特別研究報告

題目

探索的ソースコード変換による自動修正コードの不要処理の除去

指導教員

楠本 真二 教授

報告者

岩瀬 匠

令和 4 年 2 月 8 日

大阪大学 基礎工学部 情報科学科

令和 3 年度 特別研究報告

探索的ソースコード変換による自動修正コードの不要処理の除去

岩瀬 匠

内容梗概

デバッグの全自動化を目的として，探索に基づく自動プログラム修正手法（Automated Program Repair; APR）が多数提案されている．探索ベースの APR ではソースコードの意味を考慮せずに探索するという性質上，修正されたソースコードは人の記述するソースコードとはかけ離れた内容となりやすい．ソースコードの最適化を目的とした，デッドコードの除去と呼ばれる研究も存在するが，APR の生成したソースコードの不要処理の除去という目的に対しては機能的に不十分である．本研究では APR が生成したソースコードの不要処理の除去を目的として，探索的なソースコード変換手法を提案する．提案手法では，遺伝的アルゴリズムに基づき，部分的な変換を繰り返し適用することでソースコードを自然な形へと少しずつ変換する．本稿では，分岐，及び整数値の加減算というプログラムにおける基本処理に限定し，提案する探索的変換手法の実現可否を確かめる．数行程度の小規模題材に対し予備実験を行った結果，全てのソースコードについて不要な部分が除去され，人の記述する自然なソースコードへと変換できることを確認した．

主な用語

自動プラグラム修正，APR，ソースコード変換，デッドコード，リファクタリング，可読性

目次

図目次

表目次

1　はじめに

自動プログラム修正（Automated Program Repair; APR）と呼ばれる，ソースコード中に含まれるバグを全自動で取り除く技術の研究が進められている [1]．APR は探索ベースの手法 [2] と意味論ベースの手法 [3] の 2 種類に大別できるが，本研究では探索ベースの手法のみに着目し，以降単に APR と呼称する．この APR ではバグを含むソースコードと対応するテストスイートを入力とし，ソースコードへの小さな改変を繰り返すことで，バグを含まない状態へと探索的に近づける．探索のメタアルゴリズムとしては遺伝的アルゴリズムが用いられることが多い．意味論ベースの手法が条件分岐のバグ [4] や同時並行性のバグ [5] などの特定のバグ種別に限定している場合が多い一方で，探索ベースの手法は理論上どのような種類のバグでも修正可能，という汎用性の面での利点が存在する．

本研究では APR が抱える課題の 1 つである，修正済みソースコードの不要処理の多さに着目する．探索ベースの APR では，修正対象となるソースコードの意味的な振る舞いを考慮せず，あらかじめ定められた改変処理を繰り返し適用する．具体的な改変処理としては，AST 単位での挿入/削除/再利用 [2] や，メソッド呼び出しの挿入/削除 [6]，変数名や演算子の変更 [7] などが用いられる．いずれの改変処理を用いても，ソースコードの意味を考慮せずに探索するという性質上，修正されたソースコードは人の記述するソースコードとはかけ離れた内容となりやすい．例えば，AST 単位での再利用の繰り返しにより，「`n++; n--;`」という互いに打ち消し合う不要な処理や，「`n++; n=10;`」のような片方が上書きされる不要な処理が発生することがある．また削除処理を繰り返した結果，空のブロック宣言「`{}`」だけが残される，といったケースも存在する．探索が深くなるほど適用される改変回数が増える．よって，修正が困難なバグの場合，APR により得られたソースコードの大半が不要な処理で構成されている，というケースも考えられる．

この不要処理の多さは，APR が抱える重要課題の 1 つであるテストへの過剰適合 [8] の解決を困難にする．多くの APR では，その成否の基準としてテストスイートの通過の可否が用いられる．そのため，改変されたソースコードが与えられた全テストには通過するが，意味的には正しくないという過剰適合の問題が避けられない．この過剰適合は特にテストスイートが不十分な場合に発生しやすい [9] ことが知られており，APR 実用に対する大きな障壁となっている．過剰適合の自動判別手法 [10] も提案されているが，得られたソースコードの最終的な意味的な確認，及び受理するか否かの意思決定は開発者に委ねる必要がある．しかし先述の通り，APR が修正したソースコードには不要処理が多く，可読性が極めて低い．これは開発者による過剰適合の目視確認作業を妨げる要因となる．

本研究では APR が生成したソースコードの不要処理の除去を目的として，探索的なソースコード変換手法を提案する．本手法では，ソースコード中に含まれる実行可能な文のうち，外的な振る舞いに影響を与えない処理を不要な処理と定義し，除去の対象とする．本手法の実現により，ソースコード中の

バグを機械が修正し，さらに機械が自然な形に変換するという利用が可能となる．提案手法では入力となるソースコードに対し，事前に用意した変換ルールを適用し，その効果を行数等のメトリクスにより評価する．この流れを遺伝的アルゴリズムに基づいて繰り返すことで，APR の生成した極めて不自然なソースコードを少しずつ自然な形に近づけていく．適用する変換ルールを外的な振る舞いに影響しない処理に限定することで，変換前後での振る舞いの等価性を保つ．

