要となる．しかし，先述の通り保守工程は長期に渡って継続する可能性があり，次のような問題が発生する場合がある．

修正対象の一部のレガシー化　レガシー化とは，旧来の技術で作成されていることを表す．保守フェーズにあるソフトウェアプロジェクトに新規に参加したメンバーは，対象ソフトウェアに関する知識を持たない．しかし，開発時とはメンバーが異なりレガシー化した部品の知識が残されていない可能性もあり，そのような状況は対象の理解を妨げる．

仕様書など関連ドキュメントの不備　多くのソフトウェアには，仕様書や設計書など対象ソフトウェアについてのドキュメントが付属する．しかし，それらドキュメントに不備がある場合，ドキュメントによる対象の理解を阻害，または間違った理解へ誘導する場合がある．これらの問題に加えソフトウェアの大規模化や複雑化により，ソフトウェアを理解することが困難となっており，修正対象の理解が不十分なままとなる場合がある．理解が不十分な状態での修正は新たな欠陥を混入する可能性があり，結果的にソフトウェアの品質低下につながる．

対象の理解が不十分な場合に修正を支援するさまざまな研究がなされており，その中の 1 つがスペクトルによる欠陥局所化である．スペクトルとは，ソフトウェアテストの実行結果，カバレッジ，プログラムスライシングなどを指し，欠陥局所化とはソフトウェアに含まれる誤りの原因となっている箇所の候補を提示する手法である．スペクトルによる欠陥局所化の特徴は対象に関する理解がなくとも利用できる点である．その 1 つとして，Agrawal らが提案したユニオンモデル [2] が挙げられる．これは，テストの実行結果とカバレッジをスペクトルとして利用する．

スペクトルによる欠陥局所化手法は多くの手法が提案されている [3]．しかし，いずれの手法にも共通して，欠陥がテスト成功時に実行される箇所に含まれる場合の検出に難がある．また，スペクトルによるものに限らず全ての欠陥局所化において，提示された候補からは欠陥の内容に関する情報が得られない．そのため，提示された候補の修正方法を検討するために対象の理解する必要がある．これら既存の欠陥局所化手法における問題に関して，ソフトウェアの修正に対する支援の改善が期待される．

[1] 愛知県立大学大学院情報科学研究科
Graduate School of Information Science and Technology, Aichi Prefectural University
[2] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
a) umemura@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

1.2 目的

本論文では，開発工程ではなく保守工程にあるような，ある程度の品質が確保されたソフトウェアに対して，欠陥の候補を提示するのみでなく，どのように修正すべきかを提示する手法を提案する．すなわち，開発が完了し，テストセットに含まれる全てのテストに成功している状況において，バグレポートや品質向上を目的としたテストの追加などにより，成功しているテストに比べ少数の失敗するテストが新たに追加された場合にソフトウェアの修正方法を提示する．これにより，欠陥の発見に要する理解や負担の軽減のみでなく，修正方法の検討にかかる理解の支援や負担の軽減が可能となる．

1.3 本論文の貢献

本論文の貢献は次の 3 点である．

- スペクトルによる欠陥局所化の問題を改善した，新たな仮説による修正箇所提示手法を提案した．提案手法により，ソフトウェアの修正における欠陥の特定と修正方法の決定にかかる負担を軽減できる．
- ミューテーション法に新たなミュータントの分類を追加した．新たな分類は，ミューテーション法で考慮されていない場合に対するものであり，これを用いた新たな解析や評価を可能とする．
- ミューテーション法で生成されるミュータントに対し，修正スコアと呼ぶ新たな評価値を定義した．この評価値により，ミューテーション法をソフトウェアの修正へ利用することが可能となる．

2. 本論文のアプローチ

2.1 スペクトルによる欠陥局所化の欠点

1.1 節にて述べたとおり，既存の欠陥局所化手法は，本質的に，テスト成功時の実行箇所にある欠陥の検出が難しいという欠点がある．ユニオンモデルを例に説明する．ユニオンモデルは，1 つの失敗したテストが実行した箇所から，成功したテストのいずれか 1 つでも実行した箇所を除外することで欠陥を局所化する．図 1 は，3 つの整数を引数として受け取り最小値を返す関数 `triMin` による例である．`triMin` は 5 行目に欠陥が存在し，正しくは `m = z` とするべき箇所が `m = y` となっている．この例では，失敗したテスト 1 つと成功したテスト 2 つを用いて，欠陥候補として挙げることに成功している．

