修士学位論文

題目

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手法の提案

指導教員

楠本 真二 教授

報告者

九間 哲士

令和 3 年 2 月 2 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 2 年度 修士学位論文

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手法の提案

九間 哲士

内容梗概

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，オープンソースソフトウェアに含まれる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，修正の成功率は高くない．そこで，著者は自動プログラム修正技術を開発するだけでなく，プログラムも自動プログラム修正技術で修正しやすいように開発するべきだと考えた．本研究では，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．自動修正適合性を利用することで，ソフトウェア開発に自動プログラム修正を導入するかを事前に判断できる．また，自動プログラム修正技術で欠陥を修正しやすくするために，自動プログラム修正技術の適用前に修正対象のプログラムをリファクタリングする研究も可能になる．自動プログラム修正技術で欠陥を修正しやすいプログラムの構造が明らかになれば，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．本研究ではプログラムの自動修正適合性を自動的に計測する手法も提案する．提案手法は，全てのテストケースが成功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する．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，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，ネストの深さあるいは循環的複雑度を小さくするリファクタリングが自動修正適合性を向上させる傾向があることを確認した．また，実際の OSS で発生した 106 個の欠陥対象に，プログラムの構造によって欠陥の修正しやすさが変化するか確認した結果，構造の変換により新たに 9 個の欠陥を修正できるようになった．

主な用語

自動プログラム修正，ミューテーションテスト，ソフトウェア品質モデル

目次

図目次

表目次

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半数以上をデバッグ作業が占めるという報告もある [1, 2]．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 [3]．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いる [4, 5, 6, 7]．オープンソースソフトウェアに含まれる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，修正の成功率は高くない．Liu らの研究では，Defects4J [8] 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 [9]．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 [10]，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 [11]．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の効果を持つかを示す指標があれば，開発者は自動プログラム修正技術を利用するかどうかを事前に判断できる．そのような指標があれば，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，欠陥の修正に寄与しないという事態を避けられる．

図 1 は ISO/IEC 25010 で定義されているソフトウェアの品質モデルである．ISO/IEC 25010 には 8 つの品質特性が含まれる．Maintainability (保守性) は欠陥の修正に関する品質特性であり，保守者がソフトウェアの欠陥をどの程度効率的に修正できるかを表す．すなわち，保守性は保守者のための特性であり，自動プログラム修正技術は考慮されていない．開発者が欠陥を取り除きやすく自動プログラム修正技術では欠陥を取り除きにくいという状況を考慮すると，保守性は自動プログラム修正がどの程度効果的かを評価するには不十分である．

ゆえに，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．本研究では，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．

- 自動プログラム修正技術をソフトウェア開発に導入するかの判断基準
- 自動プログラム修正技術で欠陥を修正しやすいようにプログラムを変換するための指標

また，本研究では自動修正適合性の計測手法も提案する．提案手法は，全てのテストケースが成功する正常なプログラムから，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

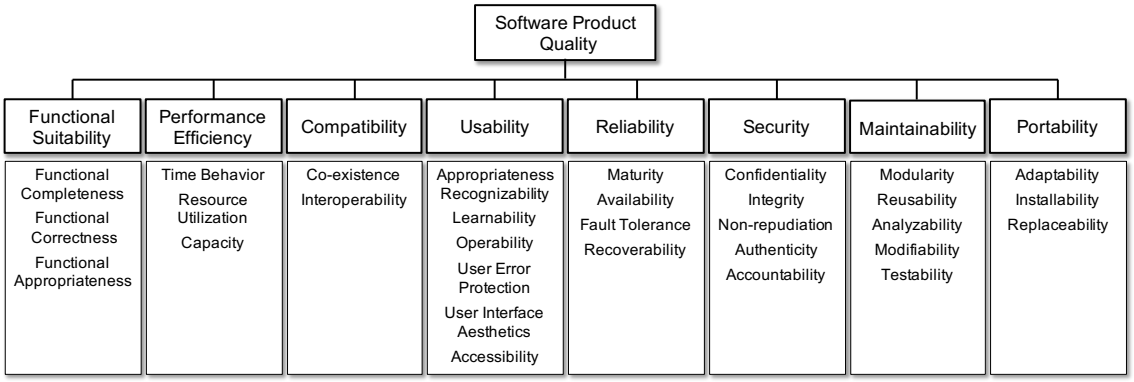

図 1　ISO/IEC 25010 のソフトウェア品質モデル

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，自動修正適合性を向上させるプログラムの変換例を発見した．また，実際の OSS を対象に，プログラムの構造によって欠陥の修正しやすさが変化するか確認した結果，構造の変換により 106 個の欠陥のうち新たに 9 個の欠陥を修正できるようになった．

本研究の貢献は以下の点である．

- 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.
- ミューテーションテスト技術を応用した自動修正適合性の計測手法を提案.
- プログラムの構造により自動修正適合性が異なる事例を発見.
- 自動修正適合性を向上させるプログラムの変換例を発見.
- プログラムの構造により OSS の欠陥の修正可否が異なる事例を発見.