本稿では，分岐，及び整数値の加減算というプログラムにおける基本処理に限定し，提案する探索的変換手法の実現可否を確かめる．実験では，まず数行程度の小規模題材に対し，ミューテーション解析を用いて人為的にバグを注入する．次に，このバグを含むソースコードを APR により修正する．最後に，得られた不要な処理を含む修正済みソースコードを提案手法により変換し，その効果を確かめる．

2 準備

2.1 APR

探索ベースの APR には，遺伝的アルゴリズムを利用する手法 [11] と変更パターンを利用した手法 [12] の 2 つが存在する．本節では，APR 分野のブレイクスルーとなった GenProg[2] が使用する，遺伝的アルゴリズムを用いた手法について説明する．

GenProg の入力は修正対象のソースコードとテストスイートであり，出力はテストを全て通過するソースコードである．APR による修正は個体の生成，評価，選択の 3 つの処理を繰り返し行う．個体の生成には変異と交叉の 2 つの方法があり，いずれかを用いてソースコードに改変を加える．次に，生成した個体がどの程度バグ修正に近づいたのか評価を行う．GenProg ではテスト通過数を用いている．最後に，この評価値に基づいて次の世代に残す個体を選択する．この一連の処理を繰り返し，入力されたテストを全て通過する個体を規定数以上生成すると，遺伝的アルゴリズムを終了する．

APR ではソースコードへの改変を繰り返し，全てのテストを通過する，機能的に正しいソースコードを生成する．この時，変異，交叉，いずれの方法を用いても，ソースコードの意味を考慮せずにソースコードを改変する．ゆえに，加えられた改変の意味の理解は難しい [10]．そのため，人の理解しやすいソースコードに変換する必要がある．

2.2 不要処理の除去を目的としたソースコード変換

ソースコードの外的な振る舞いを保ちつつ，不要処理を除去するソースコード変換のアイデア [13][14] がこれまでに多数提案されてきた．最も広く知られた手法として，Fowler による不吉な臭いとそのリファクタリング手法 [13] が挙げられる．Fowler のリファクタリング手法では，一時変数や分岐などのメソッドを構成する文単位での改善方法から，メソッド単位，クラス設計単位などの手法が幅広く定義されている．リファクタリング可能箇所の自動検出を目的とした支援ツール [15] も数多く提案されており，開発者は手軽にリファクタリングを実施できる．

しかし，APR の生成したソースコードへのリファクタリング手法の適用には限界がある．その理由の 1 つは，リファクタリング手法の多くが人の記述したソースコードを前提としている点にある．リファクタリングの前提となる不吉な臭いは，開発者が陥りがちな悩みや間違いを基準としている．他方，APR の生成するソースコードは，人がまず記述しないような極めて不自然な部分を含む．例えば，「`n++; n--;`」のような不要であることが誰の目から見ても明白，かつ開発者が記述しないような処理はリファクタリング手法の対象外となる．不吉な臭いは開発者がある程度の意思を持って記述した「高水準」な改善可能箇所であるのに対して，APR の生成したコードに含まれる不要な処理は意思のない「低水準」であると見なすこともできる．

このような「低水準」な不要処理を表す概念として，デッドコード [14] が広く知られている．しかしこのデッドコードの除去も，APR の生成したソースコードの不要処理を除去するという目的に対しては機能的に不十分である．様々なデッドコードの定義が存在するが，未使用や到達不能といった，実行経路に存在しない文やメソッドなどが共通してデッドと定義される [14]．よって先述の「`n++; n--;`」のような互いに打ち消し合う不要な処理は除去の対象外となる．

ソースコードの自動変換という観点では，IDE 等に組み込まれたフォーマッタが最も広く利用されている技術であるといえる．このフォーマッタは，改行や空白文字などのトークン区切り文字を対象として，自動的にソースコードを見やすい形に整える．これらトークン区切り文字をどこに挿入すべきか，という問題は一般的にはコーディングスタイルと呼ばれる．他方，本稿で変換対象となる不要な処理とは，実行可能な文のうち機能的には不要という部分であり，コーディングスタイルに基づいた変換とはその変換の観点が異なる．

2.3 Motivating Example

本研究で対象とするソースコードの変換について，具体例を用いて説明する．図 1 に実際の APR により得られた実プロジェクトのバグの修正パッチを示す．このパッチは複数 APR の実験データセット，RepairThemAll[16] から取得したパッチである．より具体的には，ARJA[17] と呼ばれる探索ベース APR が生成した，Apache Math プロジェクトで発生したガンマ式に対するバグ [18] の修正パッチである．本パッチは 2 つの diff ブロック（3 行目から 28 行目と，29 行目から 34 行目）で構成される．

前半の diff ブロックは 1 行の削除と 19 行の追加行で構成されるが，パッチの適用前後で機能的な振る舞いの変化は全くなく，全て不要な改変である．この改変箇所は，メソッドの返り値となる一時変数 `ret` を算出する処理に該当する．まず，6 行目の if 式と追加する 8 行目の if 式が全く同じであるため，8 行目と 9 行目，及び if ブロック外の 27 行目のみが実行経路上にあり，それ以外の 10 行目から 26 行目は完全なデッドコードとなる．また if ブロック外の 27 行目の存在により 8 行目と 9 行目は完全に不要となり，さらに追加する 27 行目と削除した 7 行目は完全に一致する．結果的に，バグを修正，つまり全テストを通過するに至った有効な改変は，2 つ目の diff ブロックのみとなる．