次に，ユニオンモデルで欠陥の推定に失敗している例を示す．図 2 では，4 行目の条件式が本来ならば `z < m` であるが `m < m` となっている．先ほどの例と同じテストケースを用いて欠陥局所化を行うと，欠陥の候補が 1 つも存在しないという結果となる．このように，ユニオンモデルでは，テスト成功時の実行パス上に存在する欠陥を候補として挙げることができない．

		テストケース			
	`int triMin(int x, int y, int z) {`	2,3,1	2,1,5	2,4,7	欠陥候補
	`  int m;`				
1 :	`  m = x;`	●	●	●	
2 :	`  if (y < m)`	●	●	●	
3 :	`  m = y;`		●		
4 :	`  if (z < m)`	●	●	●	
5 :	`  m = y;  /* bug */`	●			×
6 :	`  return m;`	●	●	●	
	`}` S/F	F	S	S	

図 1 ユニオンモデルによる欠陥局所化の成功例

		テストケース			
	`int triMin(int x, int y, int z) {`	2,3,1	2,1,5	2,4,7	欠陥候補
	`  int m;`				
1 :	`  m = x;`	●	●	●	
2 :	`  if (y < m)`	●	●	●	
3 :	`  m = y;`		●		
4 :	`  if (m < m)  /* bug */`	●	●	●	
5 :	`  m = z;`				
6 :	`  return m;`	●	●	●	
	`}` S/F	F	S	S	

図 2 ユニオンモデルによる欠陥局所化の失敗例

ユニオンモデルが欠陥の候補をあげることに失敗する原因は，ユニオンモデルを含むスペクトルによる欠陥局所化の背景にある次の仮説と考えられる．

a　失敗したテストの実行パス上に欠陥がある可能性は高い

b　成功したテストの実行パス上に欠陥がある可能性は低い

テストの実行環境による外的要因を除けば，欠陥を実行したことが原因となりテストが失敗すると考えられるため，a の仮説はもっともらしい．しかし，`if` 文や `while` 文の条件式などテストの成否に関わらず実行される箇所も存在するため，そのような欠陥は仮説 b により候補から除外されやすい．この仮説 b の不完全さが，成功時に通過した箇所に欠陥がある場合の欠点となっていると考えられる．

2.2 本論文の仮説

本論文では，スペクトルによる欠陥局所化とは異なる次の仮説を用いる．

1　ソフトウェアの一部を改変しソフトウェアテストを実行した結果，改変前のソフトウェアで失敗していたテストが改変後のソフトウェアで成功した場合，その改変によって欠陥が修正された可能性がある．

2　ソフトウェアの一部を改変しソフトウェアテストを実行した結果，改変前のソフトウェアで成功していたテストが改変後のソフトウェアで失敗した場合，その改変によって新たな欠陥を混入した可能性が高い．

この仮説はショットガン・デバッギング [4] と深い関連がある．ショットガン・デバッギングとは，バグが含まれる

ミューテーション名	ミューテーション概要	例のオリジナル状態	ミューテーション後
四則剰余演算子の交換	剰余を含む四則演算の交換	`x = x % y`	`x = x + y`
論理演算子の交換	論理演算子を他の論理演算子へ置換	`x == y`	`x != y`
比較演算子の交換	比較演算子を他の比較演算子へ置換	`x >= y`	`x > y`
論理型式の論理定数への置換	論理型の式を `true`, `false` へ置換	`x == y`	`false`
二項演算式の置換	二項演算式をその右辺または左辺へ置換	`(a != b) && c`	`c`
数値定数の 0 への置換	数値定数を 0 へ置換	`x = 100`	`x = 0`
数値定数の符号反転	数値定数を (-1) 倍	`x = 100`	`x = -100`
メソッド呼び出しの削除	メソッド呼び出しのみの文を削除	`Foo.bar()`	`<deleted>`

表 1 基本的なミューテーション

ソフトウェアに対してランダムに改変を加え，その結果偶然に欠陥が修正されることを期待するデバッグ手法であり，本来は，比較的低い成功率に対し時間を浪費し，場合によってはさらなるバグを導入する可能性のある手法である．しかし，ランダムな改変を何度も試行した結果として，どのような改変を行うとソフトウェアの振る舞いがどのように変化するかという情報を得ることができ，この得られる情報は理解を進めるためのヒントとなると考えられる．すなわち，ショットガン・デバッギングに基づく本論文の仮説は，欠陥の局所化のみでなく，理解のためのヒントを与えることが可能と考えられ，欠陥の内容に関する情報が提示されない問題への対応となる．また，本論文の仮説はテスト結果の変化に着目する．そのため，テストの成否に関わらず実行される箇所にある欠陥も検出対象に含まれ，スペクトルによる欠陥局所化手法にみられる不均一さがない．

