修士学位論文

題目

自動修正適合性の再計測
—大規模データセットと多種ミュータント演算子を利用して—

指導教員

楠本 真二 教授

報告者

前島 葵

令和 4 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

2019 年度 修士学位論文

自動修正適合性の再計測
―大規模データセットと多種ミュータント演算子を利用して―

前島 葵

内容梗概

効率的なデバッグ作業の実現を目的とした自動プログラム修正（APR ）に関する研究が数多く行われている．しかしながら，現状の APR 技術で修正できるバグはあまり多くない．このような現状から，APR 技術がバグを修正しやすいプログラムを人間が書くようになれば，APR 技術により多くのバグを自動で修正できるようになるとの考えにより，自動修正適合性という指標が先行研究により提案された．自動修正適合性とは，対象のプログラムにおいて APR 技術がどの程度バグを修正できそうかを表すソフトウェア品質指標である．自動修正適合性の利用により，開発者はこの数値が高いソフトウェアを開発できるようになり，その結果として APR 技術が多くのバグを修正できるようになる．先行研究では，自動修正適合性のアイデアが提案される一方，その計測対象が小規模なプログラムのみであり，APR 技術とプログラム構造の関係が十分に明らかになったとはいえなかった．そこで本研究では，大規模なプログラム群を対象にして自動修正適合性の計測実験を行う．また，プログラム構造を変化させるためのミューテーション演算子を先行研究よりも多く用いることで，より信頼性の高い数値を算出できるように計測を行う．計測の結果，プログラム構造が異なれば自動修正適合性の値も異なることがわかり，プログラムの複雑度や APR ツールによっても自動修正適合性が変化することがわかった．ステートメント数，サイクロマチック数および実行可能経路数に対する自動修正適合性の傾向を調査したところ，単純な文の組み合わせで記述されたプログラムほど修正しやすいことがわかった．

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

1 はじめに 1

2 自動修正適合性 3

2.1 計測手法 3

2.2 先行研究の調査結果 5

2.3 先行研究の計測方法 5

3 本研究における計測の変更点 8

3.1 実験対象プログラム 8

3.2 ミュータント生成器 9

4 実験 1 11

4.1 実験概要 11

4.2 実験対象プログラム 13

4.3 ミュータント生成 13

4.4 APR ツール 14

5 実験 1 結果 15

6 実験 2 34

6.1 実験概要 34

6.2 実験対象プログラム 34

6.3 ミュータント生成 34

6.4 APR ツール 34

7 実験 2 結果 35

8 妥当性への脅威 38

9 おわりに 39

謝辞 40

参考文献 41

図目次

1 自動修正適合性の計測手法の概要 5
2 先行研究と本研究の計測方法の比較 7
3 ミュータント生成数 21
4 問題別の自動修正適合性の箱ひげ図 (自動修正適合性 0 の点を除いたもの) 21
5 ツール別の自動修正適合性の箱ひげ図 (自動修正適合性 0 の点を除いたもの) 23
6 ステートメント数に対する修正できたプログラムの分布 25
7 ステートメント数の多いプログラム 26
8 ステートメント数の少ないプログラム 27
9 サイクロマチック数に対する修正できたプログラムの分布 29
10 サイクロマチック数の多いプログラム 30
11 サイクロマチック数の少ないプログラム 31
12 実行可能経路数に対する修正できたプログラムの分布 33
13 実行可能経路数の多いプログラム 33
14 機能等価メソッド 37

表目次

1 先行研究で利用されたミューテーション演算子 6
2 本研究で追加したミューテーション演算子 10
3 実験対象問題 13
4 問題概要例 14
5 先行研究と本研究の手法で生成されたミュータント数 17
6 ミューテーション演算子別の生成数と修正数 17
7 自動修正適合性 18
8 自動修正適合性の分布 (A 問題) 19
9 自動修正適合性に差が生じたメソッドグループ数 35

1 はじめに

ソフトウェア開発において，デバッグ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, 2]．そのため，デバッグ支援に関する研究が盛んに行われており，自動プログラム修正 (Automated Program Repair: APR) と呼ばれる技術が注目を集めている [3]．APR とは，バグを含むプログラムからバグを自動的に取り除く技術である．

これまでに多くの APR 手法が提案されており [3]，多くのバグに APR 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 [4] に含まれる 395 個のバグのうち 25 個しか正しく修正できなかったと報告された [5]．このような現状から，APR の研究開発に加えて APR の対象であるプログラムの面からも自動修正を支援すべきだと先行研究 [6] で提案された．APR 技術がバグを修正しやすいプログラムを人間が書くようになれば，APR 技術により多くのバグを自動で修正できるようになるとの考えにより，自動修正適合性という指標が先行研究 [6] により提案された．自動修正適合性とは APR 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指標である．自動修正適合性を利用し対象プログラムを APR が作用しやすいように変更できるようになれば，自動で多くのバグを除去できるようになる．

先行研究によって，リファクタリングによる自動修正適合性の違いの分析が行われた．その結果，同じ機能を持つプログラムでも構造が違えば自動修正適合性に差があると明らかになった．しかし，実験の規模が小さく一部のリファクタリングの調査しか行われていない．そのため，プログラム構造の違いが APR にどの程度影響を与えているか十分に明らかではない．また，リファクタリングだけでなくプログラムを構成する制御文など自動修正適合性に影響を与える要因は他にも存在すると著者らは考えた．

本研究では，先行研究で提案された自動修正適合性の計測実験を大規模に行うことにより，より信頼性の高い数値を計測することを目指す．具体的には，より多くのプログラムに対してより多くのミューテーション演算子を利用して変異プログラムを生成する．これにより，先行研究に比べて遙かに多い数の変異プログラムから自動修正適合性の数値が計測できるため，どのようなプログラム構造が APR と親和性が高いのかを明らかにできる．

本研究では以下に示す 7 つの Research Question を設定した．

RQ 1： 先行研究と比較してミュータント数は増加したか？

RQ 2： 同じ機能を持つプログラムでも構造の違いによって自動修正適合性の差が生まれるか？

RQ 3： プログラムの複雑さによってどの程度自動修正適合性が異なるか？

RQ 4： APR ツールによってどの程度自動修正適合性が異なるか？

RQ 5： ステートメント数が少ないほど修正しやすいか？

RQ 6： サイクロマチック数が少ないほど修正しやすいか？

RQ 7： 実行可能経路数が少ないほど修正しやすいか？

同じ機能を持つが構造の異なるプログラムを AtCoder とオープンソースソフトウェアから収集し，自動修正適合性の計測を行った．計測の結果，AtCoder から収集した解答者のプログラム構造の違いによって自動修正適合性に差が生じることがわかった．また，AtCoder Beginner Contest の問題難易度が低い，複雑でない機能を持つプログラムほど自動修正適合性が高くなることがわかった．APR ツールによっても自動修正適合性は異なり，`if` 文の条件式に対して修正を試みる APR ツールが AtCoder の解答プログラムと相性が良いことがわかった．さらに，ステートメント数，サイクロマチック数や実行可能経路数が多いほど APR ツールにとってプログラムが修正しやすいことが明らかになった．言い換えると，複雑なプログラムより単純な処理の組み合わせで記述されている長いプログラムの方が修正がしやすいとわかった．

また，オープンソースソフトウェアから収集した同機能異実装なメソッド群に対しては，276 のグループのうち，61~73 グループは，構造による自動修正適合性の差があることがわかった．

以降，2 節で自動修正適合性について述べる．3 節で本研究における計測の変更点について述べる．4 節で実験について述べ，5 節でその結果について示す．6 節では 2 つ目の実験について述べ，7 節でその結果を示す．8 節で妥当性の脅威について述べ，最後に 9 節で本研究のまとめと今後の課題について述べる．

2 自動修正適合性

本節では先行研究 [6] で提案された自動修正適合性について述べる．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プログラムでは，そのプログラム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．先行研究により以下 4 点に示す自動修正適合性の利用目的が提案された．

- ソフトウェア開発に APR を導入するのかの判断
- APR による保守を前提としたソフトウェア開発
- APR の効果を高めるためのリファクタリングの研究
- APR が誤ったプログラムを生成する問題の原因究明

2.1 計測手法

図 1 に先行研究 [6] で提案された自動修正適合性の計測手法の概要を示す．計測手法の入力は，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．同一のバグに対して同一の 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 [7]．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バグも異なる [8]．

自動修正適合性の計測は以下の 3 つのステップで構成される．

Step1. ミュータント生成

Step2. APR ツールの適用

Step3. 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．

Step1. 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．

Step2. APR ツールの適用　Step1 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，そのミュータントバグを含むと見なされない．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 APR ツールを実行せず，ミュータントは Step3 の計測においても用いない．

Step3. 自動修正適合性の算出　Step2 の実行結果より自動修正適合性を算出する．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である．自動修正適合性は以下の式で算出される．

