自動修正適合性：新しいソフトウェア品質指標とその計測

九間 哲士[1]　肥後 芳樹[1]　柗本 真佑[1]　楠本 真二[1]　安田 和矢[2]

概要：ソフトウェア開発におけるデバッグコストの削減を目的とした自動プログラム修正に関する研究が盛んに行われている．自動プログラム修正技術は，欠陥を含むプログラムとテストスイートを入力とし，欠陥を取り除いたプログラムを自動的に出力する．これまでに多くの自動プログラム修正手法が提案され，多くの欠陥に適用されてきたが，修正の成功率は高くない．このような現状から，著者らは自動プログラム修正技術の研究開発に加えて，他のアプローチからもプログラムの自動修正を支援すべきと考えた．本論文では，自動修正適合性という新しいソフトウェアの品質指標を提案する．自動修正適合性は，対象のプログラムに対して自動プログラム修正技術がどの程度効果的に作用するのかを表す指標である．自動修正適合性を計測することにより，自動プログラム修正技術が対象のプログラムに効果的に作用するかを事前にある程度判断できるようになる．また，自動プログラム修正技術が効果的に作用するようにプログラムを開発できるようになる．本論文では自動修正適合性を自動的に計測する手法も提案する．さらに，自動プログラム修正技術で修正しやすいプログラムの構造の調査を目的として，小規模なプログラムを対象にして構造の差異による自動修正適合性の違いを分析する．

1. はじめに

ソフトウェア開発において，デバッグ作業は多大なコストを要する作業である．ソフトウェア開発コストの過半数をデバッグ作業が占めるという報告もある [1, 7]．そのため，デバッグ支援の研究が盛んに行われており，自動プログラム修正と呼ばれる技術が近年注目を集めている．自動プログラム修正とは，欠陥を含むプログラムから自動的に欠陥を取り除く技術である．これまでに多くの自動プログラム修正手法が提案されており [6]，多くの欠陥に自動プログラム修正技術が適用されてきたが，現在のところ修正の成功率は高くない．Liu らの研究では，Defects4J [10] に含まれる 395 個の欠陥のうち 25 個しか正しく修正できなかった [14]．Marginean らは自動プログラム修正技術を Facebook 社のソフトウェアに適用しているが [15]，そのような例は稀である．内藤らは，自動プログラム修正技術を企業のプロジェクトに適用するのは困難であると主張している [19]．

現状では，自動プログラム修正技術を実際に対象のプログラムに適用しない限り，自動プログラム修正技術が対象のプログラムにどの程度効果的に作用するのかを把握することはできない．そのため，「コストをかけて自動プログラム修正技術をソフトウェア開発に導入したが欠陥の修正にほとんど寄与しない」という事態が発生してしまう．このような現状から，著者らは自動プログラム修正の観点からソフトウェアを評価する指標が必要だと考えた．本論文では自動修正適合性という新しいソフトウェア品質指標を提案する．自動修正適合性は，自動プログラム修正が対象のプログラムに対してどの程度効果的に作用するかを表す．自動修正適合性は以下のような目的で利用できる．

- ソフトウェア開発に自動プログラム修正を導入するのかの判断
- 自動プログラム修正による保守を前提としたソフトウェア開発
- 自動プログラム修正の効果を高めるためのリファクタリングの研究
- 自動プログラム修正が正しくないプログラムを生成してしまう問題の原因究明

また，本論文では自動修正適合性の自動計測手法も提案する．提案手法は，ミューテーションテスト技術を利用して対象プログラムから人工的な欠陥を含むプログラムを複数生成し，それらの欠陥を自動プログラム修正技術でどの程度修正できたかを計測する．修正できた欠陥の数が多いほど，自動修正適合性が高いことを表す．

本論文では，自動プログラム修正技術で修正しやすいプログラムの構造の調査を目的として，小規模なプログラムを対象にプログラムの構造の違いによる自動修正適合性の違いを分析した結果についても述べる．

[1] 大阪大学大学院情報科学研究科 大阪府吹田市
[2] 日立製作所 東京都千代田区

2. 準備

2.1 自動プログラム修正

自動プログラム修正[*1]とは，欠陥を含むプログラムと失敗するテストケースを含むテストスイートを入力として受け取り，全てのテストケースが成功するプログラムを出力する技術である．APR は，生成と検証に基づく手法 [21] と，プログラムの意味論に基づく手法 [11] に大別される．生成と検証に基づく手法は，入力として与えられたプログラムをある戦略に基づいて書き換え，その後テストの成否を確認する手法である．この手法では，全てのテストケースが成功するプログラムが生成されるまでに大量のプログラムが生成される．意味論に基づく手法は，プログラムとテストスイートからプログラムが満たすべき条件を推測し，その条件を満たすようなプログラムを合成する手法である．