APR により図 1 のソースコードを得られた場合，上記のようなソースコード読解を行った上で，どの改変がテスト通過に寄与しているのか，その改変は意味的に正しいか，過剰適合ではないかを確認する必要がある．しかし，本図のように APR の生成コードのほとんどが不要な改変で構成されると，その可読性は低く意味的な確認も困難になる．加えて，多くの言語では return 節や throw 節などのある種のジャンプ命令を含んでおり，目視確認のコストを増加させる要因となる．例えば本図の場合，22 行目に例外を送出する処理が追加されており，この処理が実行経路に存在するかを確認する必要がある．また，メソッド呼び出しによる作用の有無も考慮に入れる必要がある．上記のような APR の生成パッ

```
--- /tmp/Arja_Defects4J_Math_103/src/../Gamma.java
+++ /tmp/Arja_Defects4J_Math_103/src/../Gamma.java
@@ -158,7 +158,26 @@
  if (Double.isNaN(a) || Double.isNaN(x) || (a <= ..
    ret = Double.NaN;
  } else if (x == 0.0) {
-   ret = 0.0;
+   if (x == 0.0) {
+     ret = 0.0;
+   } else if (a >= 1.0 && x > a) {
+     ret = 1.0 - regularizedGammaQ(a, x, epsilon ..
+   } else {
+     double n = 0.0;
+     double an = 1.0 / a;
+     double sum = an;
+     while (Math.abs(an) > epsilon && n < maxIte ..
+       n = n + 1.0;
+       an = an * (x / (a + n));
+       sum = sum + an;
+     }
+     if (n >= maxIterations) {
+       throw new MaxIterationsExceededException( ..
+     } else {
+       ret = Math.exp(-x + (a * Math.log(x)) - l ..
+     }
+   }
+   ret = 0.0;
  } else if (a >= 1.0 && x > a) {
@@ -177,7 +196,7 @@
  }
  if (n >= maxIterations) {
-   throw new MaxIterationsExceededException(maxI ..
+   ret = 1.0;
  } else {
```

図 1: 探索ベース APR で得られた実プロジェクトバグの修正パッチ．diff の前半部分は 1 行の削除と 19 行の追加で構成されるが，パッチの適用前後で機能的な振る舞いの変化は全くなく，全て不要な改変である．

チの理解を支援するためには，APR の生成したソースコードを前提とした，ソースコードの変換手法が必要であるといえる．

3 提案手法

3.1 手法の概要

提案手法において，ソースコード内に存在する外的な振る舞いに影響を与えない文を不要な文と定義する．また，変換とは，この不要な文を除去する処理である．図 2 に提案する探索的変換手法の概要を示す．入力は変換対象となる APR が生成したソースコード，出力は入力と機能的に等価で，不要処理が除去されたソースコードである．提案手法では遺伝的アルゴリズムを用いて，変換，評価，選択の 3 つの処理を繰り返す．この 3 つの処理を合わせて 1 世代と呼称する．まず変換処理では，事前に用意された複数の変換ルールからランダムに 1 つ選び，ソースコードを部分的に変換する．この時，変換対象のソースコード 1 つに対して，使用するルールの決定を複数回行い，部分的な変換が行われたソースコードを複数用意する．次に，部分的な変換が行われたソースコードをそれぞれ評価する．評価値として，ソースコードの巡回的複雑度や行数などのメトリクスを用いる方法が考えられる．最後に，選択処理ではそれぞれの評価値をもとに次世代の変換処理の対象となるソースコードを選択する．以上を繰り返し，評価値が改善しないまま世代数だけが増える等，これ以上の変換処理が不可能となった時，最も評価値の良いソースコードを出力する．

提案手法の特徴としては変換ルールが追加可能である点と，遺伝的アルゴリズムによる探索的なソースコード変換があげられる．変換ルールの事前定義不足により，変換ができなかったソースコードが存在した場合，適切なルールの追加により変換が可能になる．また，探索的な実行により，自然な変換が可能となる．詳細は 3.3 節で述べるが，ルールの適用順序を事前に定義するような，非探索的な変換では，ソースコード内に不要な処理が残る場合が存在する．

3.2 変換ルール

提案手法は，外的な振る舞いを保ちつつ，ソースコードに変更を加える．変換前後での振る舞いの保持として，変換後のソースコードに対してテスト実行をする方法と，変換ルール自体が振る舞いを変化させないことを保証する方法がある．テスト実行をする場合，提案手法における変換対象は APR が修正したソースコードであるため，APR の入力となったテストスイートを用いる．この方法の長所は極めて柔軟にコードの書き換えが可能な点である．短所としては毎回テストを実行する必要があり所要時間が長くなる点である．変換ルール自体が機能の保持を満たす場合，機能を変化させない単純なルールに限定して変換する．長所は機能保持の確認が不要なので，前者と比べて，所要時間が短くなる点である．短所としては変換可能なソースコードの対象が限定されてしまう点である．提案手法では後者のみを用いる．この理由としては，一回の探索に時間を要して柔軟にコードを書き換えるよりも，単純なルールを用いて，一回辺りの変換にかかる時間を短くし，探索の回数を増やす方が効果的だと考えたか