3. ミューテーション法とその拡張

ソフトウェアに改変を加えてテストを実行し，テスト結果の変化について解析する既知の手法として，ミューテーション法がある．しかし，これは意図的に欠陥を混入しテストの品質を評価する手法であり，修正方法の提示へ利用するためには機能が不十分である．本節ではミューテーション法について述べた後，修正方法の提示へ利用するための問題と，それに対応するための拡張について述べる．

3.1 ミューテーション法

ミューテーション法では，改変を加える元となるソフトウェアをオリジナル，改変が加えられたソフトウェアをミュータント，ミュータントを生成する際に加えられた改変をミューテーションといい，ソフトウェアへ加えた改変がテストセットにより検出された割合でテストセットの品質を評価する．

ミューテーション法の手順を次に示す．

M.0 用意されたテストセットを用いてオリジナルのテストを行い，オリジナルが成功したテストのみからなるサブセットを選択する．

M.1 オリジナルから複数のミュータントを生成する．この際に用いられるミューテーションはツールにより異なるが，よく使用される 8 種類の基本的なミューテーションを表 1 に示す．

M.2 M.0 にて定めたサブセットを用いて各ミュータントのテストを実行し，テスト結果に応じてミュータントを次のように分類する．

Kill ミュータントがテストセットに含まれるいずれかのテストに失敗した．テストセットがミュータントに含まれるミューテーションを発見できた．

Alive ミュータントがテストセットに含まれる全てのテストに成功した．テストセットがミュータントに含まれるミューテーションを発見できなかった．

M.3 各ミュータントの分類から，生成されたミュータントに対するテストセットの評価値であるミューテーションスコアを算出する．ミューテーションスコアはミュータントの総数に対する Kill に分類されたミュータントの数の割合と定義され，生成されたミュータントの数を M，そのうち Kill と分類されたミュータントの数を M_k とすると，式 1 のようになる．

$$MS = \frac{M_k}{M} \tag{1}$$

ミューテーションスコアが高いテストセットは，ソフトウェアへ間違って加えられる改変の多くを発見できる質の良いテストセットとされる．

3.2 ミューテーション法の拡張

ミューテーションスコアはテストセットの評価値であり，ミュータントの評価値ではない．ミューテーション法を修正法の提示へ利用するためには，オリジナルに含まれる欠陥の修正方法として各ミュータントがどれほど適切かを評価する必要がある．本論文では，個々のミュータントに対する，テスト結果の変化に基づく評価値を 2 つ定義し，次に，この評価値を用いてソフトウェアに対する修正候補としてのミュータントの妥当性を表す修正スコアを定義する．

3.2.1 テスト結果変化の分類

3.1 節にて述べた元のミューテーション法によるミュータントの分類は，オリジナルが成功したテストのみを対象としており，オリジナルが失敗したテストについては利用されない．また，テストセットに対するミュータントの分

		ミュータントのテスト結果	
		成功	失敗
オリジナルのテスト結果	成功	Alive	Repair
	失敗	Kill	Failure

表 2 テストに対するミュータントの分類

類であり，個々のテストに対するミュータントの分類ではない．

個々のテスト結果に対してミュータントを次のように分類する．

Alive オリジナルが成功したテストにミュータントも成功した．ミューテーションがテストを失敗させる要因とならないと考えられる．

Kill オリジナルが成功したテストにミュータントが失敗した．ミューテーションがテストを失敗させた可能性があると考えられる．

Repair オリジナルが失敗したテストにミュータントが成功した．このテストに限ってはミュータントが欠陥を修正したと考えられる．

Failure オリジナルが失敗したテストにミュータントも失敗した．このテストに限ってはミュータントが欠陥を修復しないと考えられる．

分類 Alive と Kill はどちらのミューテーション法にも存在するが，元のミューテーション法は欠陥のないプログラムにてテストセットを，提案手法では欠陥のあるプログラムにてミュータントを評価することから意味合いが異なる．元のミューテーション法において良い結果とは，テストセットのミューテーションスコアが高いことと言える．ミューテーションスコアの増加につながることからミュータントが分類が Kill となることが望ましく，MS の低下につながることから Alive と分類されることは望ましくない．

一方，提案手法において良い結果とは，欠陥の修正により別の欠陥が発生するような間違った修正ではなく，正しい修正方法を提示できることである．そのため，修正方法の候補であるミュータントがテストに失敗していないことを示す分類 Alive が望ましく，修正候補にもかかわらずテストに失敗していることを示す分類 Kill は望ましくない．

テストセット評価では，常にテストセット全体を一式でとらえ，ミュータントはテストセットのいずれかで失敗したか否かにより分類される．対して提案手法では，テストセットに含まれる個々のテストに対するミュータントの実行結果が分類される．