2.2 ソフトウェアテストの品質

ソフトウェアテストは，ソフトウェアが仕様通りに動作するかを検証し，ソフトウェアの欠陥を検知する役割を持つ．ソフトウェアテストにおける品質とは，対象のソフトウェアをどの程度検証できているかを表す．ソフトウェアテストの品質の評価には，テストカバレッジやミューテーションテストが用いられる．

テストカバレッジは，テストによって実行されるプログラムの要素の割合を表す．テストカバレッジが高いほどテストの品質は高いと評価される．代表的なテストカバレッジとして以下のテストカバレッジが挙げられる．

命令網羅　全ての実行可能な文 (命令) のうち、テストで実行された文の割合を表す．

分岐網羅　全ての判定条件のうち、テストで実行された判定条件の割合を表す．判定条件は `if` 文などの条件式を表す．判定条件が真の場合，偽の場合を少なくとも 1 回実行すれば分岐網羅は 100%となる．

条件網羅　全ての条件のうち、テストで実行された条件の割合を表す．1 つの判定条件が AND や OR で接続された複数の条件からなる場合，分岐網羅は判定条件全体の真偽に着目するのに対し，条件網羅は個々の条件の真偽に着目する．

ミューテーションテストでは対象のプログラムの一部を書き換え，ミュータントと呼ばれる人工的な欠陥を含むプログラムを生成する．ミュータントに対してテストを実行し，ミュータントが持つ人工的な欠陥をテストで検知できるかを確認する．機械的にミュータントを大量に生成し，テストが検知できる欠陥の割合を測定することでテストの欠陥を検知する能力を評価する．ミュータントは，ミューテーション演算子と呼ばれるプログラムの変換ルールに基づいて生成される．例えば，`n++`を `n--`に変換するミューテーション演算子や`n > 0`を`n < 0`に変換するミューテーション演算子がある．

3. 自動修正適合性

著者らはこれまでの研究において，現状の APR 技術で修正可能なバグが少ないことが企業がソフトウェア開発において APR を導入することのボトルネックになっていることを明らかにしている [19]．しかし，著者らは APR を適用する前に APR によってバグを修正できそうかをある程度判断できるようになれば，現状の APR 技術でも広く普及する可能性があると考えている．

広く普及している自動化技術として，ロボット掃除機を考えてみる．ロボット掃除機は，床掃除をする自走式のロボットである．ただし，家具が多い場合や床にものが散らかっている環境では，ロボット掃除機では十分な掃除を行うことができない．ロボット掃除機の導入を検討する際には，部屋の中に障害物が多いかどうかや床がフローリングかどうか等が導入の判断材料になる．ロボット掃除機がどのような環境であれば活躍できるのかが明らかであるからこそ，ロボット掃除機が活躍できそうな場合にのみロボット掃除機を導入できる．

現在，対象ソフトウェアのソースコードがどのように実装されていれば APR が有効に作用するのか明らかではない．著者らは APR が有効に作用するにはソースコードがどうあるべきかについて研究を行っており，本論文では自動修正適合性という新しいソフトウェア品質指標を提案する．自動修正適合性とは，対象のプログラムでバグが発生した場合に APR がどの程度そのバグを修正するのに寄与しそうかを表す指標である．自動修正適合性が高いソースコードでは，そのソースコードにおいて発生したバグが APR によって修正できる可能性が高くなる．

自動修正適合性は，第一適合度と第二適合度の 2 つの要素からなる．第一適合度は全てのテストケースが成功するプログラムの生成に APR がどの程度の効果を持つかを表す．第二適合度はオーバーフィット[*2]のないプログラムの生成に APR がどの程度の効果を持つかを表す．

例えば，自動修正適合性は以下の用途で利用できる．

ソフトウェア開発に APR を導入するかの判断

APR をソフトウェア開発に導入するにはコストを必要とする．例えば，数多く存在する APR ツールの中からプロジェクトに合ったツールを選定し，そのツールの利用方法を学習する必要がある．また，人間がソフトウェアの品質を保証するためのテストと APR に必要なテストには乖

[*1] Automated Program Repair, 以降 APR と略す

[*2] オーバーフィットとは，プログラムが全てのテストケースを通過するが正しくない振る舞いを持つ状態を指す．

離があるため，APR を導入するにはテストを拡充する必要もある [19]．自動修正適合性を利用すれば，既存のプロジェクトに対して APR がどの程度の効果を持ちそうかを事前に把握できる．対象プロジェクトの自動修正適合性が高ければ APR を導入するという判断が可能になり，コストをかけて APR を導入したが期待した効果を得られないという事態を防げる．

