特別研究報告

題目

コード片再利用に基づいたプログラム自動修正に対する再利用候補限定に向けた調査

指導教員

楠本 真二 教授

報告者

横山　晴樹

平成 27 年 2 月 13 日

大阪大学 基礎工学部 情報科学科

平成 26 年度 特別研究報告

コード片再利用に基づいたプログラム自動修正に対する再利用候補限定に向けた調査

横山　晴樹

内容梗概

デバッグ工数の削減に向けて，プログラムの自動修正が望まれる．一般に，プログラムの修正はバグの同定と同定したバグの修正により構成される．つまり，自動修正に向けて，バグ同定とバグ修正の自動化が必要である．これまでに，自動バグ同定の研究は盛んに行われているのに対し，自動バグ修正に関する研究はあまり行われていない．自動バグ修正は，バグを含んだソースファイルに変更を加え，修正プログラムを生成するものである．加える変更には，コード片の追加や削除などがある．コード片の追加は，修正に用いるコード片をバグ同定によって特定された位置に挿入する操作である．しかし，修正候補は構文が許容する限り無限に存在するため，追加するコード片の決定は困難である．そこで，再利用に基づく自動修正手法が提案されている．再利用に基づく自動修正手法では，修正候補を同一プロジェクト内の既存コードに限定している．追加するコード片は，修正候補よりランダムに 1 行選択する．しかし，大規模なソフトウェアを対象とした場合，再利用されるコード片候補の数が膨大になるという問題がある．再利用候補をある程度絞り込むことができれば，より短時間で自動修正を行えると期待できる．仮に，追加元コード片の周辺領域と追加先コード片の周辺領域間に何らかの関係があれば，再利用候補を限定することができる．そこで本研究では，追加元コード片の周辺領域と追加先コード片の周辺領域間における類似度について調査を行った．本研究では，OSS を対象として，追加元コード片の周辺領域と追加先コード片の周辺領域の類似度 (A)，同一プロジェクトの各周辺領域と追加先コード片の周辺領域の類似度 (B) を算出し，類似度 A と類似度 B の比較を行った．調査の結果，4 つの OSS プロジェクトにおいて，類似度 A の中央値および平均値が B よりも高くなることがわかった．また，仮説検定により，A と B の平均値に差が認められることを示した．

主な用語

デバッグ
プログラム自動修正
コード再利用
ソフトウェアリポジトリマイニング

目 次

表 目 次

図 目 次

1　まえがき

ソフトウェアの安全性や信頼性の向上のために徹底的なデバッグが必要である．デバッグとはソフトウェアの故障を取り除く作業のことを示す．一般的に，デバッグはソフトウェアの故障を検出し，ソフトウェアの故障を引き起こす原因となったプログラム中の記述（欠陥）を同定し，同定した欠陥を修正する作業により構成される．

デバッグは工数のかかる作業として知られており，ソフトウェア開発者はソフトウェア開発作業の内，50 % をデバッグに費やすといわれている [1]．また，アメリカにおいて 1 年間にデバッグで費やされる費用は約 3000 億ドルであるという調査もされている [2]．

そのため，デバッグにかかる工数を削減することを目的として，デバッグの自動化に関する研究がなされてきた．ソフトウェアの故障検出はテストケースにより行う．テストケースはソフトウェアに期待する特定の動作を記述しており，ソフトウェアがテストケースを成功することはソフトウェアがそのテストケースに記述した動作を満たすことを示す．逆に，ソフトウェアがテストケースを失敗することはソフトウェアがそのテストケースに記述した動作を満たさないことを示す．ソフトウェア開発者はテストケースの失敗によりソフトウェアの故障を検出する．ソフトウェアの故障検出を徹底的に行うためには，膨大な数のテストケースを生成する必要があり人手で行うと工数がかかる．よって，テストケース生成の工数削減を目的とし，膨大な数のテストケースを自動生成する研究がなされてきた [3, 4, 5, 6]．開発者はテストケースによりソフトウェアの故障を検出すると，故障の原因となった欠陥を同定する必要がある．欠陥は膨大なプログラム中の数箇所に存在するため，人手で行うと工数がかかる．よって，欠陥同定に関する工数削減を目的とし，欠陥同定を自動で行う研究がなされてきた [7, 8, 9]．デバッグを構成する 3 工程の内，ソフトウェアの故障検出と欠陥同定の自動化については多くの研究がなされており，実用段階であるが，同定した欠陥の自動修正に関する研究はあまり行われていない．さらに，欠陥修正には開発者がプログラムの動作を理解し，どのようにプログラムを修正すれば欠陥が取り除かれるのかを熟慮する必要がある。そのため，欠陥修正は工数がかかる作業となる．つまり，デバッグ作業の完全な自動化にはプログラムの自動修正が必要である．

そこで，近年プログラムの自動修正に注目が集まっている．プログラム自動修正に関する研究は 2 つに大別される．1 つは確率的な手法で，ランダムな工程を繰り返し行うことでプログラムの修正を行う．もう一方はプログラムが満たすべき性質を表す論理式を元に修正を行う．論理式を元にする修正手法では，複雑な論理式を SMT ソルバ　[10] で解く必要がある．そのため，現在の CPU の処理能力では，現実的な時間に論理式を解くことは困難である．よって，本研究では確率的な手法に着目する．

