社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

自動修正適合性を用いた修正しやすいプログラム構造の評価

前島　葵† 肥後　芳樹† 藤本　章良† 柗本　真佑†
楠本　真二† 安田　和矢††

E-mail: †{a-maejim,higo,a-fujimt,shinsuke,kusumoto}@ist.osaka-u.ac.jp, ††kazuya.yasuda.fd@hitachi.com

あらまし　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実験を行う．

キーワード　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

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]．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．APR とは，バグを含むプログラムから自動的にバグを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 [2]，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 [3]．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 [4] で提案された．先行研究によって，自動修正適合性というソフトウェア品質指標が提案された．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を分析が行われた．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変化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，リファクタリングの違いが 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．

2. 自動修正適合性

本章では先行研究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

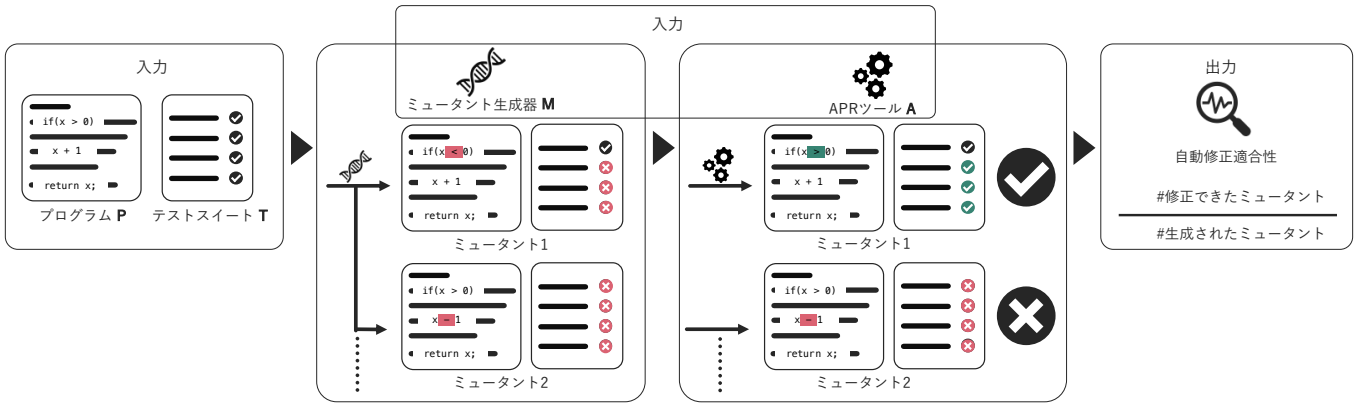

図 1　自動修正適合性の計測手法の概要

2.1　計 測 手 法

図 1 に先行研究 [4] 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 [5]．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 [6]．

提案手法は以下の 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，そのミュータントは Step3. では用いない．

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．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， 自動修正適合性は 0.7（= 7/10）となる．

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の変化が見られた．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

本章では，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「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 6 種のリファクタリングが対象とされた．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 行から 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 [7] のミューテーション演算子を利用した(注1)．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．