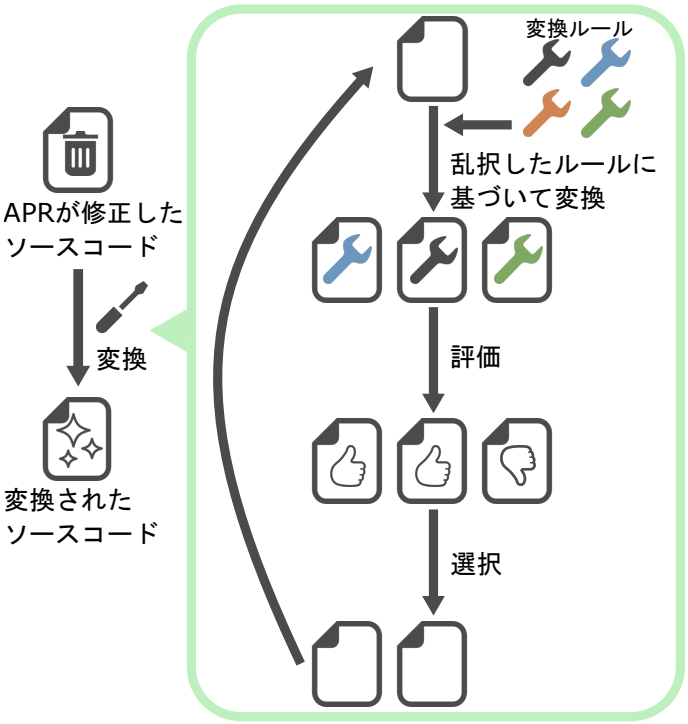

図 2: 提案する探索的変換手法の概要

らである．変換可能な対象が限定されるという短所については，プラグイン形式でのルール追加により変換対象を拡充する．

実用的な変換のためには変換ルールの充足が必要不可欠である．しかし，本稿では提案手法の実現可否を目的としている．そのため，分岐と整数の加減算を扱うプログラムに頻出する，機能的に意味を持たない不要な箇所に限定した．表 1 に本稿で使用する変換ルール一覧を示す．これらの変換ルールは単純なものであり，変換前後で機能の保持が行われている．

3.3 探索的ソースコード変換

変換ルールの適用方法には，事前にルールの適用順序を決める方法と，探索的に適用する方法がある．提案手法では，変換ルールを探索的に適用する方法を選択する．これは，事前に適用順序を決めて変換を行うと，自然な変換ができない場合が生じるからである．例えば，あらかじめ変換ルールの適用順序を空ブロックの削除，打消しあう単項演算子の削除という順番に決めていた場合を考える．与えられた int 型の引数に 1 を加算するプログラムにおいて，バグが含まれているソースコードを修正し，図 3(a) が得られたとする．この時，ソースコードは図 3(b) を経て，図 3(c) のように変換される．一方，修正後のソースコードとして図 4(a) が得られた場合，このソースコードには空ブロックが存在しないので，図 4(b) を経て，図 4(c) のように変換される．図 3(c) では変換後のソースコードに不要な処理は存在

```
1 void plus(int n) {
2   n++;
3   n++;
4   {
5   }
6   n--;
7   return n;
8 }
```

(a) APR の修正ソースコード 1

```
1 void plus(int n) {
2   n++;
3   n++;
4   n--;
5   return n;
6 }
```

(b) 空ブロックの削除後

```
1 void plus(int n) {
2   n++;
3   return n;
4 }
```

(c) 打消しあう単項演算子の削除後

図 3: 不要な処理が残らない変換

```
1 void plus(int n) {
2   n++;
3   {
4     n--;
5     n++;
6   }
7   return n;
8 }
```

(a) APR の修正ソースコード 2

```
1 void plus(int n) {
2   n++;
3   {
4     n--;
5     n++;
6   }
7   return n;
8 }
```

(b) 空ブロックの削除後

```
1 void plus(int n) {
2   n++;
3   {
4   }
5   return n;
6 }
```

(c) 打消しあう単項演算子の削除後

図 4: 不要な処理が残る変換

しないが，図 4(c) では不要な処理である空のブロック宣言「{}」が残っている．これはルールの適用順序を事前に決めたためである．このような場合を回避するために，提案手法では事前にルールを決めるのではなく，探索的な変換を行う．

表 1: 変換ルール

変換ルール	変換前	変換後
同一変数に対する打ち消しあう単項演算子文の削除	`n++;` `n--;`	
同一変数に対する上書きが行われる部分の削除	`n++;` `n = 1;`	`n = 1;`
式を持たないブロック宣言の削除	`{n++;}`	`n++;`

制御分の条件式が常に true や false である制御文の省略	`if (true) {`
	`  n++;`
	`}`
		`n++;`

空のブロックの削除	`n++;`
	`{ }`
	`n--;`
		`n++;`
		`n--;`

不要な return 文の削除	`if (n > 0) {`
	`  n++;`
	`} else {`
	`  n--;`
	`  return n;`
	`}`
	`return n;`
		`if (n > 0) {`
		`  n++;`
		`} else {`
		`  n--;`
		`}`
		`return n;`

if-else 文の後にある処理を if-else 内に挿入	`if (n > 0) {`
	`  n++;`
	`} else {`
	`  n--;`
	`}`
	`n++;`
		`if (n > 0) {`
		`  n++;`
		`  n++;`
		`} else {`
		`  n--;`
		`  n++;`
		`}`