以降，2 章では準備について述べる．3 章では提案する自動修正適合性について述べる．4 章では自動修正適合性の計測手法について述べる．5 章では 1 つ目の実験の実験方法や結果について述べる．6 章では 2 つ目の実験の実験方法や結果ついて述べる．7 章では妥当性の脅威について述べる．8 章では本研究のまとめについて述べる．

2 準備

2.1 自動プログラム修正技術

2.1.1 概要

自動プログラム修正とは，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．自動プログラム修正は，大きく分けて欠陥限局とプログラム改変の 2 つの要素で構成される．欠陥限局では，テストケースの実行時の情報を基にプログラム中の欠陥を含む箇所を推測する．プログラム改変では，欠陥限局によって欠陥を含む箇所であると推測された箇所を書き換える．

2.1.2 欠陥限局

欠陥限局とは，プログラムの欠陥箇所を推測する技術である．自動プログラム修正で用いられる欠陥限局の手法として，実行経路情報に基づく欠陥限局手法 (Spectrum-Based Fault Localization，以降，SBFL) がある．SBFL は失敗テストケースで実行された文ほど欠陥を含む可能性が高いという考えに基づいて欠陥限局を行う．SBFL は，各テストケースの成否と実行経路情報を用いて，各文に対する疑惑値，すなわち欠陥を含む可能性を表す値を出力する．実行経路情報とは，各テストケースの実行時にプログラム中のどの文が実行されたかを表す情報である．疑惑値の計算手法はこれまでに数多く提案されている．Abreu らは，7 つの疑惑値の計算手法を比較し，Ochiai [12] が優れた手法であると結論付けている [13]．Ochiai の計算式は以下の式で表される．

$$susp(s) = \frac{fail(s)}{\sqrt{totalfail \times (fail(s) + pass(s))}}$$

式中の各変数は以下の意味を表す．

$susp(s)$　文 s の疑惑値
$totalfail$　失敗テストケースの総数
$fail(s)$　文 s を実行した失敗テストケースの数
$pass(s)$　文 s を実行した成功テストケースの数

2.1.3 自動プログラム修正

自動プログラム修正の手法は，生成と検証に基づく手法 [14] と，プログラムの意味論に基づく手法 [15] 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．生成と検証に基づく手法では，全てのテス

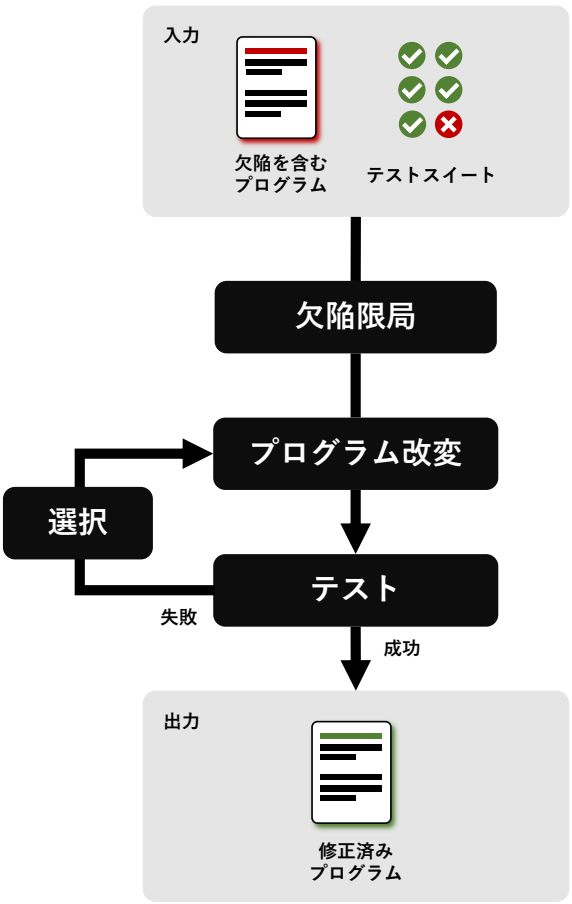

図 2　GenProg の処理の流れ

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．

生成と検証に基づく手法の 1 つに GenProg [16] がある．GenProg は遺伝的アルゴリズムを利用した手法である．GenProg の処理の流れを図 2 に示す．GenProg は欠陥限局後，以下の 3 つの処理を繰り返し，プログラムを修正する．

- プログラムの改変
- 変異プログラムのテスト
- 変異プログラムの選択

プログラムの改変では，GenProg は以下の 3 つの操作により修正対象のプログラムを改変し，複数の変異プログラムを生成する．

```
01  int abs(int n) {
02    if(n >=0)
03      return n;
04    else
05      return n;
06  }
```