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．

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

(注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

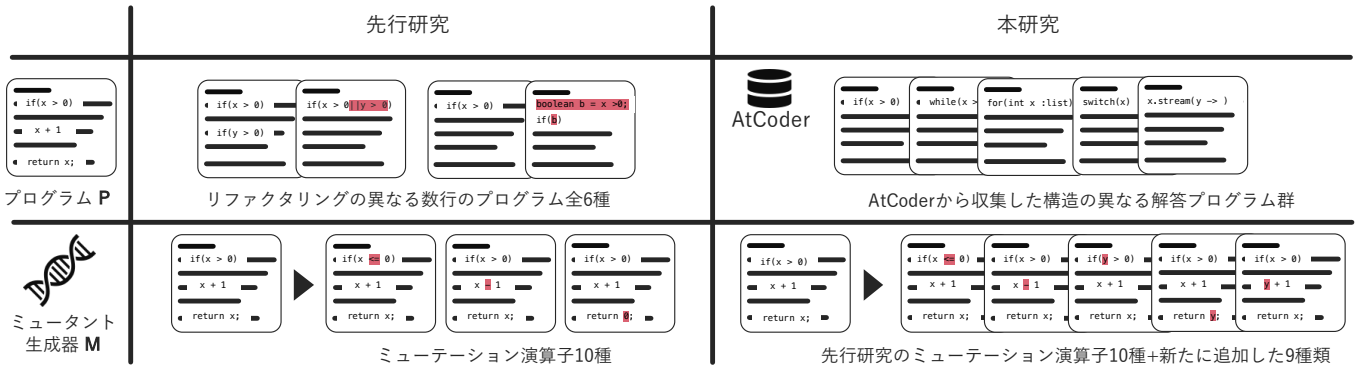

図 2　先行研究と本研究の計測方法の比較

4 つの問題で構成されている(注2)．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 から同じ問題を解くための様々な構造を持ったプログラム群を収集した．

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) [8] 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．先行研究で調査されたミューテーション演算子に含まれるバグパターンや，Java の例外処理に関するバグパターンを除いて表 2 に示す 9 種類のミューテーション演算子を実装した．

4.　実験の結果と考察

4.1　実 験 設 定

実験対象プログラム

表 3 に示す ABC の 101 回，102 回の A, B 問題の合計 4 題を調査対象とした．表 3 に各問題の概要を示す．

(注2)：2019 年 4 月 27 日開催の ABC125 以前は 4 題構成

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`

ミュータント生成

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ため作成することの多いメソッドである．

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 [9]　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．

jMutRepair [10]　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 [11]　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．

実験設定は 3 つの APR ツール全て共通し，世代数は 10 であり，各世代の個体数は 10 である．

4.2　本研究における計測の変更点の評価

本章では，本研究の提案手法である実験対象プログラムの増加数と実験対象ミュータント演算子の増加数の評価を行う．

実験対象プログラムの増加数

表 4 に ABC の各問題の生成されたミュータント数と修正できたミュータント数を示す．280 個から 300 個の解答プログラムを収集し，1,474 個から 2,481 個のミュータントが得られ

表 2　本研究で追加し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
Change Identifier Used	`Superclass a`	`Subclass a`
Change Numeric Literal	`if(x<0)`	`if(x<1)`
Wrong Function Name	`getEdgeCount()`	`getNodeCount()`
Same Function Less Args	`method(a,b,c)`	`method(a,b)`
Same Function Change Caller	`a.method()`	`b.method()`
Same Function Swap Args	`method(a,b)`	`method(b,a)`
Change Operand	`a+b`	`a+c`
More Specific If	`if(a&&b)`	`if(a)`
Less Specific If	`if(a||b)`	`if(a)`

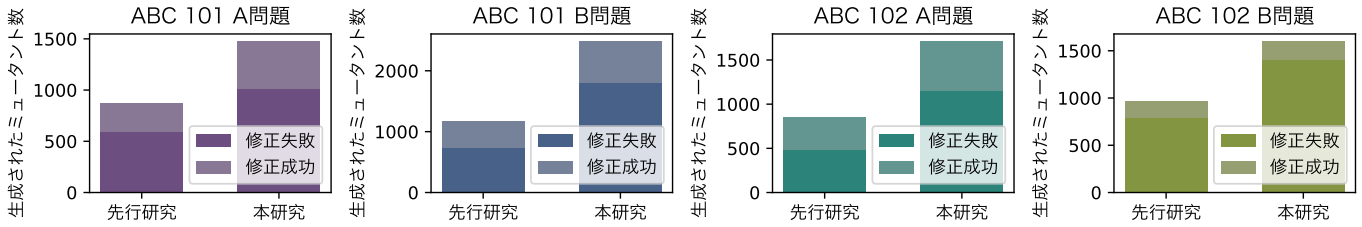

図 3　先行研究と本研究のミュータントの生成数の比較

た．また修正できたプログラムも 15% から 33% 程度得られ，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十分な比較が行えると考えた．