APR による保守を前提としたソフトウェア開発
プロジェクトの開発初期から自動修正適合性を高く保っておけば，欠陥が見つかった場合に APR を利用して欠陥を修正できる．

APR の効果を高めるリファクタリングの研究
自動修正適合性をリファクタリングを評価する指標として利用すれば，プログラムのどのような要素が APR に影響を与えるかを明らかにできる．APR で欠陥を修正しやすいようにプログラムを変換することで，これまで修正できなかった欠陥を修正できるようになる．

オーバーフィットを防止する研究
第一適合度に影響を与えるプログラムの要素と第二適合度に影響を与えるプログラムの要素の比較により，オーバーフィットを発生させやすいプログラムの要素を明らかにできる．

4. 自動修正適合性の計測

自動修正適合性の自動計測手法も提案する．提案手法のキーアイディアは，対象プログラムから人工的な欠陥を含むプログラム (ミュータント) を複数生成し，それらの欠陥をどれだけ修正できるかを計測することである．ミュータントの生成にはミューテーションテスト技術を利用する．

提案手法の流れを図 1 に示す．提案手法への入力は以下の 4 つの要素である．

- プログラム P
- テストスイート T
- ミュータント生成器 M
- APR ツール A

出力はプログラム P の自動修正適合性である．P の自動修正適合性は T，M，A に依存する値である．ミュータント生成器によってプログラムに適用されるミューテーション演算子が異なるため，生成されるミュータントの数も種類も利用するミュータント生成器によって異なる．同一の欠陥に対して同一の APR ツールを適用してもテストスイートの品質によって修正の可否が変化する可能性がある [23]．また，APR ツールによって修正の戦略が異なるため修正できる欠陥も異なる [4]．

提案手法は以下の 3 つのステップで構成される．

ステップ 1　ミュータント生成

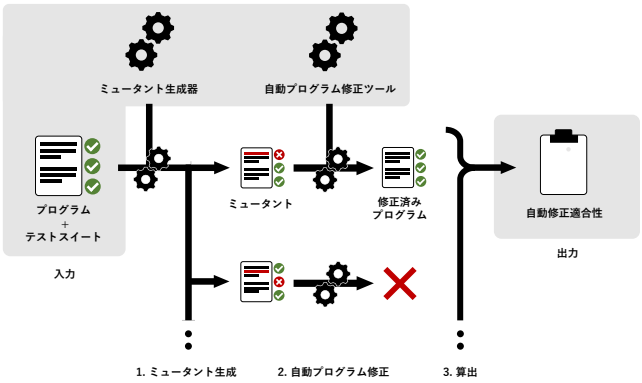

図 1　提案手法の流れ

ステップ 2　APR の適用
ステップ 3　自動修正適合性の算出
以降，各ステップについて説明する．

ステップ 1. ミュータント生成
ミュータント生成器 M を用いてプログラム P からミュータントを複数生成する．各ミュータントは異なるミューテーション演算子の適用により生成されるため，同じミュータントが複数生成されることは無い．

ステップ 2. APR の適用
ステップ 1 で生成された各ミュータントとテストスイート T を入力として APR ツール A を実行し，全てのテストケースが成功するプログラムを生成する．ミュータントに対して T に含まれる全てのテストケースが成功する場合，そのミュータントはステップ 3 では用いない．

ステップ 3. 自動修正適合性の算出
ステップ 2 の実行結果より自動修正適合性を算出する．提案手法では自動修正適合性は，生成された全ミュータント (全てのテストケースが成功するミュータントを除く) のうち A で修正済みプログラムを生成できたミュータントの割合と定義する．ここでの修正済みプログラムは，第一適合度を計測する場合は T に含まれる全てのテストケースが成功するプログラムを意味し，第二適合度を計測する場合はそのうちオーバーフィットのないプログラムを意味する．オーバーフィットのないプログラム群は全てのテストケースが成功するプログラム群の部分集合であるため，第一適合度が第二適合度よりも低くなることは無い．

例えば，ステップ 1 でミュータントが 10 個生成され，ステップ 2 で 7 個のミュータントの修正に成功した場合，第一適合度は 0.7（= 7/10）となる．また，修正に成功した 7 個のミュータントのうち，3 個がオーバーフィットなしで修正できた場合，第二適合度は 0.3（= 3/10）となる．

```
01  int example(int value) {
02    if (0 < value) {
03      value--;
04    }
05    else if (value < 0) {
06      value++;
07    }
08    return value;
09  }
```