(a) 欠陥を含むプログラム

テストケース	入力n	期待値	出力値
t1	10	10	10
t2	-10	10	-10

(b) テストスイート

```
01  int abs(int n) {
02    if(n >=0)
03      return n;
04    else
05      return -n;
06  }
```

(c) オーバーフィットのない修正

```
01  int abs(int n) {
02    if(n >=0)
03      return n;
04    else
05      return 10;
06  }
```

(d) オーバーフィットのある修正

図 3　オーバーフィットの例

挿入　欠陥があると推測された文の前あるいは後ろに文を挿入する．

削除　欠陥があると推測された文を削除する．

置換　削除と挿入を同時に行う．

挿入や置換の操作で利用する文は修正対象のプログラムからランダムに選択される．生成された変異プログラムに対してテストを実行し，全てのテストケースが成功するプログラムが存在する場合，そのプログラムを修正済みプログラムとして出力する．全てのテストケースが成功するプログラムが存在しない場合，テストの成功率が高い変異プログラムを選択し，その変異プログラムを再び改変する．この一連の処理を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繰り返す．

2.1.4　オーバーフィット

自動プログラム修正の課題の 1 つとしてオーバーフィットがある．自動プログラム修正におけるオーバーフィットとは，生成された修正済みプログラムが入力として与えられたテストスイートに過剰適合することを意味する．すなわち，入力として与えられた全てのテストケースは成功するが，入力値が異なるテストケースは失敗するプログラムを生成してしまうという問題である．

オーバーフィットのある修正の例を図 3 に示す．図 3(a) のプログラムは入力値の絶対値を返すプログラムである．このプログラムは 5 行目に欠陥があり，図 3(b) のテストケース t_2 が失敗する．図 3(a) のプログラムに対するオーバーフィットのない修正は図 3(c) のような修正である．図 3(c) では `return n;` を `return -n;` に置換している．図 3(c) のプログラムは図 3(b) に含まれない入力値を与えても正しく動作する．一方，図 3(d) はオーバーフィットのある修正である．図 3(d) では `return`

`n;` を `return 10;` に置換している．図 3(d) のプログラムは図 3(b) の全てのテストケースが成功するが，$n = -1$ のような入力を持つテストケースを実行するとテストに失敗する．

自動プログラム修正技術によって出力されるプログラムにはオーバーフィットのあるプログラムも多い [9]．そのため，オーバーフィットを防ぐための研究も行われている [17, 18]．

2.2 ソフトウェアテストの品質

2.2.1 概要

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が高いほど，対象のソフトウェアに欠陥が混入する可能性は低くなる．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．

2.2.2 テストカバレッジ

テストカバレッジ [19] はソフトウェアテストの品質を評価する指標の 1 つである．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 [20]．

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

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．

2.2.3 ミューテーションテスト

ミューテーションテスト [21] はソフトウェアテストの品質を評価する手法の 1 つである．ミューテーションテストではテスト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，ミュータント演

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

本研究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，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の効果を持つかを表す．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自動プログラム修正技術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のないプログラムの生成に自動プログラム修正技術がどの程度の効果を持つかを表す．

自動修正適合性は以下のような用途で利用できる．

- ソフトウェア開発に自動プログラム修正技術を導入するかの判断

  自動プログラム修正技術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，数多く存在する自動プログラム修正ツールの中からプロジェクトに合った自動プログラム修正ツールを選定し，そのツールの利用方法を学習するコストがある．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自動プログラム修正に必要なテストには乖離があるため，自動プログラム修正技術を導入するにはテストを拡充する必要もある [11]．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自動プログラム修正技術がどの程度の効果を持つかが分かる．既存のプロジェクトの自動修正適合性が高ければ自動プログラム修正技術を導入するという判断が可能になり，コストをかけて自動プログラム修正技術を導入したが，期待した効果を得られないという事態を防げる．
- 自動プログラム修正技術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自動プログラム修正技術を利用して欠陥を修正できる．
- 自動プログラム修正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自動プログラム修正に影響を与えるかを明らかにできる．自動プログラム修正技術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．