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

例えば， Step1 でミュータントが 10 個生成され，Step2 で 7 個のミュータントの修正に成功した場合，

$$\text{自動修正適合性} = \frac{7}{10} = 0.7$$

となる．

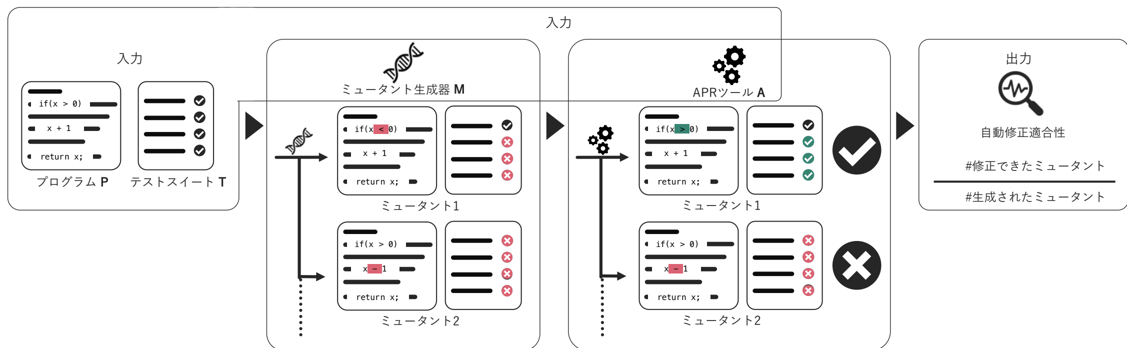


図 1: 自動修正適合性の計測手法の概要

2.2 先行研究の調査結果

先行研究ではリファクタリングを題材に調査が行われた．自動修正適合性を向上させるリファクタリングが分かれば，APR 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ができなかったバグを修正できるようになる．

先行研究の調査により，プログラムの構造によって自動修正適合性に差があると明らかになった．その中でも，一時変数のインライン化のリファクタリングは調査が行われた 7 つの APR ツールのうち 5 つのツールで自動修正適合性が向上した．

2.3 先行研究の計測方法

本節では，先行研究で行われた計測方法について述べ，続く 3 では本研究の計測方法の変更点について述べる．また，図 2 に先行研究と本研究の計測方法の比較を示す．

実験対象プログラム

先行研究では，自動修正適合性の調査のために作成された単一のメソッドで構成される小規模なプログラムに対して実験が行われた．プログラムの構造の違いとしてリファクタリングが題材にされた．様々なリファクタリングがある中で，先行研究では Fowler によって「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に限定し，単一のメソッド内で完結する以下の 6 種のリファクタリングが調査対象とされた [9].

- 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．
- 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．
- 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．

- 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．
- 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．
- 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．

全 6 種の各リファクタリングの題材につき，リファクタリング適応前と適応後のプログラムが作成された．作成されたプログラムの行数は 9∼14 行であり，平均行数は 11 行と小規模であった．

ミュータント生成器

先行研究では，ミュータントを生成するために PIT [10] のミューテーション演算子を利用した[*1]．PIT はミューテーションテストの分野でミュータントの生成に広く用いられている [11]．先行研究では表 1 に示すミューテーション演算子が利用された．

表 1: 先行研究で利用された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 "";`
Null Returns	`return object;`	`return null;`
Change Boolean Literal	`true`	`false`

[*1] 先行研究で True Return および False Return とされていたミューテーションは本研究では Change Boolean Literal と統一した

	先行研究	本研究
実験対象プログラム	6機能2構造 12プログラム 先行研究の著者らが作成	AtCodertとOSSから収集した構造の異なる 解答プログラム群 AtCoder: 14,310 OSS: 728
ミューテーション演算子	PITを利用したミューテーション演算子10種類	PITを利用したミューテーション演算子10種類 SStuBSを利用したミューテーション演算子9種類

図 **2:** 先行研究と本研究の計測方法の比較

3　本研究における計測の変更点

3.1　実験対象プログラム

本実験では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計測するため，同じ機能を持つが構造の異なるプログラム群を対象とする．ここで機能が同じとは，同じ入力を与えると同じ出力を返すメソッドを指す．プログラムの収集には，AtCoder[*2] とオープンソースソフトウェアを利用した．

AtCoder は，AtCoder 社が運営しているプログラミングコンテストサイトである．AtCoder では，定期的にいくつかのプログラミングコンテストが開催されている．AtCoder Beginner Contest(ABC) はプログラミング初心者向けのコンテストであり，最も難易度の低い A 問題から最も難易度の高い D 問題の 4 つの問題で構成されている[*3]．各問題は問題を説明する問題文，入力の制約，入力・出力の形式およびテストケースに当たる入出力例から構成されている．プログラミングコンテストの参加者は，与えられた問題をいかに素早く正確に解けるかを目指し，各々が考えた解答プログラムを提出する．本研究では，ABC から収集された同じ問題を解くが異なる構造を持った 14,310 個のプログラム群を対象とする．

また，オープンソースソフトウェアからも同じ機能を持つが構造の異なるメソッド群を収集した．以降，同じ機能を持つが構造の異なるメソッドを機能等価メソッドと呼ぶ．機能等価メソッドの収集にはメソッドの静的な特徴と動的な振る舞いを利用した．機能等価メソッドの収集は以下の 4 つのステップで構成される．

Step1. 引数の型と返値の型が等しいメソッドのグループ化

Step2. 各メソッドに対してテストケースを生成

Step3. グループ内で相互にテストケースを実行することにより振る舞いの等価性を判定

Step4. Step4 の結果を目視で確認

Step1 において，メソッドの静的な特徴である引数の型と返値の型を利用する．引数の型と返値の型が等しいメソッドは同じ機能を持つ可能性があるとし，同じグループに割り振る．**Step2,3** において **Step1** で振り分けた同じグループに属するメソッドが同じ動的な振る舞いをするかどうかを，それらに対して単体テストを実行することにより判定する．自動テスト生成手法を利用し，対象のメソッドからテストを生成した．自動テスト生成手法は，対象メソッドに対して通過するテストを生成する．つまり，メソッド A から生成した全てのテストケースはメソッド A をパスする．これは，メソッド A から生成したテストケースはメソッド A の振る舞いを表現しているといえる．この性質を利用して，同じグ

[*2] https://atcoder.jp

[*3] 2019 年 4 月 27 日開催の ABC125 以前は 4 題構成

ループに所属する各メソッドからテストを自動生成し，そのテストを相互に実行する．メソッド A から生成したテストをメソッド B がパスし，かつ，メソッド B から生成したテストをメソッド A がパスする場合は，メソッド A とメソッド B の振る舞いがある程度等価であることを表している．**Step4** では，**Step2,3** において機能等価であるとわかったメソッドを目視で確認する．

以上から収集された 276 の機能等価メソッドグループに属する 728 のメソッドを実験対象とする．

3.2 ミュータント生成器

本研究では，ミュータントを生成するために Simple Stupid Bugs(SStuBs) [12] から得られるバグパターンを用いた．SStuBs は，オープンソース Java プロジェクトから抽出された単一ステートメントのバグ修正変更のデータセットである．SStuBs では，頻出した単一ステートメントのバグ修正が 16 種類のバグパターンに分類された．本研究では，著者が SStuBs のバグパターンを参考にしてソースコードを書き換えるツールを実装した．

16 種類のバグパターンの中で，表 2 に示す 9 種類のミューテーション演算子を実装した．実装しなかったバグパターンについて，3 種類は先行研究で用いられたミューテーション演算子に含まれているため除外した．2 種類は Java の例外処理に関わる `throws` 節の取り外しによるバグ修正であった．これは，ミューテーション演算子としてプログラムに適用しても，自動修正適合性計測の条件であるコンパイルが通るミュータントを生成できないため除外した．また別の 1 種類は `static` 修飾子の取り外しによるバグ修正であり，同様にコンパイルが通るミュータントを生成できないため除外した．また別の 1 種類はメソッド呼び出しを，引数の数が多いオーバーロードメソッドの呼び出しに変更するバグ修正である．引数の少ないオーバーロードメソッドへの書き換えを行うミューテーション演算子は Same Function Less Args として実装を行ったが，引数の多いオーバーロードメソッドへの書き変えのミューテーション演算子はコンパイルが通るミュータントを生成するための適切な引数を生成できないため除外した．

表 2: 本研究で追加したミューテーション演算子

<table>
<tr><th rowspan="2">ミューテーション演算子</th><th colspan="2">変換例</th></tr>
<tr><th>変換前</th><th>変換後</th></tr>
<tr><td>Change Identifier Used</td><td><code>Superclass a</code></td><td><code>Subclass a</code></td></tr>
<tr><td>Change Numeric Literal</td><td><code>if(x<0)</code></td><td><code>if(x<1)</code></td></tr>
<tr><td>Wrong Function Name</td><td><code>getEdgeCount()</code></td><td><code>getNodeCount()</code></td></tr>
<tr><td>Same Function Less Args</td><td><code>method(a,b,c)</code></td><td><code>method(a,b)</code></td></tr>
<tr><td>Same Function Change Caller</td><td><code>a.method()</code></td><td><code>b.method()</code></td></tr>
<tr><td>Same Function Swap Args</td><td><code>method(a,b)</code></td><td><code>method(b,a)</code></td></tr>
<tr><td>Change Operand</td><td><code>a+b</code></td><td><code>a+c</code></td></tr>
<tr><td>More Specific If</td><td><code>if(a&&b)</code></td><td><code>if(a)</code></td></tr>
<tr><td>Less Specific If</td><td><code>if(a||b)</code></td><td><code>if(a)</code></td></tr>
</table>

4　実験 1

4.1　実験概要

実験では，どのようなプログラム構造が APR と親和性が高いのかを明らかにする．以下に示す 7 つの Research Question を設定した．

RQ 1 : 先行研究と比較してミュータント数は増加したか？
RQ 2 : 同じ機能を持つプログラムでも構造の違いによって自動修正適合性の差が生まれるか？
RQ 3 : プログラムの複雑さによってどの程度自動修正適合性が異なるか？
RQ 4 : APR ツールによってどの程度自動修正適合性が異なるか？
RQ 5 : ステートメント数が少ないほど修正しやすいか？
RQ 6 : サイクロマチック数が少ないほど修正しやすいか？
RQ 7 : 実行可能経路数が少ないほど修正しやすいか？

