Toward Improving the Readability of Automatic Program Modification by Genetic Algorithms

Masayoshi TAKEDA* , Junya HISHIKAWA* , Erina MAKIHARA** and Keiko ONO***

(Received January 16, 2023)

Code debugging has become difficult due to the increasing complexity of source code in the face of rapid development of information technology. For this reason, automatic program modification is attracting attention. GenProg, which is a common automatic program repair tool using a genetic algorithm, uses only the test pass rate for evaluation and thus does not guarantee the readability of modification results. Therefore, this study aims to improve the readability of modified codes. We propose to consider the degree of similarity between GenProg generated codes with source codes, which are rated highly readable, to make individuals generated through the genetic algorithm more readable.

Key words : automatic program modification, genetic algorithm, GenProg, readability

キーワード : 自動プログラム修正, 遺伝的アルゴリズム, GenProg, 可読性

遺伝的アルゴリズムを用いた自動プログラム修正の可読性向上に向けて

竹田 将好， 菱川 潤哉， 槇原 絵里奈， 小野 景子

1. はじめに

IT 技術の発展に伴い，ソースコードが肥大化かつ複雑化し，デバッグ作業が困難になっている．デバッグ作業は多大な労力を必要とする作業であり，ソフトウェアの開発工数において半数以上を占めるといわれている．そのため，デバッグ作業の支援に関する研究は数多く行われ，なかでも自動プログラム修正が注目されている．

自動プログラム修正の方法には主に意味解析と遺伝的アルゴリズム [1] の 2 種類ある．意味解析を用いた SemFix[2] では，バグ箇所が満たすべき制約に基づいたプログラムを作成する．一方，遺伝的アルゴリズムを用いた GenProg[3] では，バグを含むプログラムとテストケースを入力とし，プログラムの改変およびテスト実行による評価を繰り返す．GenProg は修正コストが低く，高精度であるため近年注目されている．しかし，評価値にテスト通過率のみを用いているため，修正結果の可読性が保証されていない．

本研究では，GenProg で用いる評価値に対して，可読性が高いソースコードとの類似度を考慮することで，修正結果の可読性向上を行う．修正結果の可読性向上

* Graduate School of Science and Engineering, Doshisha University, Kyoto
Telephone:+81-774-65-6930, Fax:+81-774-65-6716, E-mail:takeda.masayoshi@mikilab.doshisha.ac.jp
** Faculty of Science and Engineering, Doshisha University, Kyoto
Telephone:+81-774-65-6930, Fax:+81-774-65-6716, E-mail:kono@mail.doshisha.ac.jp
*** Faculty of Science and Engineering, Doshisha University, Kyoto
Telephone:+81-774-65-6930, Fax:+81-774-65-6716, E-mail:emakihar@mail.doshisha.ac.jp

により保守・デバッグ作業の効率向上や，プログラム解析が容易になることでの他研究への貢献が考えられる．提案手法では，評価値に可読性が高いソースコードとの類似度を加味することで，生成個体となるソースコードの可読性を高くすることを目指す．

2. GenProg

2.1 GenProg の概要

GenProg は，遺伝的アルゴリズムを用いた自動プログラム修正法である [3]．遺伝的アルゴリズムでは，解候補を生物の個体に見立て，個体が終了条件を満たすまで遺伝的操作を繰り返す．そして，終了条件を満たした個体は，最適解に近い解となる．つまり，1 個体は 1 個のプログラムの修正案であり，解は個体集団で最良な修正後のプログラムである．

GenProg の流れを Fig. 1 に示す．まず，初期個体であるバグを含むプログラムに対して推定したバグに変更を加え，個体を複数生成する．バグの推定にはバグ限局 [4] を用いる．バグ限局とは，バグを含むプログラムに複数のテストを実行し，各テストの成否とテストの際に使用されたプログラム文の情報を用いることで，バグ箇所を推定する手法である．次に，生成した個体の評価を行う．評価値にはテスト通過率が用いられ，すべてのテストケースに通過した個体があれば，修正を終了する．すべてのテストケースに通過した個体がない場合，評価値の高い個体を一定数選択し，次世代の個体の生成元とする．生成，評価および選択を解が得られるまで繰り返す．