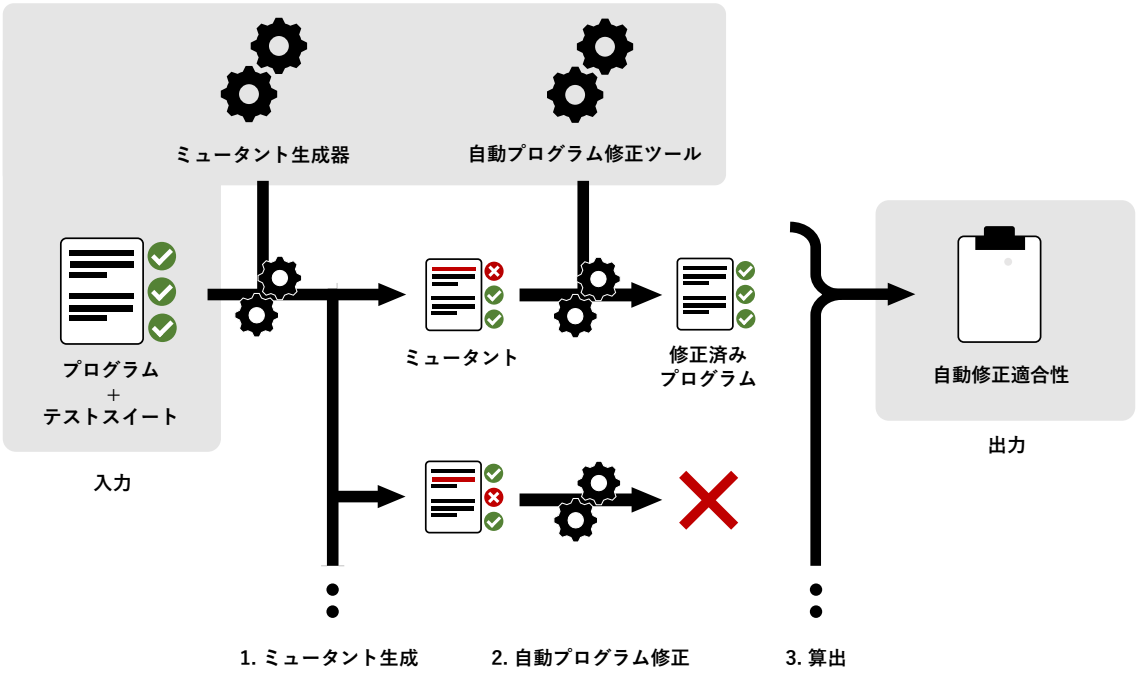

図 4　提案手法の流れ

4　自動修正適合性の計測

本研究では自動修正適合性の提案に加え，自動修正適合性の計測手法も提案する． 提案手法のキーアイディアは，計測対象の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 4 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
- 自動プログラム修正ツール 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自動プログラム修正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 [22]．また，自動プログラム修正ツール

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 [23].

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

ステップ 2　自動プログラム修正技術の適用

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.

4.1　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，同じミュータントは生成されない.

4.2　ステップ 2. 自動プログラム修正技術の適用

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 を入力として自動プログラム修正ツール 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欠陥を含むプログラムとはみなされない．全てのテストケースが成功するミュータントに対しては A は実行せず，そのミュータントをステップ 3 における「生成された全ミュータント」から除外する.

4.3　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．ゆえに，自動修正適合性は以下の式で算出される.

$$\text{自動修正適合性} = \frac{\text{修正済みプログラムを生成できたミュータントの数}}{\text{生成されたミュータントの数}}$$

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 $\subset$ 全てのテストケースが成功するプログラムであるため，第一適合度 $\geq$ 第二適合度が必ず成り立つ.

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，

$$\text{第一適合度} = \frac{7}{10} = 0.70$$

となる.

さらに，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ていた場合，

$$\text{第二適合度} = \frac{3}{10} = 0.30$$

となる．

5 実験 1

5.1 実験概要

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か確認する点，および自動プログラム修正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，複数の自動プログラム修正ツールを用いて，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．

5.2 実験対象

本実験では，単一のメソッドで構成される小さなプログラムの自動修正適合性を計測する．対象のプログラムは，後述する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．各プログラムに対するテストスイートは分岐網羅が 100 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以下の 6 つのリファクタリングを対象とした [24]．本実験では，単一のメソッドで構成される小さなプログラムの自動修正適合性を計測するため，単一のメソッド内で完結するリファクタリングを対象とした．

- 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．説明用変数の導入を適用する前後のプログラムを図 5 に示す．
- 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．一時変数の分離を適用する前後のプログラムを図 6 に示す．
- 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．重複した条件記述の断片の統合を適用する前後のプログラムを図 7 に示す．
- 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，それらを 1 つの条件記述にまとめる．条件記述の統合を適用する前後のプログラムを図 8 に示す．
- 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．制御フラグの削除を適用する前後のプログラムを図 9 に示す．
- 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．ガード節による入れ子条件記述の置き換えを適用する前後のプログラムを図 10 に示す．

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

```
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のプログラム

```
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のプログラム

図 5　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のプログラム

```
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のプログラム

図 6　一時変数の分離

```
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のプログラム

```
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のプログラム

図 7　重複した条件記述の断片の統合

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．

```
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のプログラム

```
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のプログラム

図 8　条件記述の統合

```
01  int removeFlag(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のプログラム

```
01  int removeFlag(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のプログラム

図 9　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のプログラム

```
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のプログラム

図 10　ガード節による入れ子条件記述の置き換え

5.3　自動プログラム修正ツール

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自動プログラム修正ツールを利用した．

- 公開されているツールである: 本実験では，自動プログラム修正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は対象外とした (e.g. ELIXIR [25]).