以降，詳しく説明する．

RQ1 : 先行研究と比較してミュータント数は増加したか？

先行研究の課題であったミュータント数の少なさを本研究では解決できているか確認する．本研究で新たに追加したミュータント演算子が AtCoder Beginner Contest のプログラムに対してどの程度ミュータントを生成できたか確かめる．また，生成されたミュータントのどの程度が修正できたか確認する．

RQ2 : 同じ機能を持つプログラムでも構造の違いによって自動修正適合性の差が生まれるか？

AtCoder Beginner Contest に提出された同じ問題を解く異なる解答者が提出したプログラムを調査し，異なる構造を持つプログラムによって自動修正適合性に差が生まれるか調査する．同じ問題を解くプログラムでも解答者によって使用する制御文や分岐の構造に違いが生まれる．プログラムの構造によって自動修正適合性に差が生まれるか，差が存在するとしたらどの程度あるのか調査する．

RQ3 : プログラムの複雑さによってどの程度自動修正適合性が異なるか？

AtCoder Beginner Contest には難易度の異なる 4 つの問記が存在する．難易度が低い問題は `if` 文や `for` 文を用いた簡単な計算処理のみで解答プログラムが記述できる．対して難易度が高い問題はソートなどアルゴリズムの知識が必要とされるものも存在する．自動修正適合性は，ABC の問題難易度によって差が生じるか調査する．例えば，難易度が低い問題の方がプログラムもシンプルに記述しやすいため

に修正しやすいのかどうか，難易度の高い問題はプログラムも複雑になりやすいために修正しにくいのかどうか調査する．

RQ4：APR ツールによってどの程度自動修正適合性が異なるか？

実験では，4 つの APR ツールを用いた．ツールによって修正のための戦略が異なるため自動修正適合性にも差が生じるか調査を行う．もし自動修正適合性の高い APR ツールが存在すれば，その修正戦略は AtCoder のプログラムと相性が良いといえる．

RQ5：ステートメント数が少ないほど修正しやすいか？

一般的に人がプログラムを書くときには，可読性を高めるためにステートメント数が少ないほど良いとされ，長すぎるメソッドは可読性が低いとされる [13]．APR にとっても人と同様にステートメント数が少ないプログラムほど修正しやすいのか調査を行う．ステートメント数が少ないほど修正しやすいと示せれば，ステートメント数を少なくするようなリファクタリングを適用すれば APR にとって修正しやすいプログラムに加工できる．例えば `if` 文の統合が効果的だとわかる．逆に，ステートメント数が多いほど修正しやすいと示せれば，ステートメント数を多くするようなリファクタリングを適用すれば良いとわかる．

RQ6：サイクロマチック数が少ないほど修正しやすいか？

サイクロマチック数はプログラムの代表的な複雑度メトリクスとして良く用いられる [14, 15]．サイクロマチックは数は，McCabe によって提案されたソフトウエアメトリクスである [14]．プログラムの制御の流れを有向グラフで表した時のノード数を `v`，エッジ数を `e`，連結成分数を `p` とすると，`e - n + 2p` で表される．具体的にはサイクロマチック数はプログラム文が別の文にジャンプする箇所を計測する．メソッド呼び出しによって増加し，分岐にあたる `if` 文，`while` 文，`for` 文や `case` によっても増加する．また，条件式のブール演算子`&&`,`||`でも加算される．McCabe はメソッドあたりのサイクロマチック数が 10 以下が望ましいと述べている [14]．

一般的に人がプログラムを書くときには，McCabe が述べたようにサイクロマチック数が少ないほど良いとされる．APR にとっても人と同様にサイクロマチック数が少ないプログラムほど修正しやすいのか調査を行う．サイクロマチック数が少ないほど修正しやすいとわかれば，分岐やネストを減少させるリファクタリングを適用すれば APR にとって修正しやすいプログラムに加工できるとわかる．対して，サイクロマチック数が多いほど修正しやすいとわかれば，APR ツールにとって人とは異なりプログラムの複雑度は関係ないと示せる．

RQ7：実行可能経路数が少ないほど修正しやすいか？

サイクロマチック数同様，実行可能経路数 (NPath) はプログラムの代表的な複雑度メトリクスとして良く用いられる [16]．実行可能経路数 (NPath) は，あるメソッドを通る非周期的な実行パスの数で表される．実行可能経路数はメソッドの開始から終了までの経路数を計測するが，サイクロマチック数は制御文の数のみを計測する．例えば，10 個の `if-else` 文が連続する場合に，サイクロマチック数は 20 であるが，実行可能経路数は 2 の 10 乗の 1,024 になる．一般的に，実行可能経路数が 200 を超えるメソッドは複雑であるといわれる．

一般的に実行可能経路数は，必要なテストケースを増加させてしまう原因にもなるため少ないほど良いとされる．APR にとっても人と同様に実行可能経路数が少ないプログラムほど修正しやすいのか調査を行う．

4.2 実験対象プログラム

ABC の 101 回から 120 回までの A, B, C, D 問題を調査対象とした．表 3 に実験対象プログラム数を示し，例として表 4 に ABC の 101 回, 102 回の問題概要を示す．ABC の問題の難易度が上がるにつれ解答者が減少するため，A 問題の解答プログラムは多く収集できたが，A 問題の解答プログラムは少数しか収集できなかった．

4.3 ミュータント生成

ミュータント生成には，表 1 に示す先行研究の 10 種のミューテーション演算子と表 2 に示す本研究で新たに追加した 9 種のミューテーション演算子を利用した．

AtCoder の解答プログラムには，プログラムの実行時間の短縮のため自作の Scanner クラスや Math クラスが存在する場合がある．そのため，本研究ではミュータント対象となるプログラム部分を，`main` メソッドおよび `solve` メソッドのみとした．`solve` メソッドは ABC 参加者が解答プログラムの記述の

表 3: 実験対象問題

問題	解答プログラム数
A 問題	5,546
B 問題	4,567
C 問題	2,949
D 問題	1,247
合計	14,310

ため作成することの多いメソッドである.

4.4 APR ツール

本実験では自動修正適合性の計測に以下の APR ツールを利用した．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.

jGenProg [17] GenProg [18] の Java 実装版である．遺伝的プログラミングに基づいてプログラムを修正する [19].

jMutRepair [20] Debroy と Wong らによって提案された修正方法であり，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 [21].

Cardumen [22] 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．jGenProg と同様に文の挿入，削除，置換によって修正を行う．jGenProg は文をそのまま利用するが，Cardumen は文をテンプレートとみなし，文の要素を別の要素に置き換えた文も利用して修正を行う.

jKali [20] Qi らによって提案された修正方法であり，コードの削除や `return` 文の挿入によりプログラムを修正する [23].

4 つの APR ツールで公平に実験を行うため，実行時間制限を 10 秒に統一した.．また，jMutRepair は網羅的に探索を行うが，jGenProg, Cardumen, jKali は欠陥限局を用いている．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．欠陥限局は，APR の有効性に影響を与えるといわれる [24]. 公平に実験を行うため jGenProg, Cardumen, jKali ではデフォルト設定である GZoltar [25] を利用した.

表 4: 問題概要例

問題	タイトル	概要
ABC101 A 問題	Eating Symbols Easy	“+”, “-” のシンボルの出現回数を計算
ABC101 B 問題	Digit Sums	与えられた整数 N の各桁の和によって N が割り切れるか判定
ABC102 A 問題	Multiple of 2 and N	与えられた正整数 N と 2 の最小公倍数を計算
ABC102 B 問題	Maximum Difference	与えられた整数列の 2 要素の差の絶対値の最大を計算

5 実験 1 結果

本節では，7 つの Research Question への回答をする．

RQ1：先行研究と比較してミュータント数は増加したか？

表 5 に先行研究と本研究のミューテーション演算子から生成されたミュータント数および修正数を示す．表 1 に示す先行研究の 10 種のミューテーション演算子からは 53,281 個のミュータントが生成された．表 2 に示す本研究で新たに追加した 9 種のミューテーション演算子から 60,400 個のミュータントが生成された．新たに追加したミュータント演算子からも先行研究で用いられたミューテーション演算子と同程度の個数のミュータントを生成できることが確認できた．

また，表 6 にミュータント別の生成数と修正数を示す．ミュータントの生成数に注目すると，最もミュータントが生成されたミューテーション演算子は，ChangeNumericLiteral で，次点が Math，さらに NegateConditionals であった．AtCoder Beginner Contest は入力で与えられた数値や数列に対して処理を行う問題が多く存在したことから，数値計算の演算子や，数値自体を変化させるミュータントは多く生成できた．

対して，NullReturns, ChangeIdentifierUsed, SameFunctionLessArgs はミュータントを生成することができなかった．本研究では ABC を対象としたため，基本的に単一のクラス単一のメソッドで解答プログラムを記述できる．そのため，他のクラスや他のメソッドを参照するミューテーション演算子はミュータントの生成できなかった．NullReturns が適用できなかった理由は，実験対象を AtCoder Beginner Contest の `main` メソッドおよび `solve` メソッドに制限したため，これらのメソッドは int 型や void 型，String 型以外の戻り値を取らなかったためである．

ChangeIdentifierUsed が適用できなかった理由は，AtCoder Beginner Contest は単一のクラスを持つ解答プログラムを提出するため，インナークラスを除いて他のクラスを用いないためである．そのため，同じ型を持つ異なる識別子は存在しなかった．SameFunctionLessArgs は，メソッドの引数が 1 つ取り除かれた引数の少ないオーバーライドメソッドが存在すればミューテーションを行う．ABC では解答プログラムにオーバーライドメソッドを実装する場合が存在せず，今回の題材ではミュータントを生成できなかった．

表 6 の修正率に注目すると，最も修正できたミューテーション演算子は NegateConditionals であることがわかる．これは，境界条件に関するミュータントであり，jMutRepair の修正戦略である `if` 文の条件式の書き換えに合致したことから多く修正できた．