個体の生成において行われる操作を以下に示す．

変異 選択によって取り出された個体のバグ箇所に対して変更を加え，新たな個体を生成する．変更は，プログラム文の挿入，削除および置換からランダムに選択される．また，挿入や置換で用いるプログラム文は初期個体から選択される．

交叉 前世代の 2 個体において，修正過程で変化したプログラム文を組み合わせ，新たな個体を生成する．交叉には 2 種類の方法がある．一点交叉では個体のある部分を交叉点とし，交叉点より後ろの部分を交換する．一様交叉では，個体の部分ごとに交換するか否かを決定する．

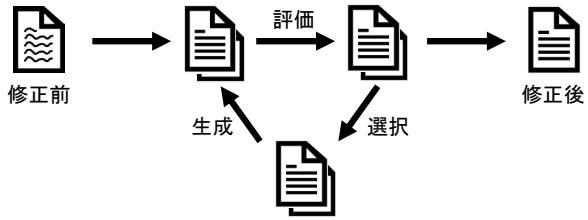


Fig. 1. Flow of GenProg

2.2 GenProg の課題点

GenProg は修正コストが低く，高精度であるが課題点も存在する．GenProg における考えられる課題点を以下に示す．

課題 1： 修正時間が膨大

すべての生成個体に対してコンパイルとテスト実行を行うため，修正に時間がかかる．近年開発された kGenProg[5] では全処理を JavaVM のヒープ内で行うことで，修正時間の短縮に成功している．

課題 2： 本来満たすべき機能を損なう可能性

テストケースに過剰適合することで，本来満たすべき機能を損なう可能性がある．具体的には，入力として与えられたテストケースは成功するが，値が異なるテストケースは失敗するというプログラムが出力される．

課題 3： 複数バグの修正が困難

修正対象に複数のバグが存在する場合，個体の評価が困難であり修正の効率を低下させる．具体的には，他の個体で修正できていないバグを修正した個体に対して，テスト通過率が低い場合は優秀だと評価できない．渡辺らの研究 [6] では，修正したバグ箇所によって個体にランク付けを行うことで，複数バグに対する効率的な修正を実現している．

課題 4： 可読性や保守性が未保証

評価値にテスト通過率のみを用いている

```
1     public int close_to_zero(int n) {
2         if (n == 0) {
3           else if (n > 0) {
4           n--;
5         } else {
6           n++;
7         }
8         n++; // bug here
9   -     return n;
10  +     n--;
11  +   return n;
12    }
```

Fig. 2. Example of modification causing wasteful processing

ため，可読性や保守性が保証されていない．岩瀬らの研究 [7] では，修正結果に対して再び遺伝的アルゴリズムを使用し，ルールに基づいた部分的な整形を繰り返すことで可読性向上を実現している．

上記の問題点のうち，本研究では課題 4 の修正結果の可読性に着目する．GenProg では修正する際に無駄な処理が発生することがある．Fig. 2 では，9 行目を削除し，10 行目と 11 行目を追加されており，“n++; n--;” という互いに打ち消し合う演算が行われている．

3. 類似度を考慮した自動プログラム修正法の提案

3.1 提案手法の概要

本稿では GenProg の可読性向上に注目し，評価値に可読性が高いソースコードとの類似度を考慮する．つまり，修正時における選択段階で可読性が高いソースコードとの類似度が高い個体を選択し，修正結果の可読性向上を目指す．類似度の計算には，GumTree[8] を使用する．また，GenProg では，評価値が高い順に個体を選択する．そこで，評価値のテスト通過率に可読性が高いソースコードとの類似度を加えることで，可読性を考慮した評価を可能にする．評価値の計算方法を式 (1) に示す．