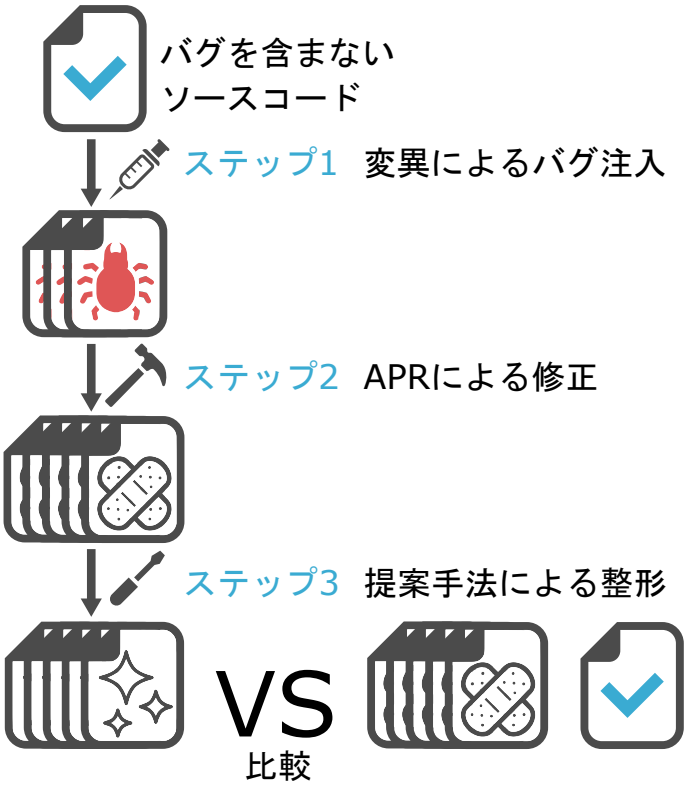

図 5: 実験概要

4 予備実験

4.1 概要

提案手法により，APR が修正したソースコードを人が記述する自然なソースコードへと変換可能か確かめるために，数行程度の小さな題材を用いた予備実験を行う．実験の流れを図 5 に示す．本実験はバグを含まないソースコードをベースラインとして実施する．このソースコードに対し，バグの注入，APR での修正，提案手法での変換の 3 つのステップを実行する．まず，APR が修正したソースコードを用意するためにソースコードにバグを注入する．バグ注入の方法として，ミューテーション解析 [19] を利用する．ミューテーション解析ではソースコード内の単一行を人為的に変更する．以降この処理を単に変異と呼称する．その際，1 つのソースコードに対し変異する箇所や内容が複数存在する．したがって，バグが注入されたソースコードは複数用意される．次に，それぞれのソースコードを APR で修正する．バグが注入されたソースコード 1 つに対しバグ修正されたソースコードを複数得る．これは，APR が複数のソースコードを出力する場合，探索が深くなり，機能的に不要な処理を多数含む可能性が上がるためである．最後に，バグ修正されたそれぞれのソースコードに対し，提案手法による変

```
1 int closeToZero(int n) {
2   if (n == 0) {
3   } else if (n > 0) {
4     n--;
5   } else {
6     n++;
7   }
8   return n;
9 }
```

図 6: 実験で用いるバグを含まないソースコード（ベースライン）

換を行う．提案手法による変換の効果を確認する方法として，変換前のソースコード及び，ベースラインのソースコードと行数を比較する．また，実用的な時間で変換可能か確認するため，APR の修正時間と提案手法の変換時間についても比較する．

4.2 実験設定

ベースラインとして用いる題材は図 6 に示すソースコードである．このソースコードは int 型の引数を受け取りその値が正であれば 1 を引き，負であれば 1 を足し，値を 0 に近づける．実験の各ステップにおける実験設定の一覧を表 2 に示す．図 6 のソースコードにミューテーション解析を用いて，表 3 に示す 4 つの変異を与えた．ソースコードを修正するための APR ツールとして，kGenProg[20] を用

表 2: 実験設定

ステップ	項目	値
1	変異数	4 個（表 3）
2	APR ツール	kGenProg[20]
	生成ソースコード数	1 試行あたり 10 個
3	対象ソースコード数	40 個
	評価値	行数

表 3: 与えた変異

変異 ID	変異した行	変異前コード	変異後コード
a	2	`n == 0`	`n != 0`
b	3	`n > 0`	`n <= 0`
c	4	`n--;`	`n++;`
d	6	`n++;`	`n--;`