3.2.2 修復率・成功維持率

本節では，各テストに対するミュータントの分類を集計する 2 つの新たな評価値を定義する．1 つ目の評価値は，オリジナルが失敗したテストに対する，あるミュータント m がその中で成功した，すなわち Repair に分類されたテストの割合であり，これを修復率 $Rep(m)$ として式 (2) のように定義する．式中の N_f はオリジナルが失敗したテストの数，m_R は各テストに対しミュータント m が Repair と分類された数を表す．

$$Rep(m) = \frac{m_R}{N_f} \tag{2}$$

修復率はミュータントに含まれるミューテーションが欠陥を修正した程度を表し，値域は $0 \leq Rep(m) \leq 1$ となる．最小値である $Rep(m) = 0$ の場合はオリジナルが失敗したテスト全てに同じく失敗しているため，ミューテーションが欠陥の修正に全く貢献していない，あるいは欠陥を修正するほどの影響力を持っていないと解釈できる．一方，最大値である $Rep(m) = 1$ の場合はオリジナルが失敗したテスト全てに成功しているため，オリジナルがテストに失敗する原因である欠陥の全てをミューテーションが修正したと解釈できる．

2 つ目の評価値は，オリジナルが成功したテストに対する，あるミュータント m がその中で成功した，すなわち Alive に分類されたテストの割合であり，これを成功維持率 $SKeep(m)$ として，式 (3) のように定義する．式中の N_p はオリジナルが成功したテストの数，m_A は各テストに対しミュータント m が Alive と分類された数を表す．

$$SKeep(m) = \frac{m_A}{N_p} \tag{3}$$

成功維持率はミュータントに含まれるミューテーションが修正前から悪化させていない程度を表し，値域は $0 \leq SKeep(m) \leq 1$ となる．最小値である $SKeep(m) = 0$ の場合はオリジナルが成功したテスト全てに失敗しているため，ミューテーションが新たな欠陥を混入していると解釈できる．一方，最大値 $SKeep(m) = 1$ の場合はオリジナルが成功したテスト全てに成功しているため，ミューテーションが新たな欠陥を混入していない解釈できる．

3.2.3 修正スコアの定義

修復率はオリジナルが失敗したテストに基づく評価値であるため，オリジナルが成功したテストについては一切考慮していない．そのため，ミュータントの修復率が最大値 1 であったとしても，オリジナルが成功したテストに失敗している可能性がある．対照的に，成功維持率はオリジナルが成功したテストに基づく評価値であるため，オリジナルが失敗したテストについては一切考慮していない．そのため，ミュータントの成功維持率が最大値 1 であったとしても，欠陥の修復には全く貢献していない可能性がある．

ソフトウェアの修正作業は，欠陥の改善と修正作業による影響を考慮する必要があることから，提案手法における最終的な評価値である修正スコアは，この両方の適性を一元的に表す必要がある．すなわち，成功維持率と修復率の高いときに大きく，いずれか一方でも低いときに小さくなる関数を選択する．ここでは両者の積を検討する．

$$Score_m(m) = Rep(m) \times SKeep(m) \tag{4}$$

なお，4.4 節にて他の定義と比較する．式 (4) にて定義される修正スコアは，$0 \leq Score_m(m) \leq 1$ の値域を持ち，修復率と成功維持率がともに 1 の場合に最大値をとる．$Score_m(m)$ が最大値であることは，与えられたテストセットで観察する限りにおいて，欠陥を修正し，かつ他の部分へ悪影響を及ぼさない正しい修正方法であると言える．修正スコアが 1 でないミュータントは，修復率か成功維持率のいずれか，あるいは両方が最大値 1 に満たない．すなわち，欠陥の修正が不完全であるか他の部分へ影響を及ぼしているため，それらへ対処するための更なる修正が必要となる．また，1.2 節で述べたように，提案手法の適用場面として仮定している成功するテストよりも失敗するテストが少数である状況では，1 テストあたりの成否による変化量が成功維持率に比べ修復率が大きい．そのため，Alive のテストが 1 増えるよりも，Repair のテストが 1 増える方が $Score_m$ は大きく増える．

4. 適用実験と評価

ソフトウェアテスト研究で広く使われている欠陥を含めたプログラムとテストの標準セットを用いて，提案手法を評価する．本節では，最初に評価に用いたプログラムと使用したミューテーションについて述べ，次に評価と結果の考察を行う．はじめに提案手法の欠陥局所化能力について評価する．提案手法により提示された修正スコアが最も高いミュータントから欠陥に関する情報を得ることが可能であるかを調べる．次に，修正スコアが 1 のミュータントが正しい修正方法であるかを確認し，提案手法が正しい修正方法を提示するための条件について考察する．最後に，修正スコアの定義の代替案をふたつ示し，3 つの定義の欠陥局所化能力を比較することにより，効果的な修正スコアの定義について考察する．