- 任意のプログラムに対して実行可能である: 本実験では，著者が作成したプログラムを入力として自動プログラム修正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は対象外とした (e.g. SimFix [26] は Defects4J [8] のプログラムに対してのみ実行可能である).
- ソースコードとテストケースのみを入力とする自動プログラム修正ツールである: 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自動プログラム修正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は対象外とした (e.g. HDRepair [27] は，開発履歴のデータを必要とする).

以上の条件に基づいて自動プログラム修正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自動プログラム修正ツールである．

- **GenProg-A** [28]， **jGenProg** [29]， **kGenProg** [30]: GenProg [16] の Java 実装版である．
- **Arja** [28]: 多目的遺伝的アルゴリズムを用いたツールである．多目的遺伝的アルゴリズムとは，複数の評価関数を用いて解を探索する遺伝的アルゴリズムの一種である．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** [28]: 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** [29]: ミューテーションに基づく修正手法 [31] 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** [32]: 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．

5.4 実験設定

本実験では，ミュータントを生成するために PIT [33] 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 [34]．しかし，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 を基にソースコードを書き換えるツールを実装した．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．

各自動プログラム修正ツールの設定を表 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．自動プログラム修正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．

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	説明	変換例	
		変換前	変換後
Conditional Boundary	関係演算子の境界を変更する	`a<b`	`a<=b`
Increments	インクリメントとデクリメントを入れ替える	`n++`	`n--`
Invert Negatives	負の数を正の数に置き換える	`-n`	`n`
Math	算術演算子を置き換える	`a+b`	`a-b`
Negate Conditionals	関係演算子を置き換える	`a==b`	`a!=b`
Void Method Calls	何の値も返さないメソッド呼び出しを削除する	`method();`	`;`
Primitive Returns	プリミティブ型の戻り値を 0 に置き換える	`return 5;`	`return 0;`
Empty Returns	戻り値の型に応じて空を表す値に置き換える	`return "str";`	`return "";`
False Returns	戻り値を false に置き換える	`return true;`	`return false;`
True Returns	戻り値を true に置き換える	`return false;`	`return true;`
Null Returns	戻り値を null に置き換える	`return object;`	`return null;`

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 11　第一適合度の計測結果

5.5　実験結果と考察

5.5.1　第一適合度

第一適合度の計測結果を図 11 に示す．図 11 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 11 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