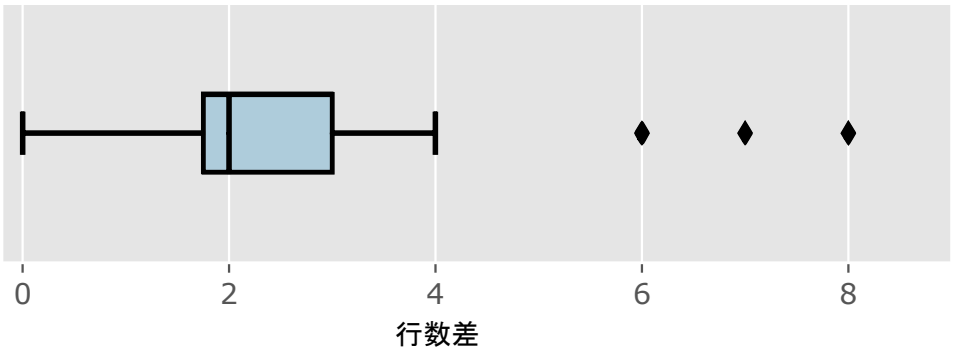

図 7: 変換前後での行数差の比較

いた．また，テストケースに関しては与える引数が正，負，0 の場合の全てを網羅している．バグ注入されたソースコード 1 つあたり，10 個の修正済みソースコードを生成する．APR により修正された 40 個のソースコードに対し，提案手法による変換を行う．提案手法における探索時の評価値には行数を用いる．行数を選んだ理由としては，題材が小規模であり，行数以外のメトリクスでは変換前後の差が生まれにくいからである．行数の変化が見られなくなるまで，つまりこれ以上の変換が不可能になった時を終了判定とした．

4.3 実験結果

4.3.1 変換前後でのソースコードの比較

提案手法による変換の効果を確認するために，各ソースコードの変換前後でのソースコードの行数の差を図 7 に示す．ここで，行数の差は変換前のソースコードの行数から変換後のソースコードの行数を引いた値である．図 7 より，多くのソースコードで行数が減少しており，変換処理が実施されていることが確認できる．ここで行数差が 0 行のソースコードはバグ修正の際に 1 つも不要処理を生まずに修正されたソースコードである．また，変換前後のソースコードの行数分布を図 8 に示す．図中の赤い点線はベースライン (図 6) の行数を表しており，この行数以下へと変換できている場合は不要処理が除去され変換が成功したと考えられる．図より全てのソースコードがベースラインの行数である 9 行以下へと変換された．今回の実験では 40 個中 35 個のソースコードが 9 行，残りの 5 個が 8 行へと変換された．次に，実際に不要処理が除去されているかを確認するために各ソースコードに対し目視を行った．図 9，図 10 に 9 行と 8 行へと変換されたソースコードの中身の一例を示す．図 9 より，変換後の行数が 9 行のソースコードは全てベースラインと同じ，もしくは if 文の条件式が異なるだけで，不要な処理は存在しなかった．図 10 より，8 行になったソースコードは APR による修正で if-else 文の条件式が変更され，8 行となっているが，ソースコードの中身には不要な処理は存在しなかった．以上を踏まえ