(a) リファクタリング前

```
01  int example(int value) {
02    boolean c1 = 0 < value;
03    boolean c2 = value < 0;
04    if (c1) {
05      value--;
06    }
07    else if (c2) {
08      value++;
09    }
10    return value;
11  }
```

(b) リファクタリング後

図 2　説明用変数の導入

```
01  int example(int x, int y) {
02    int result;
03    if (y < x) {
04      result = x - y;
05      result = result / 2;
06      return result;
07    } else {
08      result = y - x;
09      result = result / 2;
10      return result;
11    }
12  }
```

(a) リファクタリング前

```
01  int example(int x, int y) {
02    int tmp;
03    int result;
04    if (y < x) {
05      tmp = x - y;
06      result = tmp / 2;
07      return result;
08    } else {
09      tmp = y - x;
10      result = tmp / 2;
11      return result;
12    }
13  }
```

(b) リファクタリング後

図 3　一時変数の分離

5.　実験

本実験の目的は，プログラムの構造の違いにより自動修正適合性がどのように変化するのかを確認する点，およびAPR ツールの違いが自動修正適合性に与える影響を確認する点にある．本実験では，複数の APR ツールを用いて，同一の機能を持つが構造が異なるプログラムの自動修正適合性を比較する．

5.1　実験対象

本実験では，単一のメソッドで構成される小規模のプログラムの自動修正適合性を計測する．プログラムの構造の違いとしてリファクタリングを題材とする．リファクタリングの種類は数多くあるが，今回は「メソッドの構成」と「条件記述の単純化」に分類されるリファクタリングのうち，単一のメソッド内で完結する以下の 6 つのリファクタリングを対象とした [16]．対象のプログラムは，6 つのリファクタリングおよびミューテーション演算子を適用しやすいように著者が作成した（図 2〜7）．また，各プログラムに対するテストスイートは分岐網羅が 100%となるように作成した．

説明用変数の導入　式の結果または部分的な結果を一時変数に代入する（図 2）．

一時変数の分離　複数回代入される一時変数を代入ごとに別の一時変数に分離する（図 3）．

重複した条件記述の断片の統合　条件式の全ての分岐先に同じ処理がある場合，その処理を条件式の外側に移動する（図 4）．

条件記述の統合　同じ処理を持つ一連の条件式がある場合，

```
01  int example(int x, int y) {
02    int tmp;
03    int result;
04    if (y < x) {
05      tmp = x - y;
06      result = tmp / 2;
07      return result;
08    } else {
09      tmp = y - x;
10      result = tmp / 2;
11      return result;
12    }
13  }
```

(a) リファクタリング前

```
01  int example(int x, int y) {
02    int tmp;
03    int result;
04    if (y < x) {
05      tmp = x - y;
06    } else {
07      tmp = y - x;
08    }
09    result = tmp / 2;
10    return result;
11  }
```

(b) リファクタリング後

図 4　重複した条件記述の断片の統合

```
01  boolean example(int x, int y) {
02    boolean result = false;
03    if (x < 0) {
04      result = true;
05    }
06    if (y < 0) {
07      result = true;
08    }
09    return result;
10  }
```

(a) リファクタリング前

```
01  boolean example(int x, int y) {
02    boolean result = false;
03    if (x < 0 || y < 0) {
04      result = true;
05    }
06    return result;
07  }
08  }
```

(b) リファクタリング後

図 5　条件記述の統合

```
01  int example(int[] array) {
02    int result = 0;
03    boolean isFound = false;
04    for (int i : array) {
05      if (!isFound) {
06        if (0 < i) {
07          isFound = true;
08        } else {
09          result++;
10        }
11      }
12    }
13    return result;
14  }
```

(a) リファクタリング前

```
01  int example(int[] array) {
02    int result = 0;
03    for (int i : array) {
04      if (0 < i) {
05        break;
06      } else {
07        result++;
08      }
09    }
10    return result;
11  }
```

(b) リファクタリング後

図 6　制御フラグの削除

```
01  int example(int value) {
02    if (0 < value) {
03      value--;
04    }
05    else if (value < 0) {
06      value++;
07    }
08    return value;
09  }
```

(a) リファクタリング前

```
01  int example(int value) {
02    if (0 < value) {
03      value--;
04      return value;
05    }
06    if (value < 0) {
07      value++;
08      return value;
09    }
10    return value;
11  }
```