4.1 評価用プログラムと使用したミューテーション

評価用プログラムは，Hutchins らによる Siemense Test Suite[5] に含まれるプログラム Tcas と付属のテストを Java へ書き直し使用した．Siemense Test Suite は，欠陥局所化や自動修復の研究で評価用プログラムとして広く使用されており，7 種類のプログラムとその付属物からなる．Tcas はその 1 つであり，次のものからなる．

- 欠陥のないプログラム（ゴールデンプログラムと呼ばれる）
- ゴールデンプログラムへ欠陥を注入した欠陥バージョンのプログラムが 41 個
- ゴールデンプログラムが成功するテストが 1608 個

Tcas は空中衝突防止装置のプログラムであり，11 個の整数を入力として受取り，0–2 の整数かエラーメッセージを出力する．これには `if` 文は含まれるが，`switch` 文，`for` や `while` のループ文，再帰呼び出しは含まれない．また，

欠陥		
1	オリジナル 45 行目	`Positive_RA_Alt_Thresh[3] = 740;`
	欠陥バージョン 45 行目	`Positive_RA_Alt_Thresh[3] = 700;`

表 3 Tcas 欠陥バージョン 8 の欠陥注入

欠陥		

1–4	オリジナル 42-45 行目	`Positive_RA_Alt_Thresh[0] = 400;`
		`Positive_RA_Alt_Thresh[1] = 500;`
		`Positive_RA_Alt_Thresh[2] = 640;`
		`Positive_RA_Alt_Thresh[3] = 740;`

	欠陥バージョン 42-45 行目	`Positive_RA_Alt_Thresh[1] = 400;`
		`Positive_RA_Alt_Thresh[2] = 500;`
		`Positive_RA_Alt_Thresh[3] = 640;`
		`Positive_RA_Alt_Thresh[4] = 740;`

表 4 Tcas 欠陥バージョン 33 の欠陥注入

Tcas に付属するテストセットは C0 カバレッジが 97%，C1 カバレッジが 93%となっている．Tcas の欠陥バージョンには，欠陥が 1 箇所のものが 34 個，2 箇所のものが 3 個，3 箇所のものが 3 個，4 箇所のものが 1 個あり，欠陥の程度も様々である．最も軽微なものはバージョン 8 であり，付属のテスト 1608 個のうち 1 個を失敗させる．その欠陥は表 3 のように注入されている．また，最も重篤なものはバージョン 33 であり，付属のテスト 1608 個のうち 1570 個を失敗させる．その欠陥は表 4 のように，4 行に渡り 4 つの欠陥が注入されている．

ミューテーション解析には，René らによる Java を対象としたミューテーションテストフレームワークである Major[6] を使用した．その理由としては，ミュータントの生成が非常に高速であること，また，テストの実行についても，Java のユニットテストライブラリである JUnit と連携しているものが多く，テストに利用できる機能も豊富に用意されており評価へ利用しやすいことが挙げられる．

今回の評価は，表 1 に示した全てのミューテーションを用い，1 つのミュータントの作成に 1 つだけのミューテーションを適用する SOM (Single Order Mutation) にて行った．一般的に，ミューテーションの種類を増やすことで修正方法の候補が増え，より効果的な修正方法を提示することが可能となる．また，1 つのミュータントの作成に複数のミューテーションを適用する HOM (Higher Order Mutation) を採用することにより，様々なミューテーションを組み合わせたミュータントを作成し，複数箇所に散らばる欠陥に対する修正方法を提示することが可能となる．しかし，ミュータントの増加は修正方法の計算に必要な時間を増大させるため，現実的な時間で計算を行うためにはある程度の制限が必要となる．今回の評価では，表 1 に示した 8 種類のミューテーション，かつ SOM とすることにより，現実的な時間で計算を行うための制限としている．

生成されるミュータントの数は，各欠陥バージョンに注入された欠陥の影響により増減する．表 1 に示したミューテーションを用いて各欠陥バージョンのミュータントを生成したところ，平均で 198 個程度のミュータントが生成され，最大は欠陥バージョン 31 と 32 の 206 個，最小は欠陥

図 3 Tcas ゴールデンプログラム

```
67 ... && (!(Down_Separation >= ALIM())));
```

図 4 Tcas 欠陥バージョン 1

```
67 ... && (!(Down_Separation > ALIM())));
```

図 5 欠陥バージョン 1 で修正スコア 1 のミュータント

```
67 ... && (!(Down_Separation >= ALIM())));
```