ミュータントの増加数

図 3 に先行研究と本研究で生成されたミュータントおよび修正できたミュータント数を示す．図 3 の横列は調査対象 ABC の各問題を示す．ABC の 4 題全体を見ると，先行研究で用いられたミュータント演算子から生成されたミュータントは 2,613 個であった．本研究の手法で新たに 2,759 個のミュータントが追加され，合わせて 5,372 個となり 51 %増加できた．また，修正できたミュータント数は先行研究の手法では 2,613 個のうち 1,247(48%) であり，本研究の手法で追加された 2,759 個のうち 613(23%) であった．

次に，図 4 にミューテーション演算子毎の先行研究と本研究で生成されたミュータントおよび修正できたミュータント数を示す．本研究で導入したミュータントのなかでは，数値を書き換える ChangeNumericLiteral が大部分を占める 2,844 個存在した．その中で 22% の 614 個が修正できた．

本研究で導入したミュータント演算子や先行研究で用いられていたミュータント演算子の中にはミュータントを生成できなかった演算子が存在した．生成できなかった演算子には，メソッドを呼び出し文を書き換えるミューテーション演算子がある．これは ABC の比較的単純なプログラムを対象にしているため，他のメソッドに依存せず単一のメソッド内で処理を記述できてしまうことが要因であった．また，ミューテーション対象のメソッドを `main` メソッドおよび `solve` メソッドと限定し，これらは `void` 型であったため，`return` 文を書き換えるミューテーション演算子もミュータントを生成できなかった．

ミュータントは生成できたが 3 つの APR ツールのどれでも修正ができなかったミュータントは，VoidMethodCalls, ChangeBooleanLiteral, MoreSpecificIf, WrongFunctionName の 4 種であった．メソッド呼び出しや，`return` 文に関するミュータントは修正が容易でないことがわかる．

表 4　生成されたミュータント数と修正できたミュータント数

問題	プログラム数	生成されたミュータント数	修正できたミュータント数 (%)
ABC101 A	300	1,474	463 (32%)
ABC101 B	300	2,481	676 (29%)
ABC102 A	280	1,709	555 (33%)
ABC102 B	280	1,598	196 (15%)

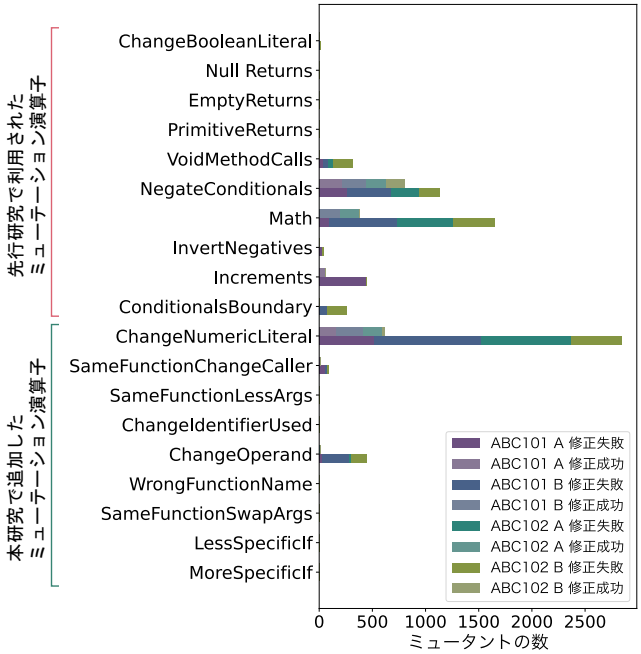

図 4　ミューテーション演算子毎のミュータント修正数

4.3　結果と考察

ミュータント毎の修正率，APR ツールの比較，問題毎の比較，ステートメント数と修正率，サイクロマチック数と修正率についてそれぞれ述べる．

ミュータント毎の修正率

図 4 から最も修正率の高いミュータントは 71% が修正できた NegateConditionals であるとわかる．条件式を書き換えるミュータントのため，バグ箇所の特定が比較的容易である可能性が高い．したがって多く修正できたと考えた．また，ChangeNumericLiteral は 22% が修正でき Math は 23% が修正できたことから，数式や数値に関するバグは APR ツールが他のバグに比べて得意だと言える．