(b) リファクタリング後

図 7　ガード節による入れ子条件記述の置き換え

それらを 1 つの条件記述にまとめる（図 5）．

制御フラグの削除　処理の流れを制御するフラグを削除し，`break`，`continue`，`return` を利用する（図 6）．

ガード節による入れ子条件記述の置き換え　後続の処理の対象外となる条件が満たされた場合に `return` する処理を先頭に記述する（図 7）．

単一のメソッド内で完結するリファクタリングとして「一時変数のインライン化」も存在する．一時変数のインライン化は，説明変数の導入と逆の変換を行うリファクタリングであるため説明変数の導入と合わせて評価する．

5.2　APR ツール

本実験では自動修正適合性の計測に以下の条件を満たす

APR ツールを利用した．

公開されている　本実験では APR ツールを実行する必要があるため，公開されていないツール (e.g. ELIXIR [20]) は対象外とした．

任意のプログラムに対して実行可能　本実験では著者が作成したプログラムを入力として APR ツールを実行するため，特定のベンチマークに対してのみ実行可能なツール (e.g. SimFix [9] は Defects4J [10] のプログラムに対してのみ実行可能である) は対象外とした．

ソースコードとテストケースのみが必要　本実験ではソースコードとテストケースの情報のみを利用する APR 手法を対象とするため，その他の入力を必要とするツール (e.g. HDRepair [12] は，開発履歴のデータを必要とする) は対象外とした．

以上の条件に基づいて APR ツールを調査したところ，以下の 7 つのツールが対象となった．対象のツールは全て生成と検証に基づく手法の APR ツールである．

GenProg-A [24]， jGenProg [17]， kGenProg [8]
GenProg [13] の Java 実装版である．

Arja [24]　多目的遺伝的アルゴリズムを用いたツールである．変異プログラムを選択する際，GenProg はテストの通過率で変異プログラムを評価するが，Arja は通過したテストケースの種類や改変した行数も考慮して変異プログラムを評価する．

RSRepair-A [24]　GenProg と同様に挿入，削除，置換によりプログラムを改変するが，遺伝的アルゴリズムではなくランダムサーチにより解を探索する．遺伝的アルゴリズムは「変異プログラムの生成と変異プログラムの選択」を繰り返してプログラムを修正する．一方，ランダムサーチは変異プログラムの選択を行わず，元のプログラムから変異プログラムを生成する処理のみでプログラムを修正する．

jMutRepair [17]　ミューテーションに基づく修正手法 [3] の Java 実装版である．条件式および `return` 文のミューテーションによりプログラムを修正する．

Cardumen [18]　修正対象のプログラムからマイニングしたテンプレートを用いて，式を置換することでプログラムを修正する．

表 1　PIT の OLD_DEFAULT のミューテーション演算子

ミューテーション演算子	変換例	
	変換前	変換後
Conditional Boundary	`a<b`	`a<=b`
Increments	`n++`	`n--`
Invert Negatives	`-n`	`n`
Math	`a+b`	`a-b`
Negate Conditionals	`a==b`	`a!=b`
Void Method Calls	`method();`	`;`
Primitive Returns	`return 5;`	`return 0;`
Empty Returns	`return "str";`	`return "";`
False Returns	`return true;`	`return false;`
True Returns	`return false;`	`return true;`
Null Returns	`return object;`	`return null;`

5.3　実験設定

本実験では，ミュータントを生成するために PIT [2] のミューテーション演算子を利用した．PIT はミュータントの生成に広く使われている [22]．しかし，PIT を含め多くのミューテーションテスティングツールはコンパイルに要する時間を削減するためにソースコードではなくバイトコードを書き換える．そのため，PIT そのものを利用するのではなく，PIT のミューテーション演算子を参考にしてソースコードを書き換えるツールを著者が実装した．実装したツールでは，表 1 に示す PIT の OLD_DEFAULT に分類されるミューテーション演算子が実装されている．

各 APR ツールの設定を表 2 に示す．kGenProg，jGenProg，Arja，GenProg-A，RSRepair-A は，1 世代で生成する変異プログラムの数を 10 とし，最大世代数は 100 とした．jMutRepair, Cardumen は生成する変異プログラムの数を制御するオプションが存在しないため，デフォルトの設定で実行した．jMutRepair 以外のツールはプログラムの修正に乱択に基づいた操作を含む．ゆえに，jMutRepair 以外のツールでは，乱数のシード値を変えて 100 回自動修正適合性を計測し，その平均値を利用した．

第二適合度計測時のオーバーフィットの判定は，改行，空白を除いて文字列的に元のプログラムと同じプログラムに復元できたかどうかを基準とした．APR におけるオーバーフィットの自動的な判定は困難な課題であり，多くの既存研究では手動で判定している．しかし，本実験では多くの修正済みプログラムが生成されるため全てを手動で確認できない．そのため，自動的にオーバーフィットを判定できるよう厳しい基準であるが元のプログラムに復元できたかどうかをオーバーフィットの基準とした．