図 6 Tcas ゴールデンプログラム

```
67 ... && (Cur_Vertical_Sep >= MINSEP) && ...;
```

図 7 Tcas 欠陥バージョン 4

```
67 ... && (Cur_Vertical_Sep >= MINSEP) || ...;
```

図 8 欠陥バージョン 4 で修正スコア 1 のミュータント

```
67 ... && (Cur_Vertical_Sep >= MINSEP);
```

バージョン 40 の 188 個であった．

4.2 提示する修正箇所の正しさ

前節で述べた通り，提案手法が提示する修正方法の精度は，使用するミューテーションの種類や，SOM か HOM のどちらを使用するかといった条件の影響を受ける．そのため，現実的な計算時間へ抑えるための制限により，修正スコアが 1 であるミュータントが生成されず，正しい修正方法の候補の提示が不可能な場合がある．しかし，修正スコアが 1 のミュータントが存在しない場合でも，欠陥に関連するミュータントを提示することにより，欠陥の候補を提示する欠陥局所化が可能な場合がある．そこで，提案手法による行単位での欠陥局所化能力を評価するために，欠陥と同じ行をミューテーションした全てのミュータントを局所化ミュータントとし，その検出能力を確認した．評価は，修正スコアの降順にミュータントを順位付けし，欠陥バージョン 41 個のうち局所化ミュータントが 1 位であった個数，1 位から 5 位までに局所化ミュータントが 1 つ以上存在した個数を調べた．結果は，欠陥バージョン 41 個のうち，1 位が局所化ミュータントであった場合が 23，5 位までに局所化ミュータントが存在した場合が 35 であった．すなわち，1 位がミューテーションを行っている行に欠陥が含まれるとした場合の精度が $23/41 = 0.56$，5 位までがミューテーションしている行に欠陥が含まれるとした場合の精度が $35/41 = 0.85$ であることを示しており，5 位までを候補とするならばある程度の精度を持つと言える．

4.3 提示する修正方法の正しさ

提案手法が正しい修正方法を提示するかを評価するために，提案手法によって修正スコアが 1 の修正として提示されたミュータントが正しい修正であるかを確認した．

修正スコアが 1 のミュータントが存在し，かつ正しい修正であった実例として欠陥バージョン 1 を示す．欠陥バージョン 1 の欠陥を含む行を図 4 に，修正スコアが 1 であったミュータントのミューテーション行を図 5 に，ゴールデンプログラムの当該行を図 3 に示す．図 3 と図 5 を比べると，欠陥バージョン 1 にて提示された修正法を適用した結果はゴールデンプログラムと同じになるため，正しい修正を提示している．

一方，修正スコアが 1 のミュータントが提示されたにも関わらず，ゴールデンプログラムとは一致しないものも確認された．例として欠陥バージョン 4 を示す．図 6 と図 8 を比べると，AND 論理式をその左辺へ置換しており，一致しないため正しい修正方法の提示ではない．

全体では，欠陥バージョン 41 個のうちの 17 個にてスコアが 1 のミュータントが提示され，このうちの 13 個が正しい修正という結果となった．すなわち，提案手法によるスコアが 1 のミュータントが正しい修正である割合は $13/17 = 76.5\%$ となる．100%ではない理由として，テストセットの網羅性が挙げられる．欠陥バージョン 4 で修正スコアが 1 のミュータントが提示された原因は，図 6 と図 8 で差が出るようなテストケースがテストセットに含まれていなかったことである．これを検証するために，新たに 21 個の境界値入力のテストを追加したところ，スコアが 1 の修正ながら正しい修正ではなかった 4 個のミュータントはいずれもスコアが 1 ではなくなり，提案手法によるスコアが 1 の修正の精度は $13/13 = 100\%$ となった．よって，提案手法の提示するスコアが 1 のミュータントが正しい修正である精度はテストセットの網羅性に依存し，テストセットの網羅性を高めることで，提案手法により提示される修正スコアが 1 のミュータントが正しい修正である可能性が高まると言える．

4.4 修正スコア算出方法の比較

提案手法の修正スコアは，修復率と成功維持率が共に高いときに大きく，どちらか一方でも低いときに小さくなる関数として，修復率と成功維持率の積と定義した．しかし，この性質を満たす関数は積以外にも存在する．有効な修正スコアの性質を考察するため，積の他に相加平均，ユークリッドノルムによる修正スコアを定義し評価を行った．

$$Score_a(m) = \frac{Rep(m) + SKeep(m)}{2}$$

$$Score_e(m) = \frac{\sqrt{Rep(m)^2 + SKeep(m)^2}}{\sqrt{2}}$$

評価は各ミュータントで修正スコアを算出し降順に順位付けし，4.2 節にて用いた 1 位が局所化ミュータントであった個数と，局所化ミュータント全ての順位の平均を調べた．

