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

菱川 潤哉
Junya HISHIKAWA

1 はじめに

近年，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言われている．そのためデバッグ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の 2 種類ある．意味解析を用いた SemFix 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[1] 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．よって，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．

2 GenProg

2.1 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．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．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 つ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 Figure 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，全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．全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．

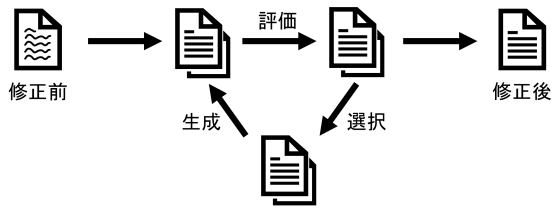

Figure 1 GenProg の流れ

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．

2.2 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．全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ことである．しかし，近年開発された kGenProg 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．また，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．加えて，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．

3 提案手法

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組み込む．つまり，ソースコードの修正時における選択段階で可読性の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[2] を使用する．GumTree は，抽象構文木のノード単位でソースコードの差分を出力する．具体的には，ノードの挿入，削除，更新，移動の 4 つの操作を出力する．

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

4 評価実験

4.1 概要

提案手法の有効性を検証するため，実験を行う．以下に実験手順を示す．

手順 1：従来手法で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する．これを，修正時に無駄な処理が発生するよう変異する．従来手法で無駄な処理が発生する修正例を Figure 2 に示す．Figure 2 では，修正時に 9 行目を削除し，10 行目と 11 行目を追加している．修正後は，8 行目と 10 行目で ”n++; n--;” という互いに打ち消し合う演算が行われており，無駄な処理が生まれている．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 Figure 3 に示す．Figure 3 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，Figure 3 の制御構造に近づけることを目指す．実験設定を示す．題材数は 10 で，言語は Java，平均行数は 11.8 行である．最小行数は 10 行，最大行数は 18 行であった．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．

4.2 結果と考察

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．また，減少した行数の平均は 1.6 行だった．以下に 2 つの失敗パターンと，解決策を示す．

パターン 1：しきい値以上の個体が生成されない場合

複数のバグが原因でバグ限局ができず，可読性の高い個体が生成されない場合がある．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個体を出力することが考えられる．これにより修正結果を出力することができる．

```
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   }
```

Figure 2 無駄な処理が発生する修正例

```
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    }
```

Figure 3 可読性が高いソースコードの例

パターン 2：行数が多い場合

題材の行数が多いとバグの箇所が相対的に少なくなり，個体の類似度が高くなる．解決策として，行数からしきい値を算出する式を定義することや，可読性が高いソースコードとの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．これにより適切なしきい値を使用することができる．

5 まとめ

本稿では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．また，可読性が向上しなかった原因と解決策を提案した．

今後の研究では，提案手法を使用した際の修正時間や精度を調査する．また，修正結果に個人の癖や傾向を反映する．これにより，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．

参考文献

1) Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley 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, pp. 3–13, 2012.

2) J. Fllerai, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, Vol. 60, pp. 313–324, Sep. 2014.