APR ツールの比較

3 種の APR ツールを比較すると jMutRepair が修正できたプログラム数が最も多かった．これは，jMutRepair は条件式あるいは `return` 文のミューテーションにより修正する戦略のためである．題材とした ABC の解答に高い頻度で `if` 文が用いられるため条件式の論理演算子を書き換えるミュータントを多

表 3　実験対象プログラム

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

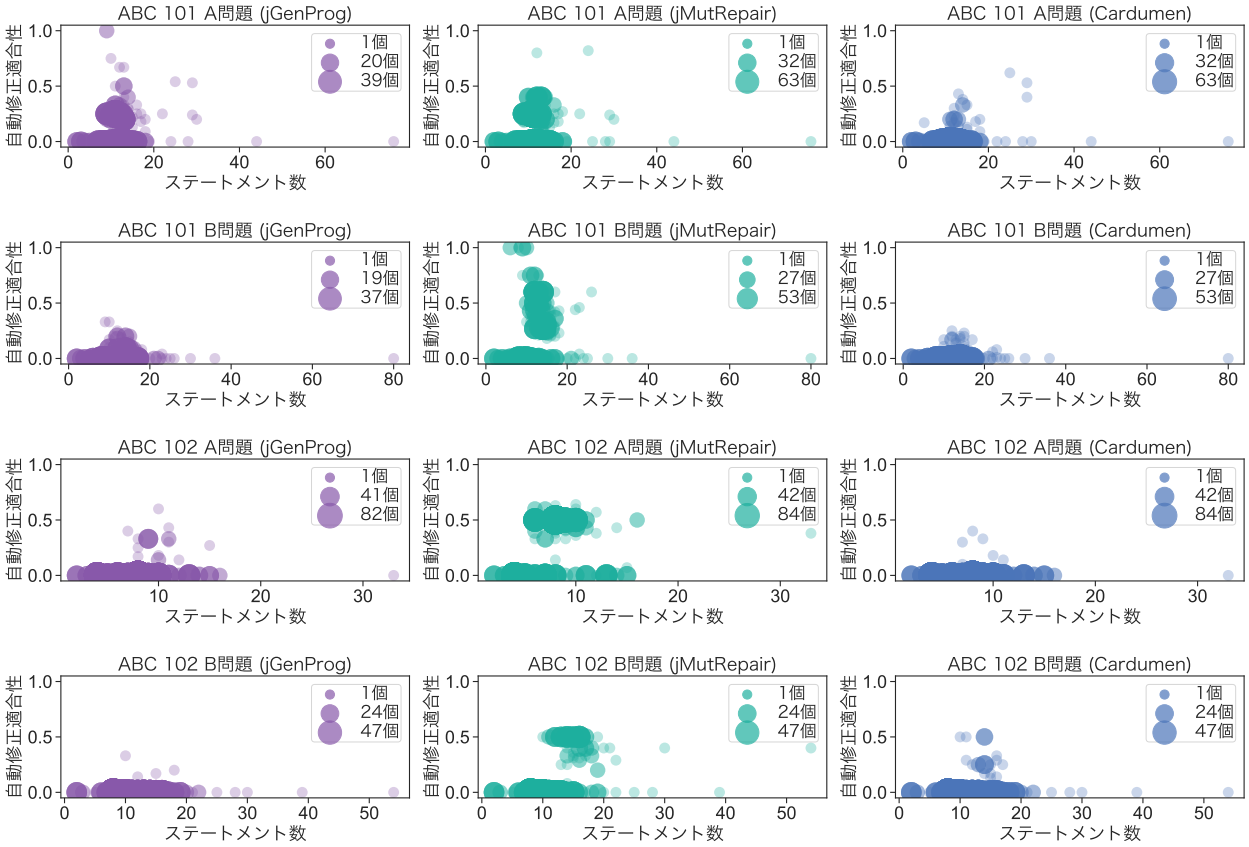

図 5　ステートメント数に対する自動修正適合性

く生成し，jMutRepair の戦略に合致するミュータントが多く存在した．対して，ABC の問題は数行に渡るような複雑な計算を必要とせず演算処理を行う文が少ないため，文を再利用した書き換えを戦略とする jGenProg や Cardumen は jMutRepair に比べて修正できるミュータントが少なかった．