表 2　自動プログラム修正ツールの設定

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

```
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 12　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

片の統合のように自動プログラム修正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自動プログラム修正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自動プログラム修正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，多くの自動プログラム修正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言える．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言える．

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 12 に示す．図 12(a)，図 12(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 12(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 12(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．これらの自動プログラム修正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，実装が類似した自動プログラム修正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．

より詳細な考察として jGenProg の結果を例に，第一適合度の計測結果をリファクタリングの性質ご

リファクタリング	第一適合度	循環的複雑度	ネストの深さ	宣言コード行数	ステートメント数
説明用変数の導入	低下			増	増
一時変数の分離	向上			増	
重複した条件記述の断片の統合	低下				減
条件記述の統合	向上	減			減
制御フラグの削除	向上	減	減	減	減
ガード節による入れ子条件記述の置き換え	向上		減		増

図 13　GenProg 利用時に第一適合度が向上/低下したリファクタリングの分析

とに分析する．各リファクタリングを適用した時のソースコードメトリクスの変化を図 13 に示す．各リファクタリングを適用時に各ソースコードメトリクスの値が高くなる場合は「増」，値が低くなる場合は「減」が記入されている．

条件記述の統合は循環的複雑度，ガード節による入れ子条件記述の置き換えはネストの深さ，制御フラグの削除はその両方を小さくする性質を持つ． これらのリファクタリングはいずれも適用後に第一適合度が向上している．このことから，循環的複雑度やネストの深さを小さくするプログラムの変換は第一適合度を向上させる可能性があると考えられる．一方，説明用変数の導入と一時変数の分離はいずれも宣言コード行数を大きくする性質を持つ．説明用変数の導入を適用した場合は第一適合度は低下し，一時変数の分離を適用した場合は第一適合度は向上する．このことから，jGenProg を利用する場合に宣言コード行数を大きくするプログラムの変換が第一適合度を向上させるかは現時点では不明である．

同様に他の自動プログラム修正ツールの結果を確認すると，kGenProg，Arja，GenProg-A，RSRepair-A はネストの深さを小さくする変換，Cardumen は循環的複雑度を小さくする変換により第一適合度が向上した．

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．
- 循環的複雑度あるいはネストの深さが小さくなるようにプログラムを変換する．

5.5.2　第二適合度

第二適合度の計測結果を図 14 に示す．図 14 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．

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 14　第二適合度の計測結果

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，自動プログラム修正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．

第二適合度の計測結果も第一適合度と同様にリファクタリングの性質ごとに分析すると，kGenProg は循環的複雑度を小さくする変換，jGenProg はネストの深さを小さくする変換により第二適合度が向上していることが分かる．jGenProg はネストの深さを小さくすると良いという点で第一適合度と第二適合度で共通した結果を得られた．

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．
- kGenProg を利用する場合は循環的複雑度が小さくなるようにプログラムを変換する．
- jGenProg を利用する場合は説明変数を導入する，またはネストの深さが小さくなるようにプログラムを変換する．

また，各自動プログラム修正ツールについて，図 11 の第一適合度と図 14 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言える．

6 実験 2

6.1 実験概要

本実験の目的は，大規模なプロジェクトの欠陥の修正しやすさが，プログラムの構造によって変化するかを確認する点にある．本実験では，OSS の開発中に実際に発生した欠陥をプログラムの構造を変化させて修正する．

6.2 実験対象

自動プログラム修正ツールは kGenProg を利用する．kGenProg を選定した理由は，実験 1 においてどの実験題材でも自動修正適合性が高かったためである．

実験題材は， Defects4J [8] を用いる．Defects4J は， Java の OSS プロジェクトの開発中に発生した欠陥の情報をまとめたデータセットである．欠陥の情報には，プログラムのどこに欠陥が存在し，どのテストケースに失敗したかなどの情報が含まれる．自動プログラム修正技術への入力 (欠陥を含むプログラムとテストスイート) が用意されているため，自動プログラム修正の分野においてベンチマークとして広く利用されている [35, 26] ．本実験では Defects4J に含まれる Apache CommonsMath(以降，Math) を実験題材として用いる．Math には 106 個の欠陥情報が含まれている．Math を実験題材とした理由は，自動プログラム修正ツールのベンチマーク論文で Math の欠陥が最も修正されていたためである [36].

6.3 実験設定

プログラムの構造の違いとして，プログラムの平坦化 [37] を題材とする．平坦化とは，複雑な文や式を複数の簡単な文へと分解する手法である．平坦化の例を図 15 に示す．図 15(a) は 1 つの `return` 文のみを含むプログラムである．この `return` 文は，メソッド呼び出しや除算など複数の処理が中で行われており，複雑な文になっている．図 15(a) のプログラムを平坦化すると図 15(b) のようになる．図 15(b) では，図 15(a) の複雑な `return` 文が複数の簡単な文に分解されている．平坦化を題材とした理由は，平坦化を適用可能な箇所が Math の中に多く存在したためである．適用可能な箇所が少ないプログラムの変換を題材とした場合，欠陥と関係のある箇所の構造を変化させられない可能性がある．

kGenProg の設定を表 3 に示す．1 世代に生成する変異プログラムの数は 10 とし，最大世代数は無制限とする．1 つの欠陥を修正に対する制限時間は 30 分とし，30 分を超えた場合は修正失敗とみなす．kGenProg は乱択の操作を含むため，1 つの欠陥につき乱数のシード値を変えて 5 回修正を試みる．

```
01  public double pow(final double exponent) {
02    return FastMath.pow(numerator.doubleValue(),   exponent) /
03           FastMath.pow(denominator.doubleValue(), exponent);
04  }
```

(a) 複雑な文を持つプログラム