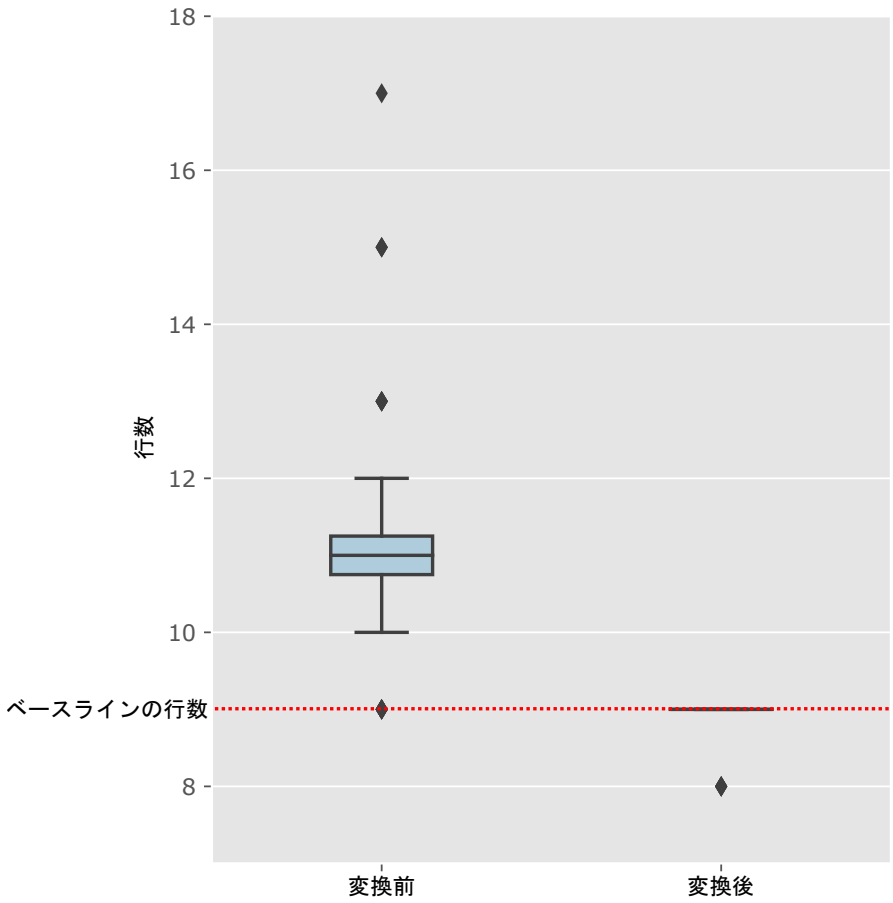

図 8: 変換前後での行数の比較

ると，40 個全てのソースコードで不要処理が除去でき，変換が成功したと言える．

4.3.2 変換の所要時間

提案手法による変換時間が実用的かを確認するために，APR の修正時間と比較する．ここで，比較対象はバグ注入されたソースコードから APR が 10 個の修正済みソースコードを出力する時間と，その修正された 10 個のソースコード全ての変換に要した時間である．その結果を図 11 に示す．横軸は表 3 の各変異 ID である．縦軸が所要時間である．変換に要した時間は平均して修正時間の 8% と極めて短く，実用的な時間で変換が完了した．また，各変異ごとの変換時間に大きな差は見られず，どのようなバグを修正したかに関わらず一定，なおかつ短時間で変換できた．修正と変換の時間にこれほどの差が出たことの要因は，3.2 節で述べた，変換前後で機能を保持する方法として，変換ルール自体が機能保持を満たす方法の採用によるテスト実行の省略が理由と考えられる．

```
1 int closeToZero(int n) {
2   if (n > 0) {
3     n--;
4   } else if (n == 0) {
5   } else {
6     n++;
7   }
8   return n;
9 }
```

図 9: 9 行へと変換されたソースコード

```
1 int closeToZero(int n) {
2   if (n > 0) {
3     n--;
4   } else if (n < 0) {
5     n++;
6   }
7   return n;
8 }
```

図 10: 8 行へと変換されたソースコード

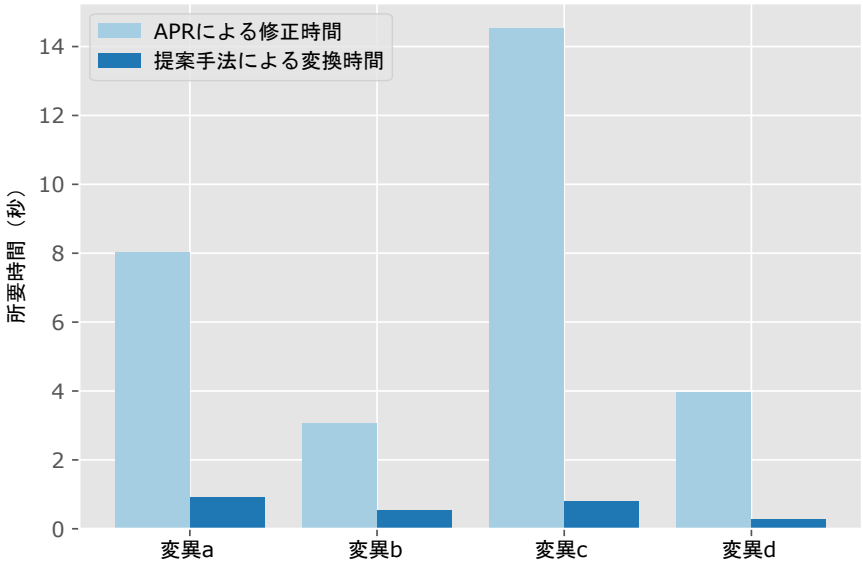

図 11: 修正と変換の所要時間の比較

4.3.3 変換ルールの使用回数

本実験において，題材との相性の良いルールを調べるために，各変換ルールの使用回数を表 4 に示す．表より，今回の実験題材において打ち消しあう部分の削除の使用回数が 30 回と，全体の約 58% を占めており，題材と相性の良いルールであると考えられる．また，使用されたルール間での共起関係の有無について調べるために，2 つのルール間で以下に定義する指標 x を算出した．

$$x = \frac{|A \to B|}{|A|}$$

ここで，$|A \to B|$ はルール A の直後にルール B が使用された回数であり．$|A|$ はルール A の使用回数である．この指標はルールの適用順序も考慮した指標であり，この値が高いほど 2 つのルールが同

時に出現し，かつ適用順序まで同じ可能性が高いということである．その結果を表 5 に示す．表より，if-else 文の後の処理を if-else 文内に挿入というルールの後に打消し部分の削除というルールの適用が 100% の確率で起こっている．この結果を踏まえると，提案手法の探索アルゴリズムにおける効率面での改善が考えられる．例えば，if-else 文の後の処理を if-else 文内に挿入というルールと打消し部分の削除を同時に適用するルールの考案といった共起性の高いルールを組み合わせて実行する改善が挙げられる．

表 4: 変換ルールの使用回数

変換ルール	使用回数
打消し部分の削除	30 回
不要 return 文の削除	10 回
if-else 文の後の処理を if-else 内に挿入	7 回
空ブロックの削除	5 回
上書き部分の削除	0 回
式を持たないブロック宣言の削除	0 回
制御文の省略	0 回

表 5: 2 ルール間における指標の値

A	B	指標の値
if-else 文の後の処理を if-else 内に挿入	打消し部分の削除	1
不要 return 文の削除	空ブロックの削除	0.2
空ブロックの削除	打消し部分の削除	0.2
打消し部分の削除	不要 return 文の削除	0.13
打消し部分の削除	打消し部分の削除	0.1
打消し部分の削除	空ブロックの削除	0.1

```
1  int closeToZero(int n) {
2    if (n > 0) {
3    } else if (n == 0) {
4      return n;
5    } else {
6      n++;
7      n++;
8    }
9    n--;
10   return n;
11 }
```