問題毎の比較

表 4 や，図 5 および図 6 から ABC102 B 問題は他の問題に比べ，修正できたプログラムの数が少ないことがわかる．他の問題は入力として 1 つの整数や 4 字程度の文字列が与えられるのに対し，ABC102 B 問題は配列が入力として与えられる．配列の各要素へのアクセスや `for` 文によるループなどの処理が必要となり，他の問題よりプログラム構造が複雑になっていた．そのため修正率が低くなっていると考えた．

ステートメント数と修正率

図 5 および図 6 に調査結果を示す．各グラフの横軸は，図 5 はステートメント数であり，図 6 はサイクロマチック数である．縦軸は修正できたミュータント数の割合である自動修正適合性を示している．グラフの各点の大きさと濃さは，その点を満たすプログラムの数から対数をとった値で表現されている．サイズが大きく色の濃い点はその点を満たすプログラムが多く存在し，対してサイズの小さく色の薄い点はその点を満たすプログラムは少ないことを示している．全体の縦列には APR ツールである jGenProg, jMutRepair, Cardumen の結果を，横列には ABC の各問題についての結果を示している．

図 5 から三項演算子を用いて必要な処理を 1 行に収めて記述するような 1 つのステートメントのみからなるプログラムは修正できないとわかる．このことから，ステートメントが複雑なプログラムは自動修正適合性が低く，変数の宣言文や代入文，`if` 文が含まれているほうが修正しやすいことがわかる．また，ABC の A 問題の 20 から 40 のステートメント数を持つプログラムは，修正できなかったミュータント数に対して修正できたミュータント数が多かった．このことから，同じ処理でも複数の文に処理を分解し，ステートメント数を増やすことで自動修正適合性が向上する可能性があると考えた．

サイクロマチック数と修正率

最後に，図 6 のサイクロマチック数については，サイクロマチック数が小さくプログラムが複雑でないほど修正できるミュータントが多い傾向があるとわかった．

5. 妥当性への脅威

本論文の段階では，ABC の限られた問題にしか調査が行えていないため，ABC の各問題の特性がプログラムに現れ，生成されるミュータントや修正可否に対して大きく影響している可能性がある．また，3 種類の限られた APR ツールに対してのみしか実験を行っていないため，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．また，実験において個体を変異させる最大世代数や個体数に制限を設けたため，制限を変更することで違った結果が得られる可能性がある．さらに，ABC の各問題に対するテストケースが少ないため，与えられたテストケースのみ成功する回答が得られる可能性がある．