修正できなかったミューテーション演算子は， EmptyReturns と WrongFunctionName であった．EmptyReturns は `return` 文の文字列を `null` にする演算子である．このことから，APR ツールにとっ

て文字列操作は不得意であることがわかる．また，WrongFunctionName は変数名を別の変数名に変える演算子であり，APR ツールはメソッド呼び出しに対する修正は他の修正に比べて不得意だといえる．

RQ1 への回答として，「先行研究で用いられたミューテーション演算子からは **53,281** 個のミュータントが生成された．本研究で新たに追加した **9** 種のミューテーション演算子からは **60,400** 個のミュータントが生成された．新たにミューテーション演算子を増加することで，ミュータント数を増加できたといえる．また新たに生成されたミュータントは **0%∼22%** 程度修正できた」といえる．

表 5: 先行研究と本研究の手法で生成されたミュータント数

	先行研究のミューテーション演算子から生成されたミュータント数 (修正数)	本研究のミューテーション演算子から生成されたミュータント数 (修正数)
A 問題	15,474(4,479)	15,662(7,181)
B 問題	18,691(3,810)	22,900(10,444)
C 問題	9,247(758)	11,126(1,996)
D 問題	9,869(882)	10,712(2,175)
合計	53,281(9,929)	60,400(21,796)

表 6: ミューテーション演算子別の生成数と修正数

	ミューテーション演算子名	生成数	修正数	修正率 (%)
先行研究	ConditionalsBoundary	6,686	2922	44%
	Increments	2,639	446	17%
	InvertNegatives	1,014	151	15%
	Math	29,739	4772	16%
	NegateConditionals	16,406	14305	87%
	VoidMethodCalls	7,513	371	5%
	PrimitiveReturns	25	15	60%
	EmptyReturns	41	0	0%
	NullReturns	0	0	NaN
	ChangeBooleanLiteral	2,293	999	44%
本研究	MoreSpecificIf	1,473	328	22%
	LessSpecificIf	1,088	58	5%
	SameFunctionSwapArgs	28	3	11%
	WrongFunctionName	37	0	0%
	ChangeOperand	6604	1,076	16%
	ChangeIdentifierUsed	0	0	NaN
	SameFunctionLessArgs	0	0	NaN
	SameFunctionChangeCaller	3,675	562	15%
	ChangeNumericLiteral	45,552	9,365	21%

RQ2：同じ機能を持つプログラムでも構造の違いによって自動修正適合性の差が生まれるか？

表 7 に ABC の解答プログラムに対しての自動修正適合性の計測結果を示す．縦軸はツール別の結果で横軸は問題別の結果である．自動修正適合性は 0 と 0 より大きい値で分けている．例えば ABC の A 問題の jGenProg に対する結果は，自動修正適合性が 0 のプログラム数が 4,692 であり，自動修正適合性が 0 より大きいプログラム数は 854 であり 15% を占めた．APR ツールは全てのバグを修正できるわけではなく，修正できない場合のほうが多い．しかし，修正できたプログラムも存在する．表 7 の右下の 4 つの APR ツールと 4 つの問題の合計欄に注目すると，自動修正適合性が 0 のプログラム数が 45,369 であり，自動修正適合性が 0 より大きいプログラム数は 11,871 であり 21% を占めた．

次に，自動修正適合性の分布をより詳しく表 8 に示す．表 8 には，A 問題のツール別の自動修正適合性の分布，中央値と分散を示す．自動修正適合性の分布については 0.1 区間区切りで示している．例えば，jGenProg の自動修正適合性は 0 のプログラムが 3,884 個，0 より大きく 0.1 より小さいプログラムが 65 個あることを示している．

表 8 から同じ機能を持つプログラムでも，解答者のプログラムの構造によって自動修正適合性の値は変化することがわかった．

次に，中央値と分散については，中央値は jMutRepair だけ 0.18 と高かったが，他の APR ツールはほとんど 0.0 に近いという結果になった．分散についても 0.0 に近いという結果だった．

RQ2 への回答として，「**同じ機能を持つプログラムでも，プログラムの構造によって自動修正適合性は異なることがわかった．自動修正適合性が 0 になる構造が多く存在する一方で，自動修正適合性が 0 を超える構造も 21% 程度存在した**」といえる．

表 7: 自動修正適合性

APR ツール	自動修正適合性 (問題別)									
	A 問題		B 問題		C 問題		D 問題		合計	
	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]	0.0	(0.0, 1.0]
jGenProg	4,692	854	3,922	646	2,539	410	1,003	244	12,156	2,154
jMutRepair	3,336	2,210	1,945	2,623	1,851	1,098	687	560	7,819	6,491
Cardumen	4,899	647	3,864	704	2,499	450	977	270	12,239	2,071
jKali	5,203	343	4,289	279	2,681	268	982	265	13,155	1,155
合計	18,130	4,054	14,020	4,252	9,570	2,226	3,649	1,339	45,369	11,871

表 8: 自動修正適合性の分布 (A 問題)

APR ツール	自動修正適合性											中央値	分散
	0.0	$(0.0, 1.1]$	$(0.1, 0.2]$	$(0.2, 0.3]$	$(0.3, 0.4]$	$(0.4, 0.5]$	$(0.5, 0.6]$	$(0.6, 0.7]$	$(0.7, 0.8]$	$(0.8, 0.9]$	$(0.9, 1.0]$		
jGenProg	4,692	48	326	175	114	63	23	58	17	16	14	0.01	0.02
jMutRepair	3,336	26	149	192	584	520	96	439	93	19	92	0.19	0.05
Cardumen	4,899	87	245	107	104	47	25	15	10	4	3	0.00	0.01
jKali	5,203	37	109	108	59	21	3	2	3	1	0	0.00	0.00

RQ3：プログラムの複雑さによってどの程度自動修正適合性が異なるか？

AtCoder Beginner Contest の問題難易度による自動修正適合性の差を調査する．難易度によって自動修正適合性が 0 より大きくなるプログラム数がどの程度異なるか，自動修正適合性の値はどの程度変化するかの 2 点について述べる．

はじめに，自動修正適合性が 0 より大きくなるプログラム数について述べる．表 7 の下段の 4 つの APR ツールの合計に注目すると，A 問題は自動修正適合性が 0 より大きいプログラム数が 4,054 であり全体の 18 % のプログラムが修正できている．B 問題は自動修正適合性が 0 より大きいプログラム数が 4,252 であり全体の 23 % のプログラムが修正できている．C 問題は自動修正適合性が 0 より大きいプログラム数が 2,226 であり全体の 19 % のプログラムが修正できている．D 問題は自動修正適合性が 0 より大きいプログラム数が 1,339 であり全体の 27 % のプログラムが修正できている．このことから，修正できたプログラム数の割合が最も多い問題は D 問題であるとわかった．

また，図 3 に問題別のミュータント生成数を載せる．図 3 から A 問題はミュータントの生成数が少なく，D 問題は多いことがわかる．これは，D 問題は他の問題に比べてプログラムが長くなる場合が多いためミューテーション演算子が適用できる箇所も多くなるからである．

生成されたミュータントのうち 1 つでも修正することができれば，自動修正適合性は 0 より大きくなる．そのため，生成されたミュータント数の多い D 問題は自動修正適合性は 0 より大きくなりやすいと考えれられる．

次に，問題ごとの自動修正適合性の差について述べる．図 4 に問題の難易度による自動修正適合性の差を 4 つの APR ツール別に示す．ただし，自動修正適合性が 0 の点を除いている．図 4 では上から A, B, C, D 問題の順に並んでいる．どの APR ツールでも A 問題の自動修正適合性の値が高い傾向にあり，D 問題についてはどの APR ツールでも最も中央値が低い傾向にあることがわかる．また，B 問題と C 問題については jMutRepair を除いて自動修正適合性の値が似た傾向にある．

これらの結果から，ABC の難易度の低い A 問題は修正できるプログラム数は少なかったが，修正できたプログラムに対しての自動修正適合性の値は高かったことがわかる．難易度の高い D 問題はミュータントの生成数が多くなるため自動修正適合性が 0 より大きいプログラム数が多かったが，自動修正適合性の値は低い傾向にあった．

RQ3 への回答として，「**プログラムの難易度によって自動修正適合性に差が生じる．難易度の低い問題ほど自動修正適合性の高いプログラムが多かった．難易度の高い問題ほど，自動修正適合性が 0 を超えるプログラム数が多い傾向にあったが自動修正適合性の値自体は低かった**」といえる．

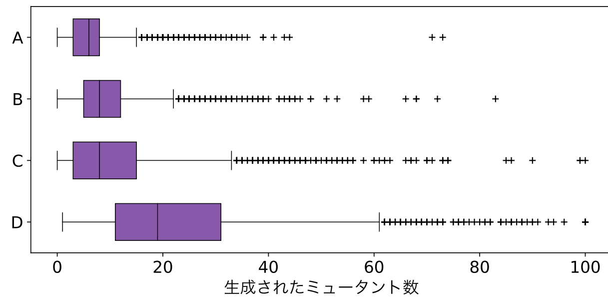
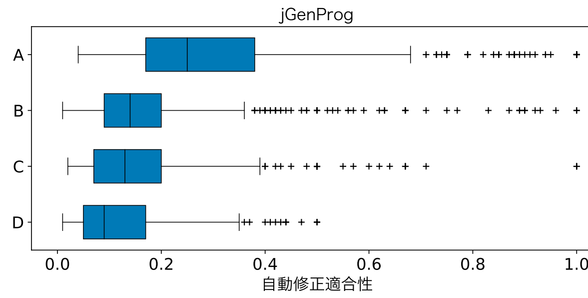
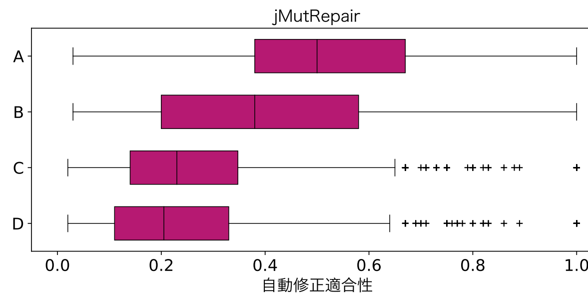
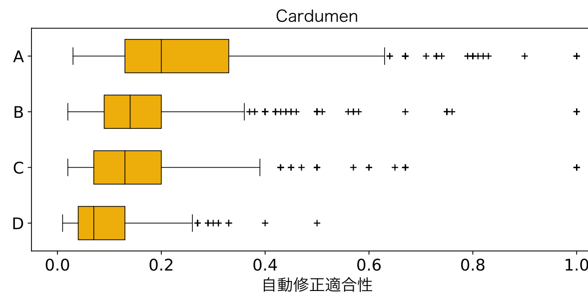