```
01  public double pow(final double exponent) {
02    double $161 = numerator.doubleValue();
03    double $145 = FastMath.pow($161, exponent);
04    double $162 = denominator.doubleValue();
05    double $146 = FastMath.pow($162, exponent);
06    double $79 = $145 / $146;
07    return $79;
08  }
```

(b) 平坦化後のプログラム

図 15　プログラムの平坦化の例

6.4　実験結果

全 106 個の欠陥に対する，修正に成功/失敗した欠陥数を表 4 に示す．ここにおける修正に成功した欠陥とは，5 回の試行のうち 1 回でも修正に成功した欠陥を表す．表 4 より，平坦化前に修正できた欠陥は 44 個，平坦化後に修正できた欠陥は 45 個であり，大きな差は見られなかった．

平坦化の前後で共通して修正できた欠陥に対する修正成功回数の比較を図 16 に示す．図 16 より平坦化前の方が修正回数が多かった欠陥は 8 個，平坦化後の方が修正成功回数が多かった欠陥は 16 個であった．平坦化後の方が修正成功回数が多い傾向が僅かに見られた．

平坦化前後の修正に成功した欠陥の集合の関係を図 17 に示す．図 17 より平坦化の前後で共通して

表 3　kGenProg の設定

設定項目	設定値
1 世代に生成する変異プログラム	10 個
最大世代数	無制限
制限時間	1 試行当たり 30 分
乱数シード	0〜4 (=5 試行)

表 4　修正に成功/失敗した欠陥の数

	修正成功	修正失敗
平坦化前	44 個	62 個
平坦化後	45 個	61 個

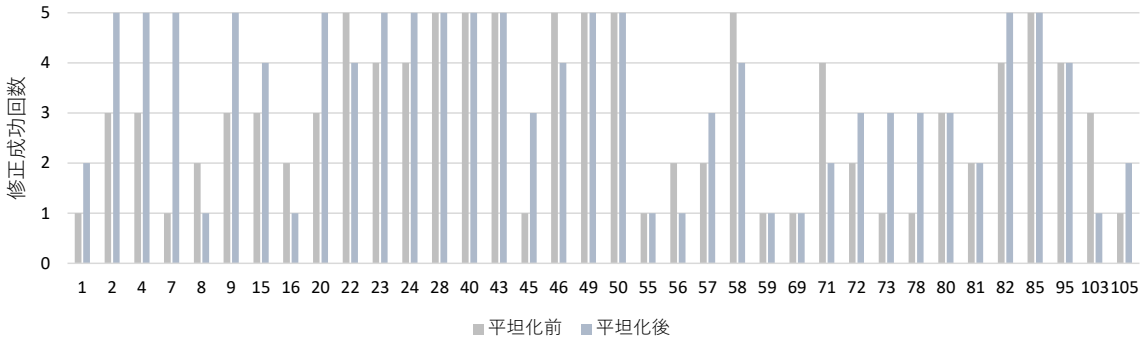

図 16　修正に成功した欠陥の修正成功回数

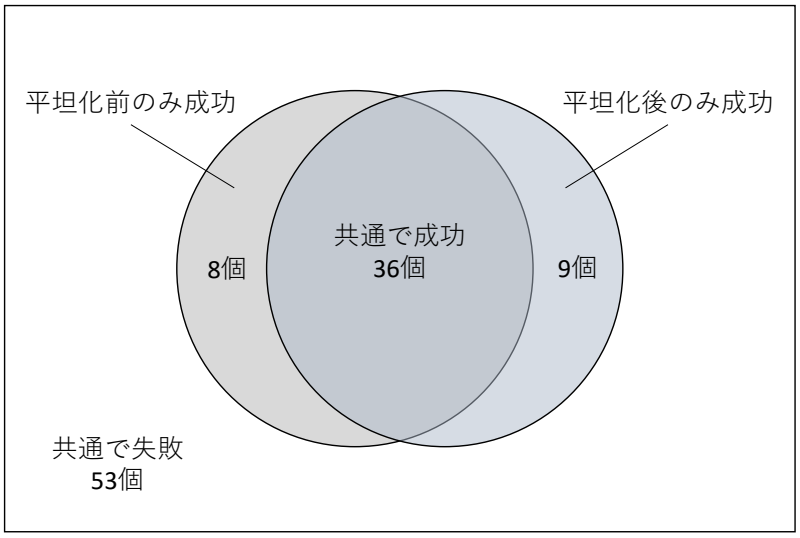

図 17　修正に成功した欠陥集合のベン図

修正できた欠陥は 36 個，平坦化前のみで修正できた欠陥は 8 個，平坦化後のみで修正できた欠陥は 9 個であった．図 16 と図 17 から，OSS で実際に発生した欠陥であってもプログラムの構造によって修正のしやすさが異なると言える．

平坦化により修正の可否が変化する要因として，以下の 3 点考えられる．

- 欠陥限局の精度の低下: kGenProg は欠陥限局に SBFL を利用している．平坦化は同一のブロックに含まれる文の数を増やす操作である．SBFL では，同一のブロックに含まれる文は同一のテストケースで実行されるため，それらに対する疑惑値は全て同じ値になる．欠陥箇所と同じ疑惑値の文が多いほど欠陥箇所が改変される可能性が低下し，修正に失敗しやすくなる．
- 欠陥箇所の分散: 1 つの複雑な文に複数の欠陥が含まれる場合，平坦化により複数の文に欠陥が分散される．欠陥が分散されることで，本来なら 1 つの文の改変で修正できる欠陥に対して複数の文の改変が必要になり，修正に失敗しやすくなる．

- 再利用候補の減少: kGenProg は修正対象のプログラムの文を再利用してプログラムを修正する．複雑な文は複数の演算の組み合わせによって構成されるため，平坦化前のプログラムでは再利用の候補となる文の種類が多い．平坦化で複雑な文が分解されると，全ての文は 1 つの演算で構成されるようになる．文の数は増加するが，組み合わせの考慮がなくなる分，文の種類は減少する．ゆえに，平坦化後には正しい文を再利用できる可能性が高くなり，修正に成功しやすくなる．

以上の結果より，OSS で実際に発生した欠陥であってもプログラムの構造によって修正のしやすさが変化することが分かった．しかし，平坦化前後で修正できる欠陥の数に差がないため，どちらの構造が欠陥を修正しやすいかは現時点で不明である．ゆえに，欠陥を修正する際は構造を変化させた複数のパターンのプログラムを用意し，それらに対して欠陥の修正を試みることで修正可能な欠陥が増加する可能性がある．

7　妥当性の脅威

7.1　実験 1

実験 1 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，実験 1 ではプログラムの構造と自動プログラム修正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 [38] 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．

7.2　実験 2

実験 2 では，Math を対象に kGenProg を実行した．Math 以外のプロジェクトを対象とした場合や kGenProg 以外の自動プログラム修正ツールをを利用した結果を評価できていない．Defects4J に含まれる他のプロジェクトの欠陥情報を実験対象とする，あるいは実験 1 で利用した他の自動プログラム修正ツールを利用することで実験 2 の妥当性を確保できる．

8　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 3 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．
- ネストの深さあるいは循環的複雑度を小さくするプログラムの変換が自動修正適合性を向上させる傾向がある．

また，OSS の 106 個の欠陥を対象に，プログラムの構造によって欠陥の修正しやすさが変化するか確認した結果，構造の変換により新たに 9 つの欠陥を修正できるようになった．

謝辞

本研究に関して，議論の中で有益かつ的確なご助言を頂きました楠本真二教授に心より感謝申し上げます．

本研究を行うにあたり，研究の着想から論文の執筆に至る全過程において，熱心なご指導を頂きました肥後芳樹准教授に深く感謝申し上げます．

本研究に関して，研究の新しいアイディアや発表資料に対するご助言を頂きました柗本真佑助教に心より感謝申し上げます．

研究室生活および研究活動を円滑に進めるために多くのご助力を頂きました事務補佐員の神谷智子氏に深く感謝申し上げます．

研究室生活において様々な相談に乗って頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の東英明氏，同富田裕也氏，同中川将氏，同華山魁生氏に深く感謝申し上げます．

研究室生活を豊かにして頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 1 年の市川直人氏，同出田涼子氏，同荻野翔氏，同藤本章良氏，同前島葵氏に深く感謝申し上げます．

研究室の美化にご助力を頂きました，大阪大学基礎工学部情報科学科 4 年の入山優氏，古藤寛大氏，谷口真幸氏，鶴智秋氏，渡辺大登氏に深く感謝申し上げます．

最後に，講義等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[2] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 2013.

[3] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67, 2019.

[4] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 802–811, 2013.

[5] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Proceedings of the 32nd International Conference on Automated Software Engineering*, pp. 637–647, 2017.