	積	平均	ユークリッドノルム
平均順位	16.83	24.67	20.08
1 位の個数	23	23	23

表 5 修正スコア定義別の平均順位と 1 位取得率の比較

MD_v は欠陥バージョン v の局所化ミュータントの集合，$rank(v, m)$ を欠陥バージョン v におけるミュータント m の順位とする．

$$\text{平均順位} = \frac{\sum_{v=1}^{41}\sum_{i=1}^{|MD_v|} rank(v, md_{v,i})}{\sum_{v=1}^{41}|MD_v|} \quad (md_{v,i} \in MD_v)$$

比較結果を表 5 に示す．これらより，積の定義が最も良い性能であると言える．その理由として，修復率か成功維持率のどちらか一方が低く他方が高い場合の修正スコアが挙げられる．積の定義は $Score_m(m) \leq \min\{Rep(m), SKeep(m)\}$ となり，修正スコアは修復率か成功維持率のどちらか低いものよりも低くなる．一方，他の 2 つの定義は $\min\{Rep(m), SKeep(m)\} \leq Score_a, Score_e \leq \max\{Rep(m), SKeep(m)\}$ となり，高いものが低いものを補うため修正スコアが中程度となる．すなわち，3.2.3 節にて述べた，2 つの条件のどちらか一方でも満たさない場合は低い修正スコアを割り当てるという以下の性質を，積の定義は満たしており，他の定義は満たしていない．そのため，積の定義では，2 つの条件を満たすミュータントはどちらか一方しか達成していないものよりも相対的に高い修正スコアを持つことになる．結果的に修正方法として適正が高いものを検出できていると考える．

4.5 適用対象の規模とテストの数による計算時間

今回の評価は対象プログラムを Tcas とし，表 1 に示した 8 種類のミューテーションによる SOM という条件にて行った．しかし，Tcas は行数が 176 と小規模であり，用意された欠陥バージョンの多くは欠陥が 1 箇所であり内容も軽微なものが多い．そのため，実際に運用されているプログラムへ適用する際に注意すべき点が 2 つある．1 つは，対象規模の増加や複雑な欠陥への対応によるミュータントの増加である．対象が大規模な場合は，ミューテーションの対象となる箇所が増加するため，ミュータントが増加する．また，複雑かつ複数箇所に及ぶ欠陥へ対応するためには，ミューテーションの種類を増やし，ミュータントを HOM にて生成する必要があると考えられるため，ミュータントが増加する．2 つは，テストの増加である．4.3 節にて，テストの網羅性を高めることにより，修正スコアが 1 のミュータントが正しい修正である精度を高めることが可能と述べた．しかし，テストの網羅性を向上させるためにはカバレッジを向上させるテストを追加する必要があるため，要求される精度が高いほど多くのテストを追加する必要がある．提案手法の計算時間は生成されるミュータントの数，テストの数，テストの平均実行時間の積であるため，これら 2 点は実行時間を増加させる要因となる．ミュータント数やテスト数が計算時間へ与える影響を確認するために，Tcas よりも規模が大きく複雑な試験用プログラムとして Siemense Test Suite に含まれる Replace を選択し，Tcas と Replace ともにオリジナルに対して提案手法を適用し計算時間を比較した．今回の評価は表 6 の仮想マシンにて行い，仮想マシン環境は VMWare Player を使用した．また，Replace はループを含むため，テストには 250 [ms] のタイムアウトを設定した．結果を表 7 に示す．これより，Replace は Tcas の 36 倍の計算時間を要することがわかる．原因の 1 つはテストの数やミュータントの数の増加であるが，もう 1 つはミューテーションの影響による無限ループの発生である．その場合はタイムアウトによりテストが強制的に失敗するまで待つ必要があり，テストの平均実行時間の増加につながる．結果的に，大規模かつ複雑なプログラムにて精度の高い計算を行う際には，実行時間を決定する 3 つ要素全てが増加するため，計算時間が大幅に増加することを確認した．しかし，提案手法にて行う処理のうち最も時間を要するテストの実行は並列化が可能であり，ミュータントの生成や修正スコアの算出など他の処理は計算時間が短い．そのため，並列処理による計算時間の短縮が効果的と考えられる．