図 3: ミュータント生成数







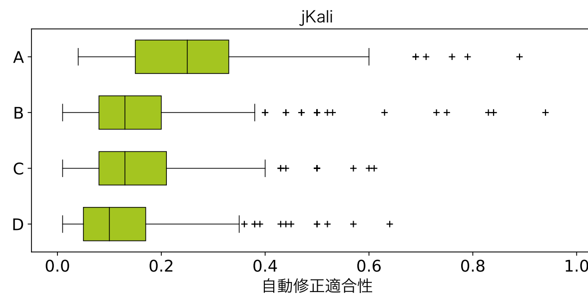


図 4: 問題別の自動修正適合性の箱ひげ図 (自動修正適合性 0 の点を除いたもの)

RQ4：APR ツールによってどの程度自動修正適合性が異なるか？

4 つの APR ツールによる自動修正適合性の差を調査する．はじめに，表 7 の右列の APR ツール別の自動修正適合性が 0 より大きいプログラム数に注目する．jGenProg は自動修正適合性が 0 より大きいプログラム数が 2,154 であり全体の 15 % のプログラムが修正できている．jMutRepair は自動修正適合性が 0 より大きいプログラム数が 6,491 であり全体の 45 % のプログラムが修正できている．Cardumen は自動修正適合性が 0 より大きいプログラム数が 2,071 であり全体の 14 % のプログラムが修正できている．jKali は自動修正適合性が 0 より大きいプログラム数が 1,155 であり全体の 8 % のプログラムが修正できている．このことから，自動修正適合性が 0 より大きいプログラムの割合は，jMutRepair, jGenProg,Cardumen, jKali の順に多いことがわかった．これは， 条件式あるいは `return` 文のミューテーションにより修正を行う jMutRepair が最も ABC のプログラムと相性が良いからだと考える．ABC は入力された数値に対して数値計算や判定を行う問題が多く，分岐が多用されるため条件式が多く登場した．また，文の再利用によって修正を行う戦略を取る jGenProg と Cardumen は修正できるプログラム数も近いという結果になった．jGenProg に対して Cardumen は文をテンプレートとした再利用を行うため jGenProg より優れていると述べられているが，本実験では差が生まれなかった．これは，実験では実行時間を制限してしまったため Cardumen にとって十分な探索が行えなかったからだと考えられる．いくつかの問題に対して APR ツールの実行制限時間を伸ばして実験したところ，Cardumen によって新たに修正できたミュータントが存在した．また，jKali の戦略である文の削除や `return` 文の挿入は他の戦略に比べてあまり効果がなかったといえる．

図 5 に APR ツールによる自動修正適合性の差を 4 つの APR ツール別に示す．ただし，自動修正適合性が 0 の点を除いている．図 4 では ABC 問題順に図が並び，各図の中は APR ツール別に並んでいる，どの問題でも jMutRepair の自動修正適合性の値が高い傾向にあることがわかる．A 問題では jGenProg の自動修正適合性の中央値が 0.25 であり，jMutRepair は 0.5 であり，Cardumen は 0.2 であり，jKali は 0.25 であった．jMutRepair と Cardumen では自動修正適合性の値の差が 0.3 存在した．

RQ1 への回答として，「**APR ツールによって自動修正適合性に差が生じることがわかった．4 つの APR ツールの中では jMutRepair が自動修正適合性が高い傾向にあり，数値計算や条件判定の多い ABC のプログラムと相性が良いとわかった．自動修正適合性の高い jMutRepair と低い Cardumen では 0.3 程度自動修正適合性に差が生じることがわかった**」といえる．

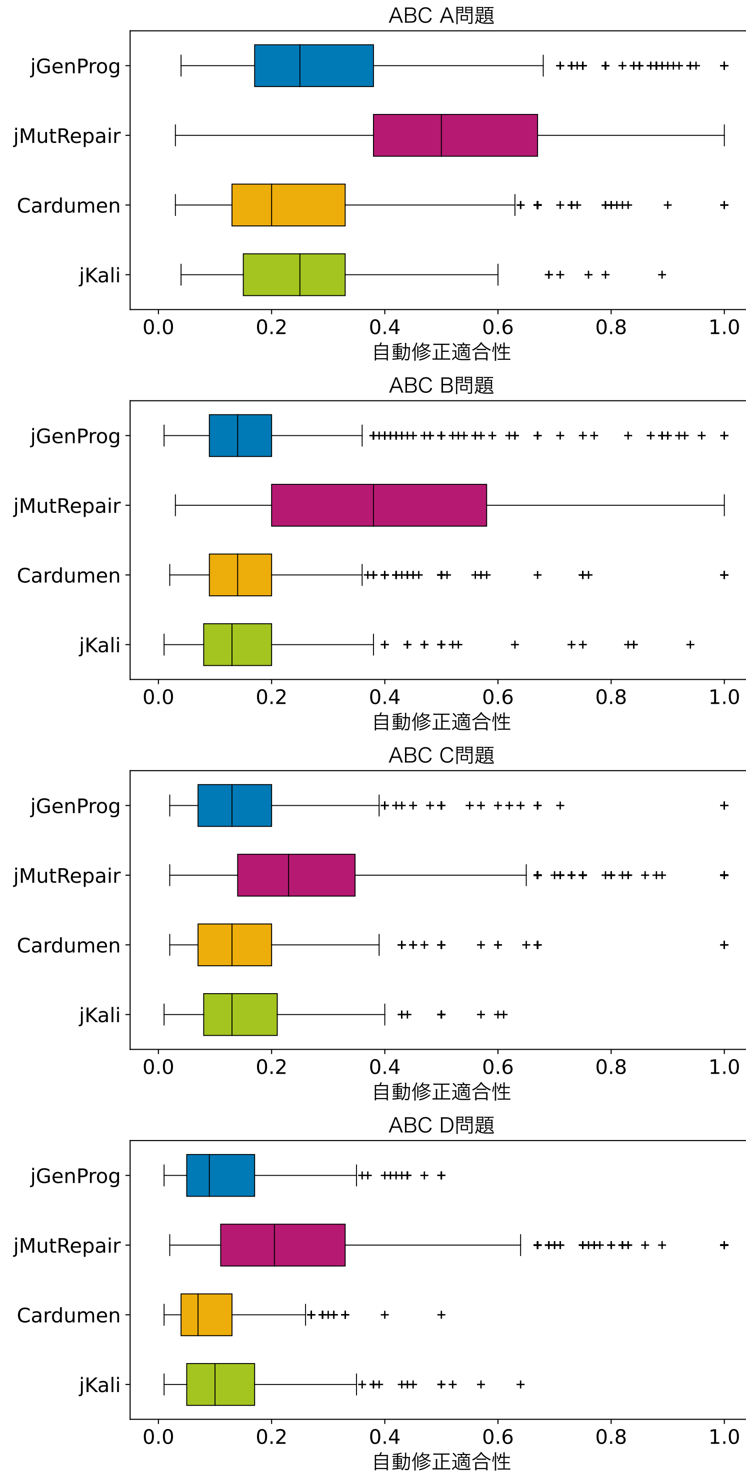


図 5: ツール別の自動修正適合性の箱ひげ図 (自動修正適合性 0 の点を除いたもの)

RQ5：ステートメント数が少ないほど修正しやすいか？

図 6 にステートメント数に対する修正できたプログラムと修正できなかったプログラムの分布を示す．修正できたプログラムとは，自動修正適合性が 0 より大きいプログラムを指す．図 6 の箱ひげ図から，修正できなかったプログラムに比べて修正できたプログラムはステートメント数が多いことがわかる．同じ機能を持つプログラムでも，解答者の平均ステートメント数より多くのステートメント数で記述された方が修正しやすい．

図 7(a) および図 7(b) に ABC の 101 回 A 問題のステートメント数が多かったプログラムを示す．これは，“+”,“-” のシンボルの出現回数を計算するプログラムである．図 7(a) のプログラムはステートメント数が 29 であり，全解答プログラムの中で最も多かった．自動修正適合性は jKali は 0 であったが，jGenProg, Cardumen は 0.53 と比較的高かった．このプログラムは `if` 文による分岐を多用しているためステートメント数が多かった．

また，図 7(b) のプログラムはステートメント数が 24 であり，図 7(a) と同様に `if` 文による分岐や `for` 文による繰り返しが用いられているためステートメント数が多かった．