$$F = T + S \tag{1}$$

F：評価値

T：テスト通過率

S：可読性が高いソースコードとの類似度

3.2 GumTree

GumTree は，入力された 2 個のソースコードに対して abstract syntax tree（以下，AST）を作成し，ノード単位でソースコードの差分を出力する．出力には，文字列として出力される編集スクリプトが用いられる．具体的には，ノードの挿入，削除，更新，移動の四つの操作を出力する．ソースコードおよび AST の例を Fig. 3 に示す．GumTree では，Fig. 3(a) のソースコードは Fig. 3(b) のような AST が作成される．そして，GumTree が出力する編集スクリプトの例を Fig. 4 に示す．Fig. 3(a) のソースコードに対して変更後のソースコードを Fig. 4(a) とすると，Fig. 4(b) のような編集スクリプトが出力提案手法では，出力された編集スクリプトを使用し，類似度を計算する．計算式を式 (2) に示す．

$$S = 1 - \frac{L}{T_1 + T_2} \tag{2}$$

S：類似度

T_1：一つ目のコードの AST に含まれるノード数

T_2：二つ目のコードの AST に含まれるノード数

L：編集スクリプトの行数

4. 類似度の検証実験

4.1 実験概要

提案手法では，修正中の個体に対して可読性が高いソースコードとの類似度を用いることで，修正結果の可読性向上を目指している．したがって，可読性が高いソースコードとの類似度を用いるには、可読性が高いソースコード間の類似度が高い必要がある．そこで，類似性を検証するための実験を行う．実験手順を以下に示す．

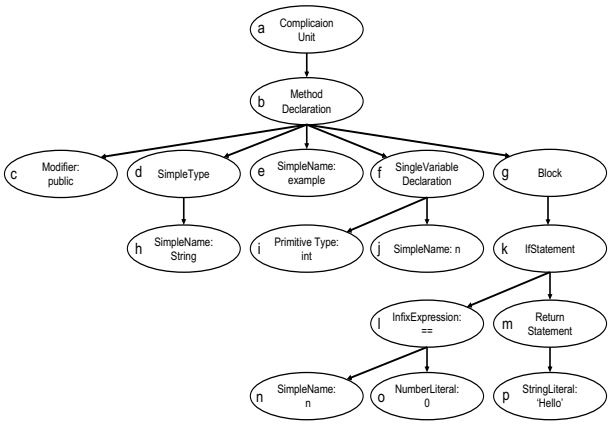
手順 1： GumTree を使用しソースコード間の類似度を算出

手順 2： ソースコードの可読性を定義

手順 3： 可読性が高いと評価されたコード間の類似度を調査

```
1   public String example(int n) {
2       if (n == 0) {
3         return ‘Hello’
4       }
```

(a) Source code



(b) AST

Fig. 3. Example of GumTree[9)]

次に，実験条件を示す. 題材として，プログラミングコンテスト AtCoder で過去に開催された AtCoder Beginner Contest (以下，ABC) のうち ABC101 から ABC103 までの A 問題 3 問を用いる．A 問題は，簡単な文法の理解を確認する問題である．各問題の提出結果から Java 言語のソースコードを 10 個ずつ取得し，5 人で評価を行う．可読性は 5 段階で評価する．5 段階評価で 0 が最も低く，5 が最も高いと評価されたと設定する．そして，5 人の平均評価が 3.5 以上のソースコードを可読性が高いソースコードと定義する．

次に，類似度を調査する手順を述べる．はじめに，ソースコード間の類似度を算出する．次に，可読性が高いソースコードと類似度が高いソースコードを比較し，式 (3) を使用して一致率を算出する. このとき，r は可読性かつ類似度が高いソースコード数であり，p は可読性が高いソースコード数である．

$$mr = \frac{r}{p} \tag{3}$$

ABC103 で実際に調査した結果を Table 1 に示す．可読性が高いソースコードとの類似度のみを記載し，可読性が高いソースコードと高い類似度を太字にしている．コード 1 と 7 の類似度が異なっているが，編集

```
1   private String example(int n) {
2       if (n == 0) {
3         return ‘Good’
4       }
5       else if (n == 1) {
6         return ‘Hello’
7       }
```