図 12: 不要処理を含む変換前のソースコード

```
1  int closeToZero(int n) {
2    if (n > 0) {
3      n--;
4    } else if (n == 0) {
5    } else {
6      n++;
7    }
8    return n;
9  }
```

(a) 提案手法により変換されたソースコード

```
1  int closeToZero(int var1) {
2    if (var1 <= 0) {
3      if (var1 == 0) {
4        return var1;
5      }
6      ++var1;
7      ++var1;
8    }
9    --var1;
10   return var1;
11 }
```

(b) 逆コンパイルにより変換されたソースコード

図 13: 提案手法と逆コンパイルの実行結果

5 関連研究

提案手法による不要処理の除去は一種のソースコードの最適化とも言える．最適化の手法には 2.2 節に述べた他にコンパイル最適化 [21] が存在する．そこで，提案手法とコンパイル最適化を比較する．比較方法としては本実験で用いた不要処理を含むソースコードに対し，コンパイル最適化後のバイトコードを逆コンパイルして比較する．図 12 に不要処理を含んだ変換対象のソースコードを示す．なお，使用言語は Java である．Java を用いた理由であるが，コンパイル後のバイトコードに対する逆コンパイルが容易なためである．それぞれの手法の実行結果を図 13 に示す．図 13(a) が提案手法の実行結果であり，不要処理の除去が確認できる．一方，図 13(b) は逆コンパイル後のソースコードであり，不要処

理の除去が確認できず，コンパイル最適化では APR が修正したソースコードを上手く変換できなかった．この理由として，Java における逆コンパイル可能なバイトコードへのコンパイルでは定数の畳み込みやデッドコードの排除等の最適化しか実行しないためと考える．ネイティブコードへのコンパイルでは定数の伝播や制御フローの解析により，さらなる最適化が実行されている [22]．しかし，ネイティブコードからソースコードへの復元は困難であり，ソースコードからソースコードへの変換である本研究と，ソースコードからネイティブコードへの最適化では出力の対象という点で異なる．

6　おわりに

本研究では機械によって生成された不自然なソースコードを自然なソースコードへと変換する手法を提案した．また，実際に小規模なプログラムを題材とした予備実験を行い，その効果を確認した．

今後の課題として，より一般的な変換ルールの追加が必須である．解決策として，変換ルールの自動生成を現在考案中である．例えば，変換前に対象のソースコードから制御フローやデータフローのグラフを作成することで，ソースコード中の各パーツの関係性を解析する．その結果を用いて，あるパーツとあるパーツが連続するときは削除を行う等のルールを生成する．このような仕組みを考案中である．また，図 1 に示したような実題材を用いた実験の実施も今後の課題である．

謝辞

本研究を実施するにあたり，多くの方にご尽力していただきました．

柗本真佑助教には私が困っている多くの時間で数多くの助言をしていただき，感謝してもしきれません．

楠本真二教授，肥後芳樹准教授には鋭く的確な意見をいただき，より良い研究へと導いてくださいました．心より感謝いたします．

楠本研究室の先輩方，同学年の皆様には研究に限らず研究室での生活等の些細な質問にも親身に教えていただきました．深く感謝いたします．

私を支えていただいた多くの方々にこの場をお借りして御礼申し上げます．

参考文献

[1] 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.

[2] 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, 2012.

[3] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *Proc. International Conference on Software Engineering*, pp. 772–781, 2013.

[4] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55, 2017.

[5] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated Atomicity-Violation Fixing. In *Proc. Conference on Programming Language Design and Implementation*, p. 389–400, 2011.

[6] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating Fixes from Object Behavior Anomalies. In *Proc. International Conference on Automated Software Engineering*, pp. 550–554, 2009.

[7] Fatmah Yousef Assiri and James M. Bieman. An Assessment of the Quality of Automated Program Operator Repair. In *Proc. International Conference on Software Testing, Verification and Validation*, pp. 273–282, 2014.

[8] 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 *Proc. Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[9] Fan Long and Martin Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proc. International Conference on Software Engineering*, pp. 702–713, 2016.

[10] He Ye, Matias Martinez, and Martin Monperrus. Automated Patch Assessment for Program Repair at Scale. *Empirical Software Engineering*, Vol. 26, pp. 1–38, 2021.

[11] 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.

[12] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proc. International Conference on Software Engineering*, pp. 802–811, 2013.

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

[14] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. A Multi-Study Investigation into Dead Code. *IEEE Transactions on Software Engineering*, Vol. 46, No. 1, pp. 71–99, 2020.

[15] Liang Tan and Christoph Bockisch. A Survey of Refactoring Detection Tools. In *Proc. Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems*, pp. 100–105, 2019.

[16] 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.

[17] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, Vol. 46, No. 10, pp. 1040–1067, 2018.

[18] René 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.

[19] 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.

[20] 柗本真佑, 肥後芳樹, 有馬諒, 谷門照斗, 内藤圭吾, 松尾裕幸, 松本淳之介, 富田裕也, 華山魁生, 楠本真二. 高処理効率性と高可搬性を備えた自動プログラム修正システムの開発と評価. 情報処理学会論文誌, Vol. 61, No. 4, pp. 830–841, 2020.

[21] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420, 1994.

[22] Michael G Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J Serrano, Vugranam C Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno Dynamic Optimizing Compiler for Java. In *Proc. conference on Java Grande*, pp. 129–141,

1999.