図 8(a) および図 8(b) に ABC の 101 回 A 問題のステートメント数が少なかったプログラムを示す．図 8(a) のプログラムはステートメント数が 3 であり，図 7(a) と図 7(b) 同様の処理をラムダ式や三項演算子を用いることで，1 行で行っている．自動修正適合性は 4 つの APR ツール全てで 0 であった．1 行あたりの処理が複雑なプログラムは修正しづらい傾向にあるとわかる．

また，図 8(b) のプログラムも同様にステートメント数が 3 であった．プログラムの記述は図 7(a) と似ており，1 行に処理が詰め込まれている．replace メソッドによる文字列の置換などを用いて `if` 文などによる分岐の必要のない実装がされている．図 7(a) と同じく自動修正適合性は 4 つの APR ツール全てで 0 であった．jMutRepair にとっては修正対象である条件式が存在しないため修正できず，jGenProg と Cardumen にとっては文の再利用が低かったことが原因だと考えられる．図 8(a) と図 8(b) で用いられている文は，この問題を解くために特化しているため再利用が低い．

ステートメント数が多いほど，1 つの文あたりの複雑度は低下し簡単な演算や分岐でプログラムが構成される．ステートメント数が少ないほど，三項演算子やラムダ式が用いられ分岐が必要とされない．人にとってはステートメント数の少ないプログラムのほうが可読性が高く良いとされるが，APR ツールにとってはステートメント数の多い方が修正しやすいことがわかった．

RQ5 への回答として，「**ステートメント数の多いプログラムの方が修正しやすい**」といえる．

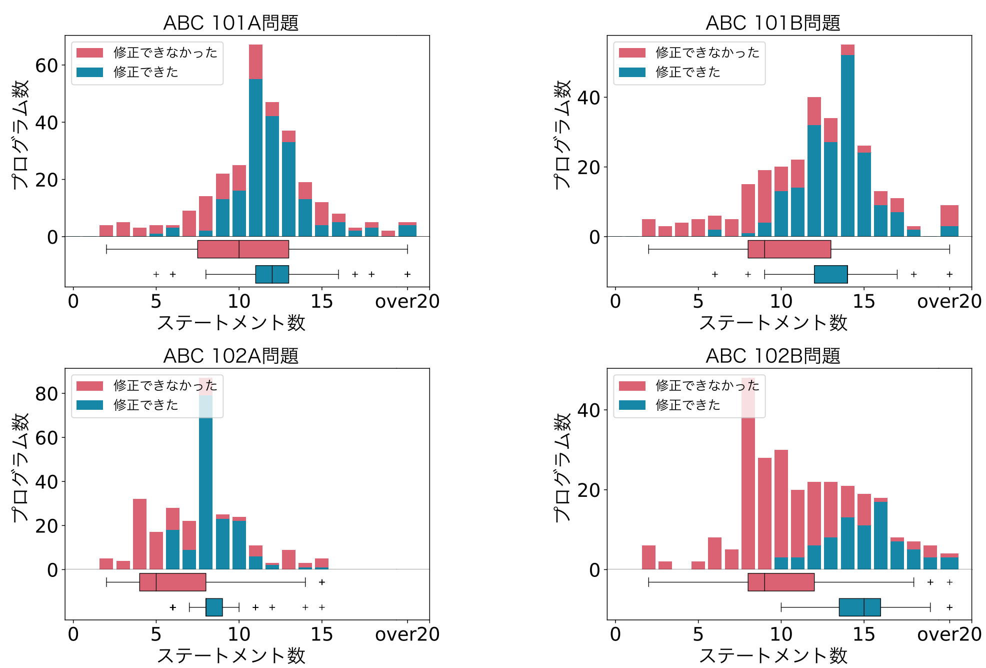


図 6: ステートメント数に対する修正できたプログラムの分布

(a) ステートメント数の多いプログラム

自動修正適合性(APRツール別)				ステートメント数
jGenProg	jMutRepair	Cardumen	jKali	
0.53	0.24	0.53	0	29

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.Scanner;

public class Main{
    public static void main(String[] args) {
        int i = 0;
        Scanner sc = new Scanner(System.in);
        String n = sc.next();
        char a = n.charAt(0);
        char b = n.charAt(1);
        char c = n.charAt(2);
        char d = n.charAt(3);
        if (a == '+') {
            i++;
        }else{
            i--;
        }
        if (b == '+') {
            i++;
        }else{
            i--;
        }
        if (c == '+') {
            i++;
        }else{
            i--;
        }
        if (d == '+') {
            i++;
        }else{
            i--;
        }
        System.out.println(i);
    }
}
```

(b) ステートメント数の多いプログラム

自動修正適合性(APRツール別)				ステートメント数
jGenProg	jMutRepair	Cardumen	jKali	
0	0.82	0	0	24

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String n = sc.next();
        char tar ='+';
        int a =0;
        for(char x: n.toCharArray()){
            if(x == tar){
                a++;
            }
        }
        if(a == 0){
            System.out.println("-4");
        }else if(a == 1){
            System.out.println("-2");
        }else if(a == 2){
            System.out.println("0");
        }else if(a == 3){
            System.out.println("2");
        }else{
            System.out.println("4");
        }
    }
}
```

図 7: ステートメント数の多いプログラム

(a) ステートメント数の少ないプログラム

自動修正適合性(APRツール別)				ステートメント数
jGenProg	jMutRepair	Cardumen	jKali	
0	0	0	0	3

```
import java.util.*;

class Main{
	public static void main(String[] A){
		Scanner s=new Scanner(System.in);
		System.out.println(s.next().chars().map(o->o=='+'?1:-1).sum());
	}
}
```

(b) ステートメント数の少ないプログラム

自動修正適合性(APRツール別)				ステートメント数
jGenProg	jMutRepair	Cardumen	jKali	
0	0	0	0	3

```
import java.io.*;
import java.util.*;

public class Main {
	public static void main(String[] args) {
		Scanner in = new Scanner(System.in);
		System.out.println(4-in.nextLine().replace("+", "").length()*2);
	}
}
```

図 8: ステートメント数の少ないプログラム

RQ6：サイクロマチック数が少ないほど修正しやすいか？

サイクロマチック数の計測には，品質計測ツールの 1 つである PMD [*4]を用いた．PMD では，McCabe の定義に従ってサイクロマチック数が計測される [14]．図 9 にサイクロマチック数に対する修正できなかったプログラムと修正できたプログラムの分布を示す．修正できなかったプログラムとは自動修正適合性が 0 のプログラムを指し，修正できたプログラムとは自動修正適合性が 1 より大きいプログラムを指す．図 9 の箱ひげ図の中央値に注目すると，修正できなかったプログラムに比べて修正できたプログラムはサイクロマチック数が多い傾向にあることがわかる．

図 10(a) および図 10(b) に ABC の 101 回 A 問題のサイクロマチック数が大きいプログラムを示す．RQ5 で示した問題と同様の "+", "-" のシンボルの出現回数を計算するプログラムである．図 10(a) のプログラムはサイクロマチック数が 17 であり，全 296 の解答プログラムの中で 2 番目に大きかった．最も多いサイクロマチック数は 18 だったが，自動修正適合性が全ての APR ツールで 0 であったため，次にサイクロマチック数が大きいプログラムを示している．自動修正適合性は Cardumen は 0 であったが，jGenProg, jMutRepair, jKali では 0.1 であった．このプログラムは `if` 文による分岐で全ての入力パターンを網羅している．ブール演算子`||`が多用されることによってサイクロマチック数が増加していた．`if` 文による分岐が多用されていて人にとっては行数が長く理解しづらいプログラムであるが，APR ツールにとっては，修正しやすかった．理由としては，jMutRepair の修正対象の条件式が存在することや，1 つの文あたりは複雑でないため文の再利用がし易いことが考えられる．

また，図 10(b) のプログラムはサイクロマチック数が 4 である．図 10(a) ほどサイクロマチック数は大きくはないが，解答プログラム全体の中では上位だった．`for` 文と `switch` 文が用いられており，自動修正適合性は jGenProg で 0.25, その他の APR ツールでは 0 であった．jGenProg の自動修正適合性が高かった理由としては，簡単な演算や分岐でできた文のため再利用性が高かったためだと考えられる．

図 11(a) および図 11(b) に ABC の 101 回 A 問題のステートメント数が小さいプログラムを示す．図 11(a) のプログラムはサイクロマチック数が 3 であり，図 11(b) のプログラムはサイクロマチック数が 1 であった．図 8(a) と図 8(b) 同様にラムダ式や三項演算子が用いられていることで，サイクロマチック数が小さくなっている．自動修正適合性は 4 つの APR ツール全てで 0 であった．

サイクロマチック数でもステートメント数と同様の傾向が得られた．分岐が多用されサイクロマチック数が増加したとしても APR ツールにとっては修正できることがわかった．簡単な演算や分岐でプログラムが構成されているほど修正しやすく，三項演算子やラムダ式が用いられた分岐がないプログラムほど修正しにくい．人にとってはサイクロマチック数の少ないプログラムのほうが可読性が高く良いと