(a) Edited source code

```
1  insert(t1, k, 2, ReturnStatement, α)
2  insert(t2, t1, 1, StringLiteral, Good)
3  insert(t3, k, 3, IfStatement, α)
4  insert(t4, t3, 2, InfixExpression, ==)
5  insert(t5, t4, 1, SimpleName, n)
6  insert(t6, t4, 1, NumberLiteral, 1)
7  move(m, t3, 2)
8  update(c, private
```

(b) edit script

Fig. 4. Example of edit script output by GumTree

スクリプトの行数が異なっているからである．

4.2 実験結果および考察

実験結果である各問題における一致率の平均を Table 2 に示す．今回使用したのは 20 から 30 行程度の簡単なソースコードが多く，ソースコードは同じ問題を使用している．それにもかかわらず一致率が低いため，可読性が高いソースコード間の類似性は確認できないと考えられる．一致率が低い理由として，可読性の評価値が被験者によって異なることが考えられる．具体的には，import が多いことで可読性が低いと感じたり，使用されていない関数があると可読性が低いと感じるなど，可読性の評価は人によって様々であった．また，ABC102 の一致率が他の問題と比べて低くなっている．原因として，ABC102 のソースコードの行数が極端に短い問題や，未使用の関数がある問題が含まれていることが考えられる．実施した調査方法では各問題のソースコード数のうち可読性が高いソースコード数の割合が高いと，一致率が高くなる傾向にあることが挙げられる．

5. 提案手法の評価実験

5.1 実験概要

4 章において，類似度の検証実験を行った結果，可読性が高いソースコード間の類似性は確認できなかっ

Table 1. Findings of similarity between source codes in ABC103

	コード 1	コード 2	コード 3	**コード 4**	**コード 5**	コード 6	**コード 7**	コード 8	**コード 9**	**コード 10**	一致率
コード 1	1	**0.7526**	**0.7532**	0.7156	0.6594	**0.7755**	0.6995	**0.7782**	**0.7305**	0.6461	0.2
コード 4	0.7156	**0.8197**	**0.7705**	1	0.7135	**0.7913**	**0.7840**	**0.7826**	0.7442	0.7179	0.2
コード 5	0.6594	**0.7992**	**0.7216**	0.7081	1	**0.7744**	**0.6958**	**0.7440**	0.64	**0.9545**	0.2
コード 7	**0.7558**	0.5445	0.6041	**0.7840**	0.6958	**0.8288**	1	**0.7949**	**0.7675**	0.6470	0.6
コード 9	0.7305	**0.7550**	**0.7764**	**0.7442**	0.645	**0.7955**	0.7149	**0.7877**	1	0.6523	0.2
コード 10	0.6461	**0.7919**	**0.7297**	0.7179	**0.9545**	**0.7918**	0.6470	**0.7466**	0.6523	1	0.2

Table 2. Results of experiments

問題	一致率の平均	可読性が高いコード数
ABC101	0.62	7
ABC102	0	3
ABC103	0.27	6

た．よって，本実験では提案手法での目標を無駄な処理を排除することによる可読性向上に留める．実験手順を以下に示す．

手順 1： 従来手法での修正時に無駄な処理が発生するソースコードを作成

手順 2： 可読性が高いソースコードを作成

手順 3： 提案手法での修正時に無駄な処理が排除できたか確認

題材として，kGenProg が用意したバグを含むソースコードを使用し，修正時に無駄な処理が発生するよう変異する．従来手法における無駄な処理が発生する修正例を Fig. 5 に示す．Fig. 5(a) では無駄な行を追加している．Fig. 5(b) ではバグを打ち消す行を追加することで，修正結果において無駄な行が生まれている．提案手法では，題材の修正時に無駄な処理が発生しないことを目指す．可読性が高いソースコードの例を Fig. 6 に示す．Fig. 6 では変数名が number だが，提案手法では変数名が n の状態を維持しつつ，Fig. 6 の制御構造に近づけることを目指す．