5.4　第一適合度の計測結果と考察

第一適合度の計測結果を図 8 に示す．図 8 の各セルにおいて上の値はリファクタリング前の第一適合度，下の値はリファクタリング後の第一適合度を表す．緑色のセルはリ

表 2　APR ツールの設定

ツール名	設定
kGenProg	–mutation-generating-count 5 –crossover-generating-count 5 –max-generation 100
jGenProg	-population 10 -maxgen 100
jMutRepair	デフォルト
Cardumen	デフォルト
Arja	-DpopulationSize 10 -DmaxGenerations 100
GenProg-A	-DpopulationSize 10 -DmaxGenerations 100
RSRepair-A	-DpopulationSize 10 -DmaxGenerations 100

リファクタリング	kGenProg	jGenProg	jMutRepair	Cardumen	Arja	GenProg-A	RSRepair-A
説明用変数の導入	1.000 0.747	0.897 0.717	0.667 0.333	1.000 0.635	0.387 0.220	0.000 0.000	0.058 0.015
一時変数の分離	0.833 0.833	0.575 0.602	0.167 0.167	0.123 0.258	0.487 0.500	0.067 0.103	0.340 0.347
重複した条件記述の断片の統合	0.833 0.750	0.602 0.556	0.167 0.167	0.258 0.202	0.500 0.068	0.103 0.002	0.346 0.006
条件記述の統合	0.918 0.875	0.643 0.703	1.000 1.000	0.188 0.320	0.123 0.000	0.000 0.000	0.005 0.000
制御フラグの削除	0.983 0.993	0.337 0.357	0.667 0.667	0.107 0.333	0.060 0.167	0.000 0.003	0.003 0.006
ガード節による入れ子条件記述の置き換え	1.000 1.000	0.897 0.950	0.667 0.667	1.000 1.000	0.387 0.647	0.000 0.203	0.058 0.415

リファクタリング適用後第一適合度が 向上 低下

図 8 第一適合度の計測結果

```
01  int example(int value) {
02    if (0 <= value) { // 0 < value
03      value--;
04    } else if (value > 0) {
05      value++;
06    }
07    return value;
08  }
```

(a) リファクタリング前

```
01  int example(int value) {
02    boolean c1 = 0 <= value; // 0 < value
03    boolean c2 = value > 0;
04    if (c1) {
05      value--;
06    } else if (c2) {
07      value++;
08    }
09    return value;
10  }
```

(b) リファクタリング後

図 9 説明用変数の導入の適用により jMutRepair で修正できなくなった欠陥

ファクタリング後第一適合度が向上したことを表し，赤色のセルはリファクタリング後第一適合度が低下したことを表す．図 8 より，リファクタリングの前後で第一適合度が異なるケースが多いことから機能が同じプログラムでも構造によって第一適合度が変化することが分かる．説明用変数の導入や重複した条件記述の断片の統合のように APR ツールによらず第一適合度が低下するリファクタリングもあれば，一時変数の分離や制御フラグの削除のように APR ツールによらず第一適合度が向上するリファクタリングも存在した．また，条件記述の統合のように APR ツールによって第一適合度が向上したり低下したりするリファクタリングも存在した．説明用変数の導入は，多くの APR ツールで第一適合度が大きく低下したことから，第一適合度への影響が大きいといえる．これは，説明用変数の導入の対となる一時変数をインライン化により第一適合度が大きく向上したともいえる．

jMutRepair に着目すると，説明用変数の導入以外のリファクタリングではリファクタリングの適用前後で第一適合度に変化が見られなかった．これは，jMutRepair が条件式あるいは return 文のミューテーションにより修正するという戦略が要因である．説明用変数の導入により修正できなくなった欠陥の例を図 9 に示す．図 9(a)，図 9(b) のいずれも 2 行目に欠陥を含み，`0 < value` が `0 <= value` に書き換えられている．図 9(a) は jMutRepair のミューテーションの対象である `if` 文の条件式に欠陥があるため修正可能である．一方，図 9(b) は，代入文に欠陥が含まれ，代入文は jMutRepair のミューテーションの対象ではないため修正できない．このように説明用変数の導入では，条件式や `return` 文に存在していた欠陥が代入文に移ることで修正できない欠陥が増加する．反対に，一時変数のインライン化により条件式や `return` 文に欠陥が移ると jMutRepair で修正できる欠陥が増加する．