[*4] https://pmd.github.io/

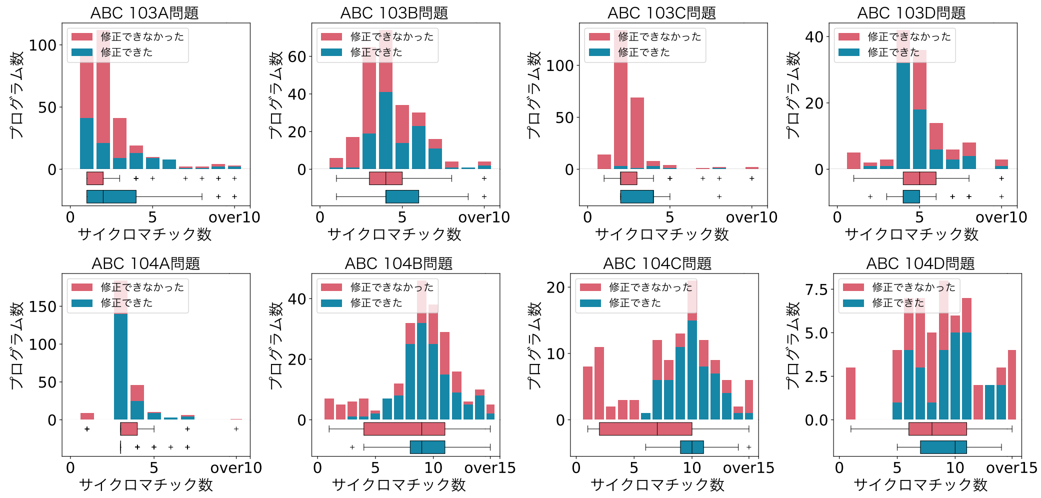


図 9: サイクロマチック数に対する修正できたプログラムの分布

されるが，APR ツールにとってはサイクロマチック数の多い方が修正しやすいことがわかった．

RQ6 への回答として，「**サイクロマチック数の多いプログラムの方が修正しやすい**」といえる．

(a) サイクロマチック数の多いプログラム

自動修正適合性(APRツール別)				サイクロマチック数
jGenProg	jMutRepair	Cardumen	jKali	
0.1	0.1	0	0.1	17

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String line = sc.nextLine();
        if(line.equals("----"))
        System.out.println(-4);
        if(line.equals("-+--")||line.equals("+---")||
                          line.equals("--+-")||line.equals("---+"))
        System.out.println(-2);
        if(line.equals("++--")||line.equals("+-+-")||
                          line.equals("+--+")||line.equals("-++-")||
                          line.equals("-+-+")||line.equals("--++"))
        System.out.println(0);
        if(line.equals("++++"))
        System.out.println(4);
        if(line.equals("+++-")||line.equals("++-+")||
                          line.equals("+-++")||line.equals("-+++"))
        System.out.println(2);
    }
}
```

(b) サイクロマチック数の多いプログラム

自動修正適合性(APRツール別)				サイクロマチック数
jGenProg	jMutRepair	Cardumen	jKali	
0.25	0	0	0	4

```
import java.util.Scanner;


public class Main {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);
        reader.useDelimiter("");
        int res = 0;
        for (int i = 0; i < 4; ++i) {
            switch (reader.next()) {
                case "+": {
                    res++;
                    break;
                }
                case "-": {
                    res--;
                    break;
                }
            }
        }
        System.out.println(res);
        reader.close();
    }
}
```

図 10: サイクロマチック数の多いプログラム

(a) サイクロマチック数の少ないプログラム

自動修正適合性(APRツール別)				サイクロマチック数
jGenProg	jMutRepair	Cardumen	jKali	
0	0	0	0	3

```
import java.util.Scanner;

public class Main {
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		char[] arr =sc.nextLine().toCharArray();
		sc.close();
		int count = 0;
		for(char c: arr) {
			count += c == '+' ? +1 : -1;
		}
		System.out.println(count);
	}
}
```

(b) サイクロマチック数の少ないプログラム

自動修正適合性(APRツール別)				サイクロマチック数
jGenProg	jMutRepair	Cardumen	jKali	
0	0	0	0	1

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        String s;
        try(Scanner cin = new Scanner(System.in)) {
            s = cin.next();
        }
        long count = s.chars().filter((c) -> c == '+').count();
        System.out.println(count - (4 - count));
    }
}
```

図 **11:** サイクロマチック数の少ないプログラム

RQ7：実行可能経路数が少ないほど修正しやすいか？

実行可能経路数の計測には，サイクロマチック数と同様に品質計測ツールの 1 つである PMD を用いた．図 12 に実行可能経路数に対する修正できなかったプログラムと修正できたプログラムの分布を示す．修正できなかったプログラムとは自動修正適合性が 0 のプログラムを指し，修正できたプログラムとは自動修正適合性が 0 より大きいプログラムを指す．図 12 の箱ひげ図の中央値に注目すると，修正できなかったプログラムに比べて修正できたプログラムは実行可能経路数が多い傾向にあることがわかる．

また，図 13 に実行可能経路数の多いプログラム例を載せる．RQ5, RQ6 で示した問題と同様の "+", "-" のシンボルの出現回数を計算するプログラムである．これは，`if` 文が並列しているため，実行可能経路数が多かった．自動修正適合性は，jGenProg で 0.07, jMutRepair で 0.29 であった．似た `if` 文の条件式と演算で記述されているため，文の再利用が高く jGenProg では自動修正適合性が高くなったと考えられる．また，jMutRepair の修正対象である条件式が用いられていることから，jMutRepair の自動修正適合性が高くなったと考えられる．

RQ7 への回答として，「**実行可能経路数の多いプログラムの方が修正しやすい**」といえる．

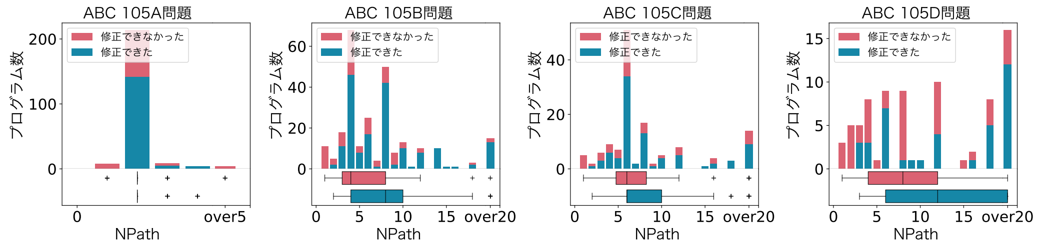


図 12: 実行可能経路数に対する修正できたプログラムの分布

図 13: 実行可能経路数の多いプログラム

自動修正適合性(APRツール別)				実行可能経路数
jGenProg	jMutRepair	Cardumen	jKali	
0.07	0.29	0	0	16

```
public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       String S = sc.next();
       int num = -4;
       if (S.charAt(0) == '+') num += 2;
       if (S.charAt(1) == '+') num += 2;
       if (S.charAt(2) == '+') num += 2;
       if (S.charAt(3) == '+') num += 2;
       System.out.println(num);
}
```

6 実験 2

6.1 実験概要

本実験では，オープンソースプロジェクトのから得られた同機能を持つメソッド群に対して自動修正適合性を計測する．同機能を持つメソッドの構造を比較することにより，どのような構造が修正しやすいか明らかにする．

6.2 実験対象プログラム

276 の同機能のメソッド集合，合計 728 のメソッドに対して自動修正適合性の計測を行う．

6.3 ミュータント生成

4 節で述べたミューテーション演算子と同様である．

6.4 APR ツール

4 節で述べた 4 つの APR ツールを使用する．

7 実験 2 結果

728 のメソッドのうち，ミュータントが生成できなかったもしくは生成されたミュータントがテストを追加してしまい自動修正適合性の計測ができなかったメソッドが 84 存在した．そのため，276 の等価機能メソッドグループのうち 39 グループが自動修正適合性の比較が行えなかった．残りの 237 の等価機能メソッドグループについて，各 APR ツールでプログラム構造による自動修正適合性に差が生じたグループ，差が生じなかったグループについて，表 9 に示す．26%∼41% のグループは構造によって自動修正適合性が異なることがわかった．

また，自動修正適合性が異なった構造について紹介する．図 14(a) から図 14(g) に機能が同一な 7 種の構造の異なるプログラムを示す．図の上部には 4 つの APR ツールによる自動修正適合性の計測結果を載せる．これらのプログラムは，2 つの String 型の引数を受け取り boolean 型を返すメソッドである．2 つの引数が同一なら true を返し，異なるなら false を返す．

同じ機能を持つプログラムでも 8 種の異なる構造の実装が存在し，自動修正適合性を計測したところ全て異なる自動修正適合性の傾向が得られた．

はじめに APR ツールに注目する．jGenProg は 8 プログラム中 1 プログラムが修正でき，jMutRepair は 7 プログラム修正でき，Cardumen は全て修正できず，jKali は 3 つ修正できたという結果になった．ABC の実験結果と同様に jMutRepair が修正できたプログラムが最も多い．これは，ABC と同様に `if` 文による条件式が多用されているプログラムのため，jMutRepair と相性が良かったと考えられる．対して，ABC とは異なり jKali が次に修正できたプログラムが多いという結果になった．これは `if` 文による判定が多用されたプログラム構造が多かったため，`return` 文を挿入する戦略が合致したためだと考えらえる．

次にプログラムの構造に注目する．似た構造を持っていたプログラムとして，図 14(a)，図 14(b)，図 14(c) に注目する．まず，図 14(a) と図 14(b) は条件式の中身は同じで，分岐の構造が異なっている．図 14(a) は `if` 文のみを使った構造に対して，図 14(b) は `else-if` 文と `else` 文を使用した構造

表 9: 自動修正適合性に差が生じたメソッドグループ数

APR ツール	自動修正適合性に差が生じたグループ数	自動修正適合性に差が生じなかったグループ数
jGenProg	61 (26%)	176 (74%)
jMutRepair	98 (41%)	139 (59%)
Cardumen	80 (34%)	157 (66%)
jKali	73 (31%)	164 (69%)

である．また，図 14(b) は else 節の中に return 文を記述している．図 14(a) と図 14(b) の自動修正適合性を比較すると，jMutRepair は図 14(a) の構造が自動修正適合性が高く，jKali は図 14(b) の構造の方が自動修正適合性が高くなった．jMutRepair は if 文を並列した構造の方が修正しやすく，jKali は if-else 文と else 文を用いた構造の方が修正しやすいとわかった．