[6] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, pp. 1–12, 2019.

[7] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th International Symposium on Software Testing and Analysis*, pp. 31–42, 2019.

[8] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[9] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the 42nd International Conference on Software Engineering*, p. 615–627, 2020.

[10] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering*, pp. 269–278, 2019.

[11] Keigo Naitou, Akito Tanikado, Shinsuke Matsumoto, Yoshiki Higo, Shinji Kusumoto, Hiroyuki

Kirinuki, Toshiyuki Kurabayashi, and Haruto Tanno. Toward introducing automated program repair techniques to industrial software development. In *Proceedings of the 26th International Conference on Program Comprehension*, pp. 332–335, 2018.

[12] Andréia da Silva, Meyer, Antonio Augusto, Franco Garcia, Anete Pereira, de Souza, and Cláudio Lopes de Souza, Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, Vol. 27, No. 1, pp. 83–91, 2004.

[13] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[14] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 1–11, 2018.

[15] Xuan Bach D. Le, Duc Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pp. 593–604, 2017.

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*.

[17] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[18] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, Vol. 23, No. 5, pp. 3007–3033, 2018.

[19] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, Vol. 29, No. 4, pp. 366–427, 1997.

[20] Edward Miller Miller. *Tutorial Program Testing Techniques*. 1977.

[21] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp. 649–678, 2010.

[22] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. A correlation study between automated program repair and test-suite metrics. In *Proceedings of the 40th International Conference on Software Engineering*, p. 24, 2018.

[23] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313, 2019.

[24] Fowler Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[25] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd International Conference on Automated Software Engineering*, pp. 648–659, 2017.

[26] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th International Symposium on Software Testing and Analysis*, pp. 298–309, 2018.

[27] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1, pp. 213–224, 2016.

[28] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.

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

[30] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kGenProg: A High-Performance, High-Extensibility and High-Portability APR System. pp. 697–698, 2018.

[31] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pp. 65–74, 2010.

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

[33] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, 2016.

[34] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization.

In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pp. 52–63, 2014.

[35] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, pp. 658–662, 2018.

[36] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4j Dataset. *Empirical Software Engineering*, Vol. 22, No. 4, pp. 1936–1964, 2017.

[37] Yoshiki Higo and Shinji Kusumoto. Flattening code for metrics measurement and analysis. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, pp. 494–498, 2017.

[38] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.