実験設定を Table 3 に示す．題材数は 10 で，言語は Java，平均行数は 11.8 行である．含まれているバグの行数は 1 または 2 行で，修正ツールは kGenProg を使用する．また，提案手法で出力する個体に類似度

```
1 -   n++; // bug here
2 +   {
3 +     n++;
4 +     n--;
5 +   }
6     return n;
7  }
```

```
1     } else {
2       n++;
3 +     return n;
4     }
5 -   n++; // bug here
6     return n;
7 }
```

(a) Add useless lines

```
1     n++; // bug here
2 +   n--;
3     return n;
4   }
```

(b) Add a line to counteract the bug

Fig. 5. Image of experimental subject

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

Fig. 6. Example of highly readable source code

のしきい値を追加した．GenProg では，すべてのテストケースに通過する個体が生成された時点で修正が終了するからである．本実験では，無駄な処理を目視で消したソースコードと可読性が高いソースコードの類似度を測った上で，しきい値を 0.92 とした．

5.2 実験結果および考察

実験結果を Table 4 に示す．題材である 10 個のソースコードのうち，7 個のソースコードで可読性向上を確認した．従来手法と比べて減少した行数は，最大で 4 行である．これは，本実験で使用した題材において，従来手法で追加される無駄な処理の行数が最大でも 4

Table 3. Experimental settings

項目	値
ソースコード数	10
言語	Java
平均行数	11.8
最小行数	10
最大行数	18
バグ行数	1 行または 2 行
修正ツール	kGenProg

行程度だったことに起因する．失敗したソースコードは 3 個だが，失敗パターンは 2 パターンに分けられる．以下に二つの失敗パターンを示す．

パターン 1：しきい値を超える個体が生成されない

実際に失敗したソースコードを Fig. 7 に示す．Fig. 7(a) では，バグである 8 行目を削除した個体が生成されなかった．原因として，複数のテストに失敗するためバグ限局ができないことが考えられる．しかし，8 行目が “n++;” である個体は可読性向上に成功していた．

Fig. 7(b) では， バグである 6 行目と 10 行目を削除した個体が生成されなかった．原因として，正の数を入力するテストには通過するため，6 行目をバグだと認識できなかったことが考えられる．しきい値を超える個体が生成されない場合は，修正結果が出力されない．解決策として，一定時間が経過した時点で類似度が一番高い解個体を出力することが考えられる．

パターン 2：可読性の低い個体がしきい値を超える

実際に失敗したソースコードを Fig. 8 に示す．Fig. 8 では，可読性の低い個体が出力された．原因として，行数が多いことでバグの箇所が相対的

```
1  public int close_to_zero(int n) {
2       if (n == 0) {
3       } else if (n > 0) {
4            n--;
5       } else {
6            n++;
7       }
8       n--; // bug here
9       return n;
10 }
```

(a) Source code failing multiple tests

```
1  public int close_to_zero(int n) {
2       if (n == 0) {
3         return 0;
4       } else if (n > 0) {
5         n--;
6         n++; // bug here
7       } else {
8         n++;
9       }
10      n--; // bug here
11      return n;
12 }
```

(b) Source code with multiple bugs

Fig. 7. In case no individuals exceeding the threshold are generated

に少なくなり，生成個体の類似度が高くなることが考えられる．解決策として，行数によってしきい値を変更することや，類似度算出の範囲をバグを含むメソッドに限定することが考えられる．

6. 妥当性の脅威

5 章で行った実験に存在する妥当性への脅威について述べる．内的妥当性に対する脅威として，kGenProg 全体で使用する乱数のシード値による影響が考えられる．本実験ではシード値として 0 を使用し，シード値が 0 の場合に従来手法において無駄な処理が発生するソースコードを使用した．遺伝的アルゴリズムでは乱