	物理マシンスペック	仮想マシンスペック
OS	Windows 7 64bit	Ubuntu 14.04 64bit
プロセッサ	Intel Xeon 12core	VMcore 8core
メモリ	24 [GByte]	12 [GByte]

表 6 評価を行った計算機のスペック

	Tcas	Replace
行数	176	642
テストの数	1608	3133
生成されるミュータントの数	196	756
計算時間	30 [分]	18 [時間]

表 7 Tcas と Replace の規模と計算時間に関する比較

5. おわりに

5.1 まとめ

本論文では，既存の欠陥局所化が持つ問題へ対応するために，ミューテーション法を拡張することによる新たな修正提示手法を提案した．提案手法では，テストに対するミュータントの新たな分類の定義，修復率と成功維持率の定義，及びそれらを用いた修正スコアの定義によりミューテーション法を拡張した．提案手法の評価を行い，1 つ目の結果として，提案手法の欠陥局所化能力は，修正スコアの降順に順位付けした場合の 1 位に限定した場合には 0.56 程度であるが，5 位までを候補とするならば 0.85 とある程

度の精度を持つことが確認された．2 つ目の結果として，提案手法により提示される修正スコアが最大値の修正方法は，テストセットの網羅性を高めることにより正しい修正である可能性が高くなることが確認された．3 つ目の結果として，提案手法による修正スコアは，修復率と成功維持率の両方が高いものほど高い修正スコアを割り当て，さらに，どちらか一方でも満たさない場合に低い修正スコアを割り当てる定義が良いことが確認された．4 つ目の結果として，大規模なプログラム，様々な欠陥への対応，精度の高い修正方法の検出が求められる際には，提案手法の計算時間を決定する要素が増加するため，計算時間が大幅に増加することが確認された．

5.2 関連研究

本論文では，ミューテーション法とその拡張を修正方法の提示へ利用した．一方で，ミューテーション法を欠陥局所化へ利用した研究が存在する．Debroy ら [7] により提案されたミューテーション法による欠陥局所化は，スペクトルによる欠陥局所化で用いられている ochiai の式 [8] によりミュータントの欠陥としての疑わしさを算出し，その疑わしさを考慮したスコアを用いて各行を順位づけする．また，Tao による疑わしさの定義 [9] は，オリジナルが成功していたテストを失敗させた場合に，ミュータントは問題のない部分へミューテーションを行ったとして，Lee らによる Naish の疑わしさ [10] からミューテーションによる影響量を減算している．しかし，提示する修正方法がどれほどの修正を行うかについて考慮しておらず，オリジナルが失敗したテストについては元のミューテーション法と同様考慮していない．

5.3 今後の課題

今後の課題として 2 点を挙げる．まず 1 点は，使用するミューテーションの種類や HOM の使用による提案手法の評価である．今回の評価にて用いた 8 種類のミュータントと SOM という制限下では，欠陥バージョン 41 個のうち 27 個は修正スコアが 1 のミュータントを提示不可能であった．しかし，使用するミューテーションの種類を増やす，あるいは HOM を使用することにより，生成されるミュータントが増加し様々な修正方法の候補を検討することが可能となる．そのため，修正方法の候補が増えることにより修正スコアが 1 のミュータントが生成される可能性があり，修正スコアが 1 のミュータントを提示できなかった 27 個についても，正しい修正方法を提示可能な可能性が高まると考えられる．2 点目は，既存の欠陥局所化手法との比較である．既存の欠陥局所化手法は提案手法と異なり，主に行を対象としている．しかし，提案手法は 1 行に対し複数の修正方法を提示しており，行に対する欠陥局所化のための評価値を持っていない．そのため，欠陥局所化に特化した評価値を提案手法に定義し，行に対する欠陥局所化という同一の観点による評価を行う必要があると考える．

謝辞 本研究は科研費 24300006 の助成を受けたものである．

参考文献

[1] 日本工業標準調査会. 日本工業規格 jis x0161:2008(iso/iec 14764：2006) ソフトウェア技術－ソフトウェアライフサイクルプロセス－保守. 2008.

[2] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pp. 143–151, Oct 1995.

[3] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pp. 273–282, New York, NY, USA, 2005. ACM.

[4] Shotgun debugging - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Shotgun_debugging/`.

[5] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pp. 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 612–615, November 2011.

[7] V. Debroy and W.E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 65–74, April 2010.

[8] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89–98, Sept 2007.

[9] Muffler a tool using mutation to facilitate fault localization 2.3. http://www.slideshare.net/elfinhe/muffler-a-tool-using-mutation-to-facilitate-fault-localization-23.

[10] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, Vol. 20, No. 3, pp. 11:1–11:32, August 2011.