次に，図 14(a) と図 14(c) の構造を比較する．これらは 2 箇所目の if 文の条件式の真偽が逆であり，return 文の内容が入れ替わっている．図 14(a) と図 14(c) の自動修正適合性を比較すると，jMutRepair は図 14(a) の方が自動修正適合性がわずかに高いという結果になった．複雑な return 文の内容ほどメソッドの終わりに記述されているほうが良いといえる．

次に，図 14(a) と図 14(d) の構造を比較する．図 14(a) の 2 箇所目の if 文と図 14(d) の 2, 3 箇所目の if 文は同じ処理を行っている．図 14(a) の条件を 2 行に分けて記述したものが 図 14(d) である．図 14(a) と図 14(d) の自動修正適合性を比較すると，jMutRepair は図 14(d) の方が自動修正適合性がわずかに高いという結果になり，jKali も図 14(d) の方が自動修正適合性が高いという結果になった．条件はできるだけ細かく分けた方が修正しやすいといえる．

次に， 図 14(e) と図 14(f) に似た構造を持っていたプログラムを示す．図 14(e) は else 文の中に return 文が記述されているのに対して，図 14(f) は else 文を用いていない．図 14(e) と図 14(f) の自動修正適合性を比較すると，jGenProg は図 14(e) の方が高く，jMutRepair はどちらも 1.0 であった．このことから，jMutRepair は不要な else 文の有無によって修正しやすさは変化しないが，jGenProg は else 文があるほうが修正しやすいといえる．

次に， 図 14(f) と図 14(g) に似た構造を持っていたプログラムを示す．これらは 2 箇所目の if 文の条件式の真偽が逆になっており，return 文の内容が入れ替わっている．図 14(f) と図 14(g) の自動修正適合性を比較すると，jKali は図 14(g) の方が高く，jMutRepair はどちらも 1.0 であった．このことから，jMutRepair は if 文の条件式の真偽を入れ替えても修正しやすさは変化しないことがわかる．

(a) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	0.57	0.00	0.00

```
boolean __target__(String s1,String s2){
      if (s1 == null && s2 == null) {
            return true;
      }
      if (s1 == null || s2 == null) {
            return false;
      }
      return s1.equals(s2);
}
```

(b) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	0.29	0.00	0.29

```
boolean __target__(String a,String b){
      if ((a == null) && (b == null)) {
            return true;
      }
      else   if ((a == null) || (b == null)) {
            return false;
      }
      else {
            return a.equals(b);
      }
}
```

(c) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	0.43	0.00	0.00

```
boolean __target__(String id1,String id2){
      if (id1 == null && id2 == null) {
            return true;
      }
      if (id1 != null && id2 != null) {
            return id1.equals(id2);
      }
      return false;
}
```

(d) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	0.71	0.00	0.86

```
boolean __target__(String a,String b){
      if (a == b) {
            return true;
      }
      if (a == null && b != null) {
            return false;
      }
      if (a != null && b == null) {
            return false;
      }
      return a.equals(b);
}
```

(e) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
1.00	1.00	0.00	0.00

```
boolean __target__(String a,String b){
      if (a == null) {
            return (b == null);
      }
      else {
            return (a.equals(b));
      }
}
```

(f) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	1.00	0.00	0.00

```
boolean __target__(String s1,String s2){
      if (s1 == null)   return (s2 == null);
      return s1.equals(s2);
}
```

(g) 機能等価メソッド

自動修正適合性(APRツール別)			
jGenProg	jMutRepair	Cardumen	jKali
0.00	1.00	0.00	0.50

```
boolean __target__(String a,String b){
      if (a != null) {
            return a.equals(b);
      }
      return a == b;
}
```

図 14: 機能等価メソッド

8　妥当性への脅威

本研究では 4 種類の限られた APR ツールに対してのみ実験を行っているため，特定のツールの戦略や特性に偏った結果が得られている可能性がある．異なる APR ツールを使用すると，本研究で得られた修正しやすいプログラム構造の結果とは別の構造の方が修正しやすいといった結果が得られる可能性がある．

また，本研究では実行時間の短縮のため APR ツールの実験設定である最大実行時間や個体数に制限を設けたため，設定を変更することで違った結果が得られる可能性がある．また，実験のミュータント数を制限するために ABC の解答プログラムにおいて，ミューテーション対象のメソッドを main メソッドと solve メソッドに限定した．他のメソッドに対するミュータントを生成していないため，ミューテーション対象をプログラム全体とした場合と結果が得られる可能性がある．

さらに，AtCoder Beginner Contest の解答プログラムに対する実験では ABC の各問題に対するテストケースが少ないため与えれられたテストケースのみ成功する解答が得られる可能性がある．オープンソースソフトウェアから収集されたプログラムに対する実験においても，テストケースが少ないため与えれられたテストケースのみ成功する解答が得られる可能性がある．

9　おわりに

本研究では，先行研究において提案された自動修正適合性について調査を行った．自動修正適合性とは APR 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指標である．自動修正適合性の利用により，開発者はこの数値が高いソフトウェアを開発できるようになり，その結果として APR 技術が多くのバグを修正できるようになる．

どのようなプログラム構造が APR にとって修正しやすいか調査を行うため，同機能を持つことなるプログラム構造を AtCoder の解答プログラムとオープンソースソフトウェアから収集した．また，SStuBs のバグパターンを利用し，ミューテーション演算子を先行研究よりも多く用いることで，より信頼性の高い数値を算出できるように実験を行った．

調査の結果，プログラム構造が異なれば自動修正適合性の値も異なることがわかり，ABC の問題難易度や APR ツールによっても自動修正適合性が変化することがわかった．ステートメント数，サイクロマチック数および実行可能経路数に対する自動修正適合性の傾向を調査したところ，単純な文の組み合わせで記述されたプログラムほど修正しやすいことがわかった．

今後の課題として，APR ツールの種類の増加が挙げられる．また，今回の調査ではオーバーフィットを考慮していないため，オーバーフィットを考慮するより厳しい指標の自動修正適合性の算出も挙げられる．

謝辞

本研究を行うにあたり，終始暖かく見守って下さった楠本 真二 教授に心より感謝いたします．

本研究の全過程において，始終熱心かつ丁寧なご指導をいただき，最後まで大きなお力添えいただきました肥後 芳樹 准教授に心より感謝申し上げます．

本研究に関して，発表資料や研究の議論の中で貴重なご助言をいただきました柗本 真佑 助教に心より感謝申し上げます．

本研究を進めるにあたり，多くのご助言をいただいた日立製作所 安田 和矢 様に感謝申し上げます．

3 年に渡って楽しい研究室生活を作って下さったので，楠本研究室で研究を続けることができました．大阪大学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 年の市川 直人 氏，同 出田 涼子 氏，同 荻野 翔 氏，同 藤本 章良 氏に心より感謝いたします．

研究生活を盛り上げていただきました，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 年の入山 優 氏，同 古藤 寛大 氏，同 髙市 陸 氏， 同 高木 一真 氏，同 谷口 真幸 氏，同 橋本 周 氏，同 渡辺 大登 氏に心より感謝いたします．

短い間でしたが，日々の研究室生活を豊かにしていただきました，大阪大学基礎工学部情報科学科 4 年の伊賀 彰 氏，同 岩瀬 匠 氏，同 小田 郁弥 氏，同 竹重 拓輝 氏，同 長谷川 和輝 氏，同 吉岡 遼 氏に心より感謝いたします．

最後に，本研究に至るまでに，講義や演習実験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，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] Ren é Just, Darioush Jalali, and Michael D. Ernst. Defects4J : A database of existing faults to enable controlled testing studies for java programs. In *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[5] K. Liu , S. Wang , A. 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 *Proc. International Conference on Software Engineering*, p. 615–627, 2020.

[6] 九間哲士, 肥後芳樹, まつ本真佑, 楠本真二, 安田和矢. 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. ソフトウェアエンジニアリングシンポジウム, 第 2021 巻, pp. 51–58, 2021.

[7] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Bö hme, and Abhik Roychoudhury. A Correlation Study between Automated Program Repair and Test-Suite Metrics. In *Proc. International Conference on Software Engineering*, p. 24, 2018.

[8] 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 *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302–313, 2019.

[9] Fowler Martin. Refactoring: Improving the Design of Existing Code. 1999.

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proc. International Symposium on Software Testing and Analysis*, p. 449–452, 2016.

[11] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proc. International Symposium on Foundations of Software Engineering*, pp. 52–63, 2014.

[12] R. M. Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proc. International Conference on Mining Software Repositories*,

p. 573–577, 2020.

[13] Raymond P.L. Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, Vol. 36, No. 4, pp. 546–558, 2010.

[14] McCabe. A Complexity Measure. *Proc. Transactions on Software Engineering*, Vol. 2, pp. 308–320, 1976.

[15] Michele Lanza and Radu Marinescu. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. 2006.

[16] Brian A. Nejmeh. NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM*, Vol. 31, No. 2, p. 188–200, 1988.

[17] Y. Yuan and W. Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067, 2020.

[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2011.

[19] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Proc. International Conference on Software Engineering*, p. 492–495, 2014.

[20] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proc. International Symposium on Software Testing and Analysis*, pp. 441–444, 2016.

[21] Vidroha Debroy and W. Eric Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proc. International Conference on Software Testing, Verification and Validation*, pp. 65–74, 2010.

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

[23] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proc. International Symposium on Software Testing and Analysis*, ISSTA 2015, p. 24–36, 2015.

[24] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. In *Proc. International Symposium on Software Testing and Analysis*, p. 191–201, 2013.

[25] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: an eclipse plug-in for testing and debugging. In *Proc. International Conference on Automated Software Engineering*, pp. 378–381, 2012.