6. 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

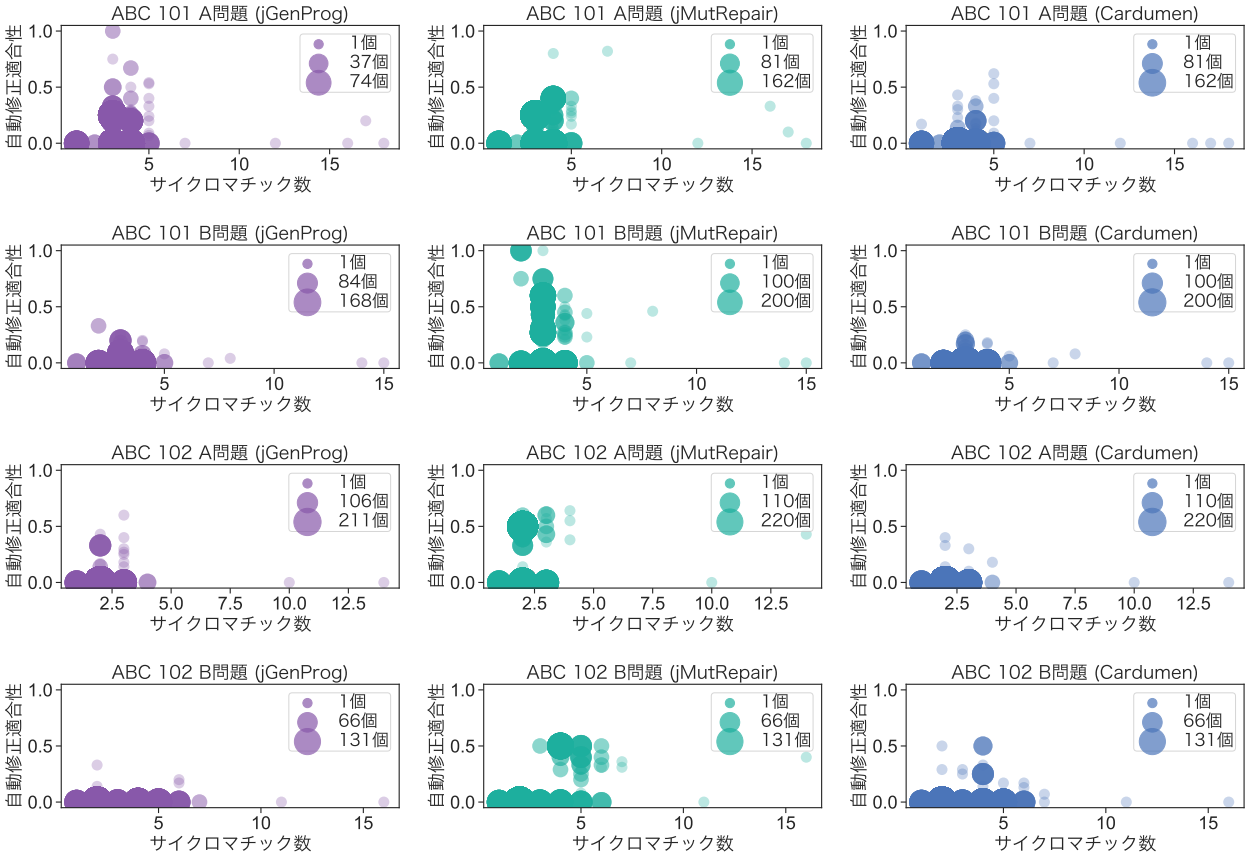

図 6　サイクロマチック数に対する自動修正適合性

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，AtCoder の解答プログラム群を利用し同じ機能を持つ，異なる構造のプログラムを実験対象とした．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，ABC の各問題で用いられたプログラム構造による自動修正適合性の差が存在し，APR ツールの得意不得意が顕著であることがわかった．

今後の調査では，実験対象プログラムを増やした分析や APR ツールの種類の増加を行う予定である．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行う予定である．

謝辞　本研究は一部 JSPS 科研費 JP21K18302 の助成を得た．

文　献

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, no.1, pp.4–12, 2002.

[2] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” IEEE Transactions on Software Engineering, vol.45, no.1, pp.34–67, 2019.

[3] K. Liu , S. Wang , A. Koyuncu , K. Kim, T.e.F. Bissyand´ e , D. Kim, P. Wu, J. Klein, X. Mao, and Y.L. Traon, “On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs,” Proc. International Conference on Software Engineering, p.615–627, 2020.

[4] 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 2021 論文集，第 2021 巻，pp.51–58，2021.

[5] J. Yi, S.H. Tan, S. Mechtaev, M. Bö hme, and A. Roychoudhury, “A correlation study between automated program repair and test-suite metrics,” Proc. International Conference on Software Engineering, p.24, 2018.

[6] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, “Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts,” Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.302–313, 2019.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “Pit: A practical mutation testing tool for java (demo),” Proc. International Symposium on Software Testing and Analysis, p.449–452, 2016.

[8] R.M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the manysstubs4j dataset,” Proc. International Conference on Mining Software Repositories, p.573–577, 2020.

[9] 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.

[10] M. Martinez and M. Monperrus, “Astor: A program repair library for java,” Proc. International Symposium on Software Testing and Analysis, pp.441–444, 2016.

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