Table 4. Results of experiments

○: 可読性が向上したプログラム，×: 可読性が向上しなかったプログラム

コード	1	2	3	4	5	6	7	8	9	10
結果	○	○	×	○	○	○	×	○	×	○
減少した行数	1	2	×	4	3	3	×	2	×	1

```
1  public int close_to_zero(int n) {
2      if (n == 0) {
3        // do nothing
4      } else if (n < 0) { // bug here
5        n--;
6      } else {
7        n++;
8      }
9      n--; // bug here
10     return n;
11   }
12
13   // 再利用されるべきメソッド1
14 public int reuse_me1(int n) {
15     if (n == 0) {
16     } else if (n > 00 {
17       n++;
18     }
19     return 0;
20 }
```

Fig. 8. In case of a large number of lines

択に基づく処理を行うため，実行結果は乱択に依存する．よって，シード値を変更して実行すると，従来手法で無駄な処理が発生しない可能性や，異なる実験結果が得られる可能性がある．次に外的妥当性について述べる．本実験では，題材として kGenProg が用意したソースコードを使用した．外的妥当性を確保するためには，別の題材を使用した実験を行う必要がある．

7. おわりに

GenProg では評価値にテスト通過率のみを用いているため，可読性や保守性が保証されていない．そこで，可読性が高いソースコードとの類似度を考慮した自動プログラム修正法を提案した．提案手法の正当性を示すため，可読性が高いソースコード間の類似度について検証実験を行った．結果，一致率は低く，可読性が高いソースコード間の類似性は確認できなかった．一致率が低い理由として，可読性の評価値が被験者によって異なることが考えられる．提案手法での目標を，無駄な処理の排除による可読性向上に留めて評価実験を行った結果，10 個中 7 個のソースコードに可読性向上を確認した．失敗するパターンとして，バグ限局が難しいことによりしきい値を超える個体が生成されない場合や，修正対象の行数が多いことにより可読性の低い個体でもしきい値を超える場合があった．

今後の研究では，シード値や題材を変更し再度実験を行う．また，修正結果に個人の癖や傾向を反映する．そのため，類似度を測定するソースコードを読む人が書いたものにする．変数の長さや名前の付け方，コメントなどから癖や傾向を読み取り GenProg の評価値に組み込む．個体の変異における挿入や置換で使用するプログラム文は，初期個体だけでなく，可読性が高いソースコードから選択する．修正結果に個人の癖や傾向を反映することで，類似度を測定するソースコードが異なる内容でも修正結果の可読性向上が可能になると考えられる．

参考文献

1) 北野宏明, “遺伝的アルゴリズム”, 人工知能, **7**[1], 26-37 (1992).
2) H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, “Semfix: Program Repair via Semantic Analysis”, *the 35th International Conference on Software Engineering (ICSE)*, 772-781 (2013).
3) C. L. Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, “A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each”, *In Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 3-13 (2012).
4) R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. C. V. Gemund, “A Practical Evaluation of Spectrum-based Fault Localization”, *Journal of Systems and Software*, **82**[11], 1780-1792 (2009).
5) Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Yomida, S. Kasumoto, “kGenProg: A High-Performance, High-Extensibility and High-Portability APR System”, *the 25th Asia-Pacific Software Engineering Conference (APSEC)*, 697-698 (2018).
6) 渡辺大登, まつ本真佑, 肥後芳樹, 楠本真二, 倉林利行, 吉村 優, 切貫弘之, 但馬将貴, 丹野治門, “多目的遺伝的アルゴリズムを用いた自動プログラム生成手法の提案 プログラミングコンテストを題材として ”, 信学技報, **120**[407], 31-36 (2021).
7) 岩瀬匠, まつ本真佑, 楠本真二, “自動生成コードの可読性向上を目的とした探索的ソースコード整形手法の提案”, 信学技報, **121**[318], 13-18 (2022).
8) J. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, “Fine-grained and Accurate Source Code Differencing”, *the 29th ACM/IEEE international conference on Automated software engineering*, **60**, 313-324 (2014).
9) Y. Higo, A. Ohtani, S. Kusumoto, “Generating Simpler AST Edit Scripts by Considering Copy-and-Paste.”, *the 32nd ACM/IEEE international conference on Automated software engineering*, 532-542 (2017).