Arja，GenProg-A，RSRepair-A は第一適合度が向上するリファクタリングと低下するリファクタリングがほとんど同じであった．これらの APR ツールはいずれも同じフレームワークを利用して実装されており，修正の戦略も類似している．このことから，実装が類似した APR ツールは修正しやすいプログラムの構造も類似する可能性が考えられる．また，Arja，RSRepair-A はリファクタリングによる第一適合度の変化が大きいケースがいくつか見られた．条件記述の統合を適用した場合に第一適合度は大きく低下し，ガード節による入れ子条件記述の置き換えを適用した場合には第一適合度が大きく向上した．

以上より，第一適合度を向上させる方法として以下の手段が有効だと考えられる．

- 一時変数をインライン化する．
- Arja，RSRepair-A を利用する場合はガード節による入れ子条件記述の置き換えを行う．

5.5 第二適合度の計測結果と考察

第二適合度の計測結果を図 10 に示す．図 10 の各セルにおいて上の値はリファクタリング前の第二適合度，下の値

リファクタリング	kGenProg	jGenProg	jMutRepair	Cardumen	Arja	GenProg-A	RSRepair-A
説明用変数の導入	0.508 0.543	0.211 0.692	0.333 0.000	1.000 0.635	0.000 0.000	0.000 0.000	0.000 0.000
一時変数の分離	0.428 0.328	0.561 0.591	0.000 0.000	0.117 0.200	0.333 0.333	0.063 0.062	0.333 0.333
重複した条件記述の断片の統合	0.328 0.352	0.592 0.540	0.000 0.000	0.200 0.120	0.333 0.000	0.062 0.000	0.333 0.000
条件記述の統合	0.620 0.685	0.590 0.000	0.000 0.000	0.188 0.000	0.000 0.000	0.000 0.000	0.000 0.000
制御フラグの削除	0.797 0.930	0.330 0.343	0.667 0.667	0.000 0.333	0.000 0.000	0.000 0.000	0.000 0.000
ガード節による入れ子条件記述の置き換え	0.508 0.281	0.211 0.318	0.300 0.300	1.000 1.000	0.000 0.000	0.000 0.000	0.000 0.000

リファクタリング適用後第二適合度が 向上 低下

図 10 第二適合度の計測結果

はリファクタリング後の第二適合度を表す．緑色のセルはリファクタリング後第二適合度が向上したことを表し，赤色のセルはリファクタリング後第二適合度が低下したことを表す．

第二適合度の計測では，オーバーフィットの基準の厳しさによりオーバーフィットのないプログラムを生成できないケースも多く存在したが，リファクタリングの前後で第二適合度が異なるケースが多いことから，第一適合度と同様に機能が同じプログラムでも構造によって第二適合度が変化することが分かる．制御フラグの削除は第一適合度と同様，APR ツールによらず第二適合度が向上した．

第一適合度と同様に，Arja，GenProg-A，RSRepair-A は第二適合度が低下するリファクタリングがほとんど同じであった．特に Arja，RSRepair-A の第二適合度は類似しており，いずれも重複した条件記述の断片の統合を適用した場合に第二適合度が大きく低下した．また，jGenProg では説明用変数の導入を適用した場合に第二適合度が大きく向上した．

以上より，第二適合度を向上させる方法として以下の手段が有効だと考えられる．

- Arja，RSRepair-A を利用する場合は重複した条件記述の断片の統合を適用しない．
- jGenProg を利用する場合は説明変数を導入する．

また，各 APR ツールについて，図 8 の第一適合度と図 10 の第二適合度を比較すると jGenProg，Cardumen は第一適合度と第二適合度の差が他のツールに比べ小さい．すなわち，jGenProg， Cardumen は全てのテストケースが成功するプログラムを生成できた場合に，そのプログラムがオーバーフィットを含む可能性が低いツールであると言える．反対に，kGenProg は第一適合度と第二適合度の差が大きいため，オーバーフィットのあるプログラムを生成しやすいツールであるといえる．

6. 妥当性の脅威

この実験では単一のメソッド内で完結する 6 種類のリファクタリングを実験対象としたが，リファクタリングの種類は他にも数多く存在する．他のリファクタリングを対象に実験することで，新たな傾向や今回の傾向を否定する例が現れる可能性がある．

また，この実験ではプログラムの構造と APR ツールの種類を変化させて自動修正適合性を計測した．本研究における自動修正適合性の計測手法は，ミュータント生成器やテストスイートの品質にも依存する手法であるが，それらがどのように影響するかは評価できていない．テストスイートの影響を評価する場合には，実験 1 で採用した分岐網羅以外のテストカバレッジを満たすようにテストを作成する方法やテストケース自動生成ツール [5] で生成したテストを利用する方法が考えられる．ミュータント生成器の影響を評価する場合は，表 1 以外のミューテーション演算子を利用する方法やミューテーション演算子による書き換え箇所を増やす方法が考えられる．

7. あとがき

本研究では自動修正適合性という新しいソフトウェアの品質指標を提案した．自動修正適合性は，自動プログラム修正技術が対象のプログラムに対してどの程度効果的に作用するかを表す．また，自動修正適合性の計測手法も提案した．提案手法は，全てのテストケースに通過するプログラムからミューテーションテスト技術を利用して人工的な欠陥を含むプログラムを複数生成し，それらの欠陥を自動

プログラム修正技術でどの程度修正できたかを計測する．自動プログラム修正技術で修正しやすいプログラムの構造の調査を目的として，小さいプログラムを対象に，プログラムの構造の違いによる自動修正適合性の違いを分析した．その結果，以下の 2 点が明らかになった．

- プログラムの構造によって自動修正適合性が変化する．
- 一時変数をインライン化すると自動修正適合性が向上する傾向がある．

参考文献

[1] Britton, T., Jeng, L., Carver, G. and Cheak, P.: Reversible Debugging Software "Quantify the Time and Cost Saved Using Reversible Debuggers" (2013).

[2] Coles, H., Laurent, T., Henard, C., Papadakis, M. and Ventresque, A.: PIT: A Practical Mutation Testing Tool for Java, *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452 (2016).

[3] Debroy, V. and Wong, W. E.: Using mutation to automatically suggest fixes for faulty programs, *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pp. 65–74 (2010).

[4] Durieux, T., Madeiral, F., Martinez, M. and Abreu, R.: Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts, *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313 (2019).

[5] Fraser, G. and Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software, *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419 (2011).

[6] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67 (2019).

[7] Hailpern, B. and Santhanam, P.: Software debugging, testing, and verification, *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12 (2002).

[8] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-Performance, High-Extensibility and High-Portability APR System, pp. 697–698 (2018).

[9] Jiang, J., Xiong, Y., Zhang, H., Gao, Q. and Chen, X.: Shaping program repair space with existing patches and similar code, *Proceedings of the 27th International Symposium on Software Testing and Analysis*, pp. 298–309 (2018).

[10] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[11] Le, X. B. D., Chu, D. H., Lo, D., Le Goues, C. and Visser, W.: S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples, *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pp. 593–604 (2017).

[12] Le, X. B. D., Lo, D. and Le Goues, C.: History driven program repair, *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1, pp. 213–224 (2016).

[13] Le Goues, C., Nguyen, T., Forrest, S. and Weimer, W.: Genprog: A generic method for automatic software repair, *IEEE Transactions on Software Engineering*.

[14] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y. L.: On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs, *Proceedings of the 42nd International Conference on Software Engineering*, p. 615–627 (2020).

[15] Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A. and Scott, A.: SapFix: Automated End-to-End Repair at Scale, *Proceedings of the 41st International Conference on Software Engineering*, pp. 269–278 (2019).

[16] Martin, F.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).

[17] Martinez, M. and Monperrus, M.: Astor: A program repair library for java, *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441–444 (2016).

[18] Martinez, M. and Monperrus, M.: Ultra-large repair search space with automatically mined templates: The cardumen mode of astor, *Proceedings of the 10th International Symposium on Search Based Software Engineering*, pp. 65–86 (2018).

[19] Naitou, K., Tanikado, A., Matsumoto, S., Higo, Y., Kusumoto, S., Kirinuki, H., Kurabayashi, T. and Tanno, H.: Toward Introducing Automated Program Repair Techniques to Industrial Software Development, *Proceedings of the 26th International Conference on Program Comprehension*, pp. 332–335 (2018).

[20] Saha, R. K., Lyu, Y., Yoshida, H. and Prasad, M. R.: Elixir: Effective object-oriented program repair, *Proceedings of the 32nd International Conference on Automated Software Engineering*, pp. 648–659 (2017).

[21] Wen, M., Chen, J., Wu, R., Hao, D. and Cheung, S.-C.: Context-aware patch generation for better automated program repair, *Proceedings of the 40th International Conference on Software Engineering*, pp. 1–11 (2018).

[22] Xuan, J. and Monperrus, M.: Test case purification for improving fault localization, *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pp. 52–63 (2014).

[23] Yi, J., Tan, S. H., Mechtaev, S., Böhme, M. and Roychoudhury, A.: A Correlation Study between Automated Program Repair and Test-Suite Metrics, *Proceedings of the 40th International Conference on Software Engineering*, p. 24 (2018).

[24] Yuan, Y. and Banzhaf, W.: ARJA: Automated repair of java programs via multi-objective genetic programming, *IEEE Transactions on Software Engineering* (2018).