確率的な手法では GenProg が有望視されている [11, 12]．GenProg は既に存在するプログラム記述を用いて欠陥は修正できると仮定してプログラムの自動修正を行う．つまり，GenProg は同定した欠陥についてその記述をソースコード中に存在する別の記述に書き換える作業を繰り返し行うことでプログラムの自動修正を行う．既存プログラム記述の選択は完全にランダムで行われる．Barr

らはバージョン管理システムで開発されたプロジェクトにおいて GenProg で用いる仮定の正しさを検証した [13]．検証の結果，開発者がコミット間に追加したソースコードの内，約 40% ものプログラム記述が既存のプログラム中に存在する記述を用いることで開発可能であることを示した．

GenProg は既存のプログラム記述をランダムに選択・挿入するため，修正対象プログラムの規模が大きくなると適切なプログラム記述の発見が困難になる．つまり，大規模なソフトウェアの自動修正では挿入候補の限定が課題となる．仮に，追加元コード片と追加先コード片に何らかの相関があれば，再利用候補を限定することができる．本研究では，追加元コード片の周辺領域と追加先コード片の周辺領域間における類似度の相関について，調査を行った．調査は，OSS のリポジトリを解析し，実際に行われたバグ修正のデータを対象とした．また，追加元コード片の周辺領域と追加先コード片の周辺領域の類似度 (A)，同一プロジェクトの各周辺領域と追加先コード片の周辺領域の類似度 (B) を算出し，類似度 A と類似度 B の比較を行った．調査の結果，4 つの OSS プロジェクトにおいて，類似度 A の中央値および平均値が B よりも高くなることがわかった．また，仮説検定により，A と B の平均値に差が認められることを示した．

以降，2 章では，プログラム自動修正の関連研究について述べる．3 章では，関連研究の課題を提示し，調査項目を述べる．4 章では，具体的な調査方法と，その実装について述べる．5 章では，調査対象と調査結果を述べる．6 章では，調査結果を踏まえた考察を述べる．最後に，7 章では，あとがきを述べる．

2 関連研究

本節では，本研究に関連する諸技術について述べる．

2.1 遺伝的アルゴリズム

遺伝的アルゴリズムは生物が環境に適応することを工学に応用したアルゴリズムである [14]．遺伝的アルゴリズムはランダムな個体生成及び，個体の評価値に基づく生存個体選択を繰りかえし行うことで動作する．まず，遺伝的アルゴリズムは初期集団と呼ばれる個体群をランダムに生成する．これが第 1 世代となる．個体数は個体群が含む個体数を定義する．生成した各個体の環境適応度は環境を模倣した評価関数により評価される．ここで，環境適応度が高い個体ほど解に近い事を示す．よって，解に近い個体を生存させるために環境適応度に基づき生存個体を選択する．生存個体の選択方法，選択数はアルゴリズム設計毎に異なる．次に，生存した個体を元に次世代を生成する．次生成方法もアルゴリズム設計毎に異なる．遺伝的アルゴリズムは世代生成を繰りかえし行うことで解を探索する．遺伝的アルゴリズムは解の発見または次世代生成回数の規定回数（最大世代数）到達で終了する．

2.2 GenProg

GenProg は遺伝的アルゴリズムに基づきプログラムの自動修正を行う手法である [11, 12]．本節では，まず，GenProg の基本的なアイデアを説明するために初期に提案された GenProg [11] について述べる．次に，改良版の GenProg [12] について述べる．

2.2.1 GenProg の概要

GenProg [11] は遺伝的アルゴリズムに基づいてプログラムを自動修正する手法である．GenProg は故障を含むソフトウェア及び少なくとも 1 つの失敗テストケースを含むテストスイートを入力とし，プログラムの自動修正を行う．ここで，入力するソフトウェアを構成するプログラムを修正対象プログラムと呼ぶ．この入力はソフトウェアの故障をテストスイートにより検出することを想定すると自然である．出力は修正プログラムである．修正プログラムは修正が完了したプログラムを示す．また，GenProg はプログラムの自動修正のみを行う手法ではなく，テストスイートをもとに欠陥同定を行い同定した欠陥を修正する手法である．欠陥の同定部分と欠陥の修正部分は独立しているため任意の欠陥同定手法を GenProg に適用することができる．

遺伝的アルゴリズムの設計では個体の表現，個体の操作及び個体の評価関数を設計する必要がある．本小節では， GenProg の動作概要を述べる．個体の表現及び個体の操作及び個体の評価関数の詳細は次小節以降で述べる．

GenProg の入力は故障を含むソフトウェア P 及びテストスイートである．テストスイートは修正対象プログラムが成功するテストケースの集合 $PosT$ 及び修正対象プログラムが失敗するテスト

Algorithm 1 : The GenProg Algorithm

Require: : Faulty program P

Require: : Testsuite including set of positive/negative testcases $PosT$ and $NegT$

Ensure: : Patch that repairs the faulty program P

1: $Path_{PostT} \leftarrow \cup_{p \in PosT}$ statements visited by $P(p)$

2: $Path_{NegT} \leftarrow \cup_{p \in NegT}$ statements visited by $P(n)$

3: $C_{sub} \leftarrow$ FaultLozalize$(Path_{NegT}, Path_{PosT})$

4: $Popul \leftarrow$ initial_population$(P, \text{pop_size}, \text{C}_{\text{sub}})$

5: **repeat**

6: $\quad Viable \leftarrow \{\langle P, Path_p, f\rangle \in Popul \mid f > 0\}$

7: $\quad Popul \leftarrow \emptyset$

8: $\quad NewPop \leftarrow \emptyset$

9: $\quad$ **for all** $\langle p_1, p_2\rangle \in$ sample($Viable$, pop_size/2) **do**

10: $\quad\quad \langle c_1, c_2\rangle \leftarrow$ crossover(p_1, p_2)

11: $\quad\quad NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$

12: $\quad$ **end for**

13: $\quad$ **for all** $\langle V, Path_V, f_V\rangle \in NewPop$ **do**

14: $\quad\quad Popul \leftarrow Popul \cup \{$mutate$(V, Path_V)\}$

15: $\quad$ **end for**

16: **until** $\exists \langle V, Path_V, f_v\rangle \in Popul.f_V =$ max_fitness

17: **return** minimize$(V, P, PosT, NegT)$

ケースの集合 $NegT$ の和集合である．ソフトウェアの故障の原因となるものが欠陥である．また，ソフトウェアの故障はテストにより検出する，よって， $NegT$ は空集合ではない．また，GenProg は全テストケースを成功するプログラムを修正プログラムとしている．すなわち，修正プログラムの質はテストケースに依存する．つまり，GenProg により導出した修正プログラムは開発者が望んだ修正を加えたプログラムでない可能性がある．Algorithm 1 に GenProg の擬似コードを示す．まず，1 から 3 行目でテストスイートに基づき欠陥同定を行う．1，2 行目で成功テスト及び失敗テストの適用時に修正対象のプログラムが実行するプログラムの行を導出する．$P(p)$ 及び $P(n)$ はそれぞれ，修正対象のプログラムに対する成功及び失敗テストの適用を示す．$Path_{PosT}$ 及び $Path_{NegT}$ はそれぞれ修正対象のプログラムが成功テスト及び失敗テスト適用時に実行するプログラム中の行を表す．3 行目では $Path_{PosT}$ 及び $Path_{NegT}$ に基づき欠陥同定を行う．欠陥同定ではプログラムの各行に疑惑値を付与する．疑惑値はプログラムの各行における欠陥である可能性を示す値である．4 から 16 行目が GenProg における遺伝的アルゴリズムを示す．4 行目で初期集団の生成を行い，5 から 16 行目の繰り返しで世代生成を繰り返す．世代生成の繰り返しは修正プログラムの発見或いは規定回数の世代生成で終了する．4 行目で関数 initial_population を用いて初期集団を生成する．pop_size は 1 世代の個体数を表す．initial_population は後述する変異処理により修正対象プログラムを変更したプログラムを pop_size 個生成する．6 行目で 1 つのテストケースも成功しない変更プログラムを除外する．次に，各変更プログラムの評価値を評価関数に基づき評価する．このとき，より多くのテストケースを成功する変更プログラムの評価値が高くなる．つまり，より修正プログラムに近いプログラムの評価値が高くなる．評価値に基づき pop_size/2 個の 変更プログラムが次世代に生存する．9 行目は 変更プログラムの評価及び生存個体の選択を表す．生存する変更プログラムの内，2 つの変更プログラム又し次世代の変更プログラムを生成する．$variant_1$ と $variant_2$ を交叉する場合，$variant_1$ の前半部分と $variant_2$ の後半部分で構成される変更プログラム及び $variant_1$ の後半部分と $variant_2$ の前半部分で構成される変更プログラムを生成する．ここで，生存する変更プログラムは交叉に用いられた変更プログラム及び交叉で生成された変更プログラムで構成されることに注意されたい．次に，生存する変更プログラムを突然変異によりさらに変更することで，次世代を生成する．13 から 15 行目が突然変異の処理を示す．遺伝的アルゴリズムは修正プログラムの生成或いは，規定回数の世代生成により終了する．GenProg で用いる遺伝的プログラムの性質上，可読性の低い冗長な修正プログラムを生成する可能性がある．そこで，delta debugging [15] を用いて冗長な記述を削減する，17 行目は修正プログラムの冗長性削減を関数 minimization を用いて行う．次小節以降で，個体の表現，個体の操作，個体の評価関数及び修正プログラムの冗長性削減について詳細に述べる．

2.2.2 個体の表現

GenProg での個体は修正対象プログラムを数行変更したプログラムである．GenProg は各変更プログラムを抽象構文木と重み付きパスで表現する．重み付きパスは抽象構文木の各ノードと各ノード

に関する疑惑値で構成するペアの系列である．疑惑値は各ノードが欠陥である可能性を示す．つまり，疑惑値が高ければ高いほどそのノードが欠陥である疑惑が強くなる．各ノードに対する疑惑値の付与は修正対象のプログラムにテストスイートを適用した結果を用いる．Algorithm 1 中の set_weights 関数は各ノードに対する重み付けを行う．GenProg は成功テストが到達するプログラム中の記述は欠陥である可能性が低いとして set_weights 関数を定義する．まず，set_weight 関数は全ノードに疑惑値を 0 付与する．次に，修正対象プログラムに失敗テストを適用した結果到達するノードに疑惑値 1.0 を付与する．次に，修正対象のプログラムに成功テストを適用した結果到達するノードに疑惑値として W_{path} を付与する．Weimer らは $W_{path} = 0.01$ としている [12]．結果として，失敗テスト及び成功テストの実行共に到達するノードの疑惑値が W_{path} 失敗テストの実行でのみ到達するノードの疑惑値が 1.0，成功テストの実行でのみ到達するノードの疑惑値が W_{path}，テストケースの実行で到達しないノードの疑惑値が 0 となる．

2.3 個体探索空間の限定

GenProg での個体は修正対象プログラムを数行変更したプログラムである．ここで，課題となるのが探索空間である．プログラム中の記述の変更は，変更対象記述の決定及び記述の変更方法の決定から構成する．つまり， 1 行の変更において個体がとりうる空間は変更対象記述数と記述の変更方法数の積となる．さらに，複数行の変更では個体がとりうる空間は変更対象記述数と記述の変更方法数を変更行数でべき乗したものとなる．このように個体がとりうる空間が膨大であるため，現実的な時間でのプログラム自動修正では探索の限定が必要となる．プログラムの修正は欠陥同定及び欠陥修正で構成する．欠陥同定は修正対象プログラムの記述に基づき有限である．しかし，プログラムが膨大になると同定対象も膨大となる．そこで，GenProg は欠陥同定の粒度をプログラムの行単位とする．また，プログラムの記述方法は無限である．すなわち，同定したプログラムの記述をどのように修正するかは無限の探索空間を持つ．そこで，GenProg は ” 既にプログラム中に存在する記述を用いることでプログラム中の欠陥を修正可能である “ という仮定を用いて欠陥の書き換え方法に関する探索空間を既にプログラム中に存在する記述に限定する．この仮定が成り立つ欠陥を含むプログラムの例を図 1 に示す．図 1 のプログラムに期待する動作は，引数として受けとる 2 つの 0 以上の整数 a, b に関する最大公約数の出力及びプログラムの停止である．このプログラムは入力として $a = 0$ が与えられたときに，b の値が 0 以外であったときに期待しない動作を引き起こす．$a = 0$ が与えられると図 1 のプログラムは 2 行目の if 文の条件を満たし，最大公約数として b の値を出力する．次に，プログラムの実行は while 文の条件判定を行う．b が 0 以外のとき，while 文の繰り返し処理を行う．繰り返し処理内では a, b の大小関係に基づき処理が分岐する．$a = 0, b$ が 0 以外のとき a よりも b が大きいので 8 行目の $b - a$ が実行される．$a = 0$ であるため，a, b の大小関係は $a < b$ のままであり，次の繰り返しでも同様の処理が行われる．このように，while 文の繰り返し判定に用いる変数 b の値が変わらないため，このプログラムは $a = 0, b$ が 0 以外の入力のとき停止しない．これは，期待しない動作である． この期待しない動作の原因は 3 行目の直後に return 文が

```
 1: void gcd(int a, int b){
 2:   if (a == 0) {
 3:     printf("%g¥n", b);
 4:     // return 0; can be repaired by inserting line 18
 5:   }
 6:   while (b != 0) {
 7:     if (a > b) {
 8:       a = a - b;
 9:     } else {
10:       b = b - a;
11:     }
12:   }
13:   printf("%g¥n", a);
14:   return 0;
15: }
```

図 1: 欠陥を含む最大公約数を導出するプログラム

抜けていることである．この文は 14 行目に存在するため，14 行目の文を 4 行目に複製することで図 1 の欠陥を修正できる．図 1 では，プログラム記述の漏れがソフトウェアの故障の原因となっている．このように，プログラム中に記述漏れがある場合，記述漏れがある行の直前の行が欠陥であるとみなすことが多い [16]．つまり，図 1 の例では，3 行目が欠陥となる．GenProg は個体の生成に関する操作を抽象構文木上のノードを操作により行う．GenProg は欠陥同定を set_weights 関数で行うが，GenProg において欠陥同定と遺伝的アルゴリズム部分は独立しているので異なる欠陥同定手法を用いることも可能である．

2.3.1 生存個体の選択

GenProg では次世代に生存する個体の選択に SUS（stochastic universal sampling）を用いる [17]．SUS では variant_i の生存確率は各個体の評価値を計算する関数 fitness を用いて式（1）で計算される．

$$prob_i = \frac{fitness(variant_i)}{\sum_{i=1}^{\text{pop\_size}} fitness(variant_i)} \tag{1}$$

Algorithm 1 の 9 行目で用いられる sample 関数は各個体の生存確率に基づき，pop_size/2 個の個体を選択し，$Viable$ に格納する．$Viable$ 中の個体は交叉に用いられた後，突然変異の操作を適用される．

2.3.2 個体の交叉

個体の交叉は 2 つの個体の重み付きパス中に存在する抽象構文木の前半ノードと後半ノードを入れ替える．2 つのプログラムの交叉後は，交叉のもととなるプログラム及び交叉により生成される新しい 2 つのプログラムが次世代に生存する．2 に交叉のアルゴリズムを示す．

2.3.3 個体の変異

GenProg はプログラムの変更で追加，削除及び入れ換えの 3 つの操作を用いる．ある個体に関する変異は 1 箇所について行われる．変異を行う箇所は疑惑値に基づき決定する．

追加操作は変異対象箇所の直後にプログラム中に既存の記述を追加する．

削除操作は変異対象箇所を削除する．

入れ替え操作は変異対象箇所をプログラム中に既存の記述で入れ替える．

Algorithm 2 : The Crossover Operation of GenProg

Require: : Parent variant $Parent_A$ and $Parent_B$

Require: : Paths $Path_A$ and $Path_B$

Ensure: : Child variant $Child_C$ and $Child_D$

1: $cutoff \leftarrow \text{choose}(|Path_A|)$
2: $Child_C, Path_C \leftarrow \text{copy}(Parent_A, Path_A)$
3: $Child_D, Path_D \leftarrow \text{copy}(Parent_B, Path_B)$
4: **for** $i = 1$ *to* $|Path_A|$ **do**
5: **if** $i > cutoff$ **then**
6: $prob \leftarrow \text{crossover}_\text{p}\text{rob}(Path_A[i])$
7: **end if**
8: **if** rand(0, 1)$\leq prob$ **then**
9: $Path_C[i] \leftarrow Path_B[i]$
10: $Path_D[i] \leftarrow Path_A[i]$
11: **end if**
12: **end for**
13: **return** minimize$(V, P, PosT, NegT)$

2.3.4 GenProg の改良

Algorithm 3 : The Improved GenProg Algorithm

Require: : Faulty program P

Require: : Testsuite T including set of positive/negative testcases $PosT$ and $NegT$

Require: : Mutation operator Mutate

Require: : Crossover operator Crossover

Require: : Full fitness predicate FullFitness

Require: : Sampled fitness SampleFit

Require: : Population size $PopSize$

Ensure: : Patch that repairs the faulty program P

1: $C_{sub} \leftarrow$ FaultLocalization(P, T)
2: $Popul \leftarrow$ initial_population$(P, \text{pop_size}, \text{C}_{\text{sub}})$
3: **repeat**
4: $Fitnesses \leftarrow$ SampleFit(Popul)
5: $Popul \leftarrow \emptyset$
6: $Parents \leftarrow$ TournSelect$(Popul, PopSize, Fitnesses)$
7: $Offsprings \leftarrow$ CrossOver$(Parents, P)$
8: $Popul \leftarrow$ Mutate$(Parents \cup Offsprings)$
9: $\langle c_1, c_2 \rangle \leftarrow$ crossover(p_1, p_2)
10: $NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
11: **until** $\exists variant \in Popul$. FullFitness$(variant) = Passed$
12: **return** minimize$(V, P, PosT, NegT)$

Goues らは GenProg における変異プログラムの表現方法，生存個体の評価関数と選択及びプログラムの変異方法を改善した [12]．Algorithm 3 に改善した GenProg のアルゴリズムを示す．

個体の表現に関する改良． GenProg はプログラムを表す抽象構文木及び重み付きパスで個体を表現する．そのため，修正対象プログラムの規模が大きくなると 1 世代がメモリ上に収まらない可能性がある．よって，Goues らは各個体を修正プログラムにたいする操作列（パッチ）で表現する．パッチはプログラム中のある記述を別の記述に置き換えるということを指示する．例えば，パッチは add(6, 15)，delete(19) のように修正 z 対象プログラムに対する操作の系列で表される．この例では，修正対象プログラムの 6 行目を 15 行目に追加したあと，19 行目を削除するという操作を示す．

生存個体の評価関数と選択に関する改良． GenProg の処理において最も時間のかかる処理は各 variant に対するテストケースの適用である．テストケースの variant に対する適用は variant の評価値を導出するために行う．variant の評価値は生存個体選択の指標となる．つまり，variant の評価値導出及び生存個体選択改善しテストケースの適用回数を減らすことで，GenProg のスケーラビリ

ティは上昇する．遺伝的なアルゴリズムは，GenProg と同様に各個体に評価値を与える処理が実行時間の大部分を占めることが多い．そこで，評価関数の適用処理を減少する方法が存在する．Goues らは NFF（Noisy Fitness Function） [18] を用いることで，評価関数の適用処理を改善した．NFF の GenProg に対する応用では，入力されたテストスイートの部分集合（サンプルテストスイート）をランダムに生成し，各 variant に適用する．各 variant の評価値はサンプルテストケースの適用結果をもとに決定する．生存個体の選択は tournament selection [19] で行う．Algorithm 3 の 6 行目が NFF 及び tournament selection の処理を示す．修正プログラムは全てのテストケースを成功する．つまり，テストケースを 1 つでも失敗する variant は修正プログラムではない．よって，全てのサンプルテストスイートを成功する variant についてのみ全テストケースを適用し正しさの検証を行う．全テストケースを成功する variant が修正プログラムである．

突然変異に関する改良．Goues らは初期の GenProg で提案された 3 つの抽象構文木に対する操作 $delete, insert, swap$ の内 $swap$ に効果が無いことを指摘した．そこで，操作の改善案として $replace$ を導入した．$replace$ はプログラム中のある行をプログラム中に存在するべつの記述で置き換える操作である．これは，プログラム中のある行を削除し，削除した行にプログラム中の記述を追加する．つまり，$Replace(i, j)$ はプログラム中のある行 i について $delete(i)$，$add(i, j)$ を連続して行うことを示す．

交叉に関する改良．個体の表現が変わったため，交叉に関する操作も変更する必要がある．個体は修正対象プログラムに対する操作系列で表現する．交叉は 2 つの操作系列を連結し，それぞれの操作系列を半分ずつ残すことで行う．例えば，2 つの操作系列 $P = \{p_1, p_2, p_3, p_4\}$ 及び $Q = \{q_1, q_2, q_3, q_4\}$ を交叉する場合，まず，操作系列の連結 $P\ to\ Q = \{p_1, p_2, p_3, p_4, q_1, q_2, q_3, q_4\}$ と $Q\ to\ P = \{q_1, q_2, q_3, q_4, p_1, p_2, p_3, p_4\}$ を作成する．次に，連結した操作系列中の各操作を 1/2 の確率で削除することで交叉処理を終える．

3　調査目的

本節では，関連研究における課題を明らかにし，課題の解決案を提示する．また，解決案の正当性証明に必要な根拠を提示し，その根拠が必要である目的を述べる．調査項目は，根拠そのものの正当性である．

3.1　関連研究の課題

前節では，コード片の再利用に基づく自動修正手法である GenProg を紹介した．再利用に基づく自動修正手法は，コード片の挿入の際，挿入候補を同プロジェクト中のコード片に限定している．この限定により，挿入候補を無限の候補から有限に削減することに成功している．しかし，対象プロジェクトの規模が増大するに伴い，挿入候補の数も増大する．挿入候補の数が多すぎると，修正が上手くいく可能性が少なくなる．バグ修正成功までの反復回数，および全体の経過時間の削減のためには，挿入候補を限定する必要がある．しかし，挿入候補を限定するためには基準が必要である．

3.2　調査項目

本研究では，挿入候補を限定する基準を提案するため，新たな基準を提案し，その基準の有効性を明らかにする．挿入候補限定の基準に向けたアイディアとして，挿入元と挿入先のコード片周辺領域の類似性を利用することを提案する．この提案は，共通する処理の片方でバグが発生した際に抜けているコードの挿入を行うというバグ修正イメージから生まれている．このバグ修正イメージを図に示す．図より着想を得て，類似しているコード領域はバグ修正を実現できるコード片を含む傾向があるのではないかと考えた．つまり，類似度が高い領域を優先的に挿入候補として採用することで，修正に成功するまでの反復回数を短縮することができるのではないかと考えた．しかしこの提案は，類似度が高ければ挿入候補として優れているという仮説に基づいている．本研究では，先述の仮説の正当性を明らかにすることを目的としている．この調査目的を達成するため，以下の研究課題を設定した．

RQ: コード片挿入先周辺の領域と挿入元周辺の領域の類似度は，挿入先周辺の領域と挿入元と無関係な領域の類似度より高くなる傾向があるか

本 RQ に回答するために，実際に行われたバグ修正の情報を用いて，類似度の比較を行う必要がある．本 RQ に対し肯定的な結果が得られた場合，類似度を用いた挿入候補の限定が有用な手法になる見込みが大きくなる．これは，GenProg を始めとした，様々な再利用に基づく手法の高速化に繋がる結果を得ることとなる．

4　調査方法

本節では，前節で上げた RQ に回答するための調査方法について述べる．また，調査方法について述べる上で必要となる定義を適宜提示する．

4.1　調査手順

本研究では，実際に行われたバグ修正の情報を必要とする．そのため，オープンソースソフトウェア（以下，OSS）のリポジトリを解析し，バグ修正の情報を手に入れる必要がある．バグ修正を行ったか否か，という情報については，コミットログを手掛かりとして情報を集める．本研究では，”fix” および ”fixed” という単語（以下，バグ修正キーワード）がコミットログ中に含まれている場合，そのコミットで追加されたコード片は全て，バグ修正に用いられたものと仮定する．この仮定を基に，バージョン管理されている OSS を調査する．

本調査における調査手順は次の通りである．

ステップ 1: 挿入コード片の抽出

ステップ 2: 類似度の算出

ステップ 3: 類似度の比較

挿入コード片の抽出では，バグ修正の情報から，どのようにして挿入コード片に関する情報を手に入れ，確保するか，その方法について述べる．類似度の算出では，類似度に関する定義を述べ，その類似度を用いた算出方法について述べる．類似度の比較では，算出して集めた類似度の集合をどのように比較し， RQ に回答し得る結果を求めるかについて述べる．

以下の説明で，各ステップにおける調査手順を示す．

ステップ 1: 挿入コード片の抽出　まず，ある開発期間中のコミットログにバグ修正キーワードが含まれているようなコミットを検索する．検索がヒットしたコミットの前後のリビジョンに注目し，前後リビジョン間のソースファイルが存在するディレクトリで，ファイル内の文字列の差分をとる．ただし，差分を取る前にソースファイルの正規化を行う．正規化とは，ソースファイルからコメント，空行，不要な空白を取り除く作業である．正規化を行うことにより，不要な情報に左右されず，正確にコード片の抽出を行うことができる．差分の結果，追加されたコード片があれば，コード片の文字列と，コード片が存在する位置（ソースファイルのパス，行番号）を保持しておく．上記作業により，リビジョン番号，コード片の文字列，コード片が存在するソースファイルのパス，行番号の組のリストを得る．

ステップ 2: 類似度の算出　本研究では，類似度の算出は 2 つの領域間を対象にする．本研究における領域の定義は次のようになる．

定義 4.1 (領域) 本研究において，領域とは，正規化済みソースコードにおける複数行に渡るコードの集まりのことを指す．領域の大きさは，ウィンドウサイズ w（整数値）により，w 行と定められる．

また，用いる類似度の指標は標準化レーベンシュタイン距離である．レーベンシュタイン距離の定義は次のようになる．

定義 4.2 (レーベンシュタイン距離) レーベンシュタイン距離は，*2* つの文字列の間で相互に編集するのにかかる最小の手数である．ある文字列に対し，文字列の挿入，削除，置換を手数 *1* で行うものとして，距離を算出する．

レーベンシュタイン距離のアルゴリズムは [20] が詳しい．レーベンシュタイン距離の最大値は長い方の文字列の長さであるため，標準化レーベンシュタイン距離 $ND(\cdot,\cdot)$ は，2 つの文字列 s_1, s_2 に対し，文字列の長さを $L(\cdot)$，レーベンシュタイン距離を $D(\cdot,\cdot)$ として，以下のように表される．

$$ND(s_1, s_2) = \frac{D(s_1, s_2)}{\max\{L(s_1), L(s_2)\}} \quad (0 \leq ND(s_1, s_2) \leq 1)$$

算出する類似度は，次の 2 種類に分類される．

類似度 A コード片挿入位置周辺の領域と挿入コード片を含む領域の類似度

類似度 B コード片挿入位置周辺の領域と挿入コード片を含まない（無関係な）領域の類似度

1 つの挿入コード片に対し，類似度 A，類似度 B の対象となる領域は複数存在する．よって，1 つの挿入コード片に対し，類似度 A のリストと類似度 B のリストを算出することとなる．上記の作業を各挿入コード片に対し繰り返し，データを保持しておく．

ステップ 3: 類似度の比較　本研究では，類似度 A の集合が，類似度 B の集合に比べ，高い類似度を持つかについて調査を行うことが目的である．（以下，類似度の集合を A 群および B 群とする．）この 2 群の比較を行うため，まずはコード片毎に収集した A 群および B 群の情報をマージする．この作業により，巨大な A 群と B 群が生成されるはずである．この 2 群を比較し，A 群が B 群より高い値を持つか，次の方法で調査する．

まず，2 群のデータに正規性があるか，正規性の検定を行う．データセットは独立しているので，1 標本コルモゴロフ・スミノフ検定を行う．検定の結果，正規分布に従う場合，次に，等分散性を検査する．等分散性は，F 検定によって調査できる．検定の結果，等分散性を有する場合，独立 2 群の比較として，スチューデントの t 検定を行う．等分散性を有しない場合は，ウェルチの t 検定を行う．また，1 標本コルモゴロフ・スミノフ検定の結果，正規性を有しない場合，マン・ホイットニーの U 検定を行う．

上記の検定の結果と，箱ひげ図などによるデータ要約により，2 群のデータにどのような差があるかを確かめる．A 群が B 群より高い値を持つという結論が得られる場合，RQ に対し肯定的に回答することができる．

4.2 調査手順の実装

本節では，前節で説明した調査手順の実装について説明する．実装では，バージョン管理システムに Subversion を用いることを想定としている．また，調査対象のプロジェクトも，Subversion を用いて管理されているものを選択している．Subversion は `svn` コマンドにより様々な機能を呼び出すことができる．また，各種機能は `svn` のサブコマンドにより呼び出すことができる．本研究では，これらの操作を Cygwin ターミナル上で行っている．また，対象とするプロジェクトのリポジトリを今後 `repos` と表記する．

ステップ 1: 挿入コード片の抽出　まず，バグ修正を行っているコミットを検索する方法について述べる．コミットログからバグ修正キーワードを検索するため，

`svn log repos`

とコマンドし，コミットログを得る．この結果に対し `grep` コマンドをかけて，バグ修正を行ったリビジョンを抽出する．

次に，

`svn export -r` リビジョン番号 `repos`

`svn export -r` 1 つ前のリビジョン番号 `repos`

とコマンドし，バグ修正前後のリビジョンをローカルに保存する．

次に，2 つのリビジョンに対しソースファイルの正規化を行う．ソースファイルの正規化には CommentRemover[21] を用いた．次に，`diff -r` コマンドを両リビジョンのルートディレクトリにかける．このコマンドにより，ディレクトリを再帰的にたどり，差分を抽出することができる．また，`-u` オプションにより，差分出力を解析しやすいユニファイド形式にしておく．

ステップ 2: 類似度の算出　類似度の算出機能は Java 上で実装し，計算結果を出力するようにした．また，同じく Java 上で，入力されたリビジョン番号に対してローカルにエクスポートされたリビジョンを解析し，各挿入コード毎に A 群と B 群のデータを生成し，csv ファイルを出力する機能を実装した．

ステップ 3: 類似度の比較　類似度の比較は，ステップ 2 で生成した csv ファイルを R 言語を用いて処理し，統計的手法で検証を行った．

表 1: 調査対象のソフトウェア

ソフトウェア	開始リビジョン	終了リビジョン	リビジョン数	行数	言語
Apache httpd	r1226340	r1542827	2,672 (98)	172,953	C
CBMC	r1965	r5185	2,978 (308)	154,077	C++
jEdit	r20729	r21996	391 (45)	119,352	Java
JabRef	r2889	r3689	433 (44)	79,657	Java

5 調査結果

5.1 調査対象

本研究ではバージョン管理システム Subversion によって管理されている OSS を対象とした．調査対象のソフトウェアは表 1 の通りである．ただし，リビジョン数の中の括弧は，バグ修正キーワードがヒットしたリビジョンの数を表している．

調査対象のソフトウェアは，C 系統の言語と Java より 2 つずつ採用した．少なくともこの 2 言語に対して，結果が言語に依存するかを見ることができる．

5.2 調査結果

本節では，表 1 のソフトウェアに対し，調査を行った結果を述べる．

箱ひげ図によるデータ要約　表 1 のソフトウェアに対する，箱ひげ図は図 2-5 のようになる．

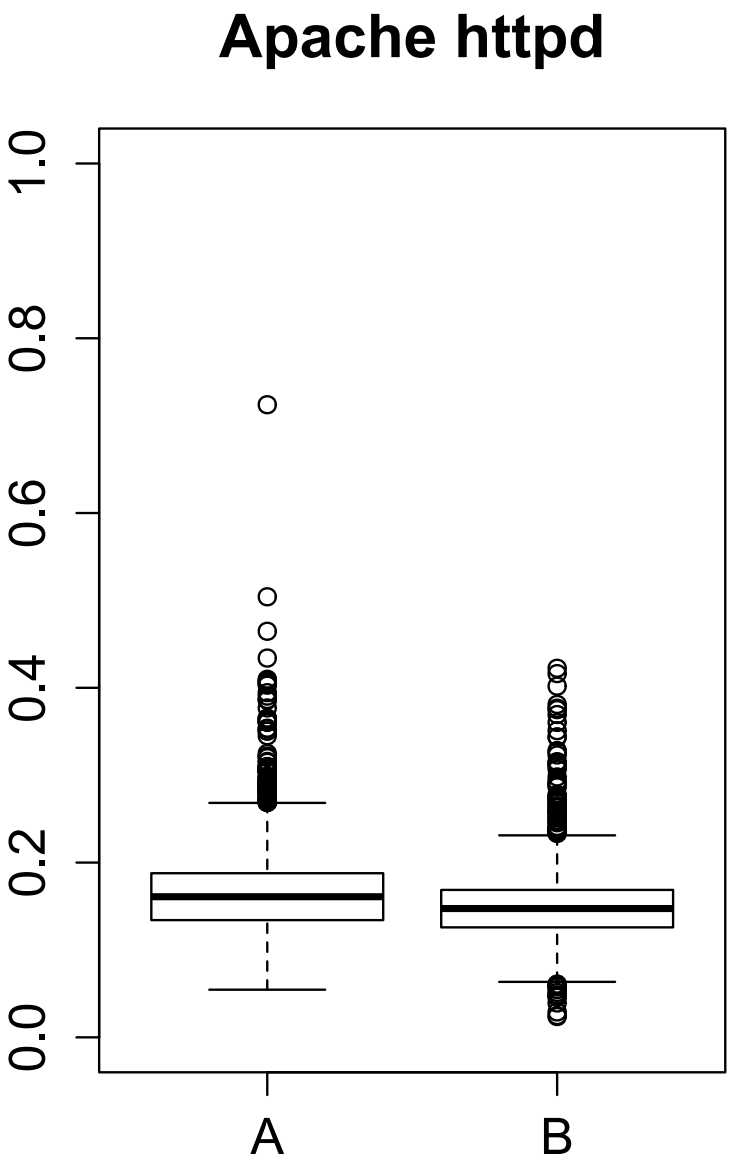

図 2: Apache httpd の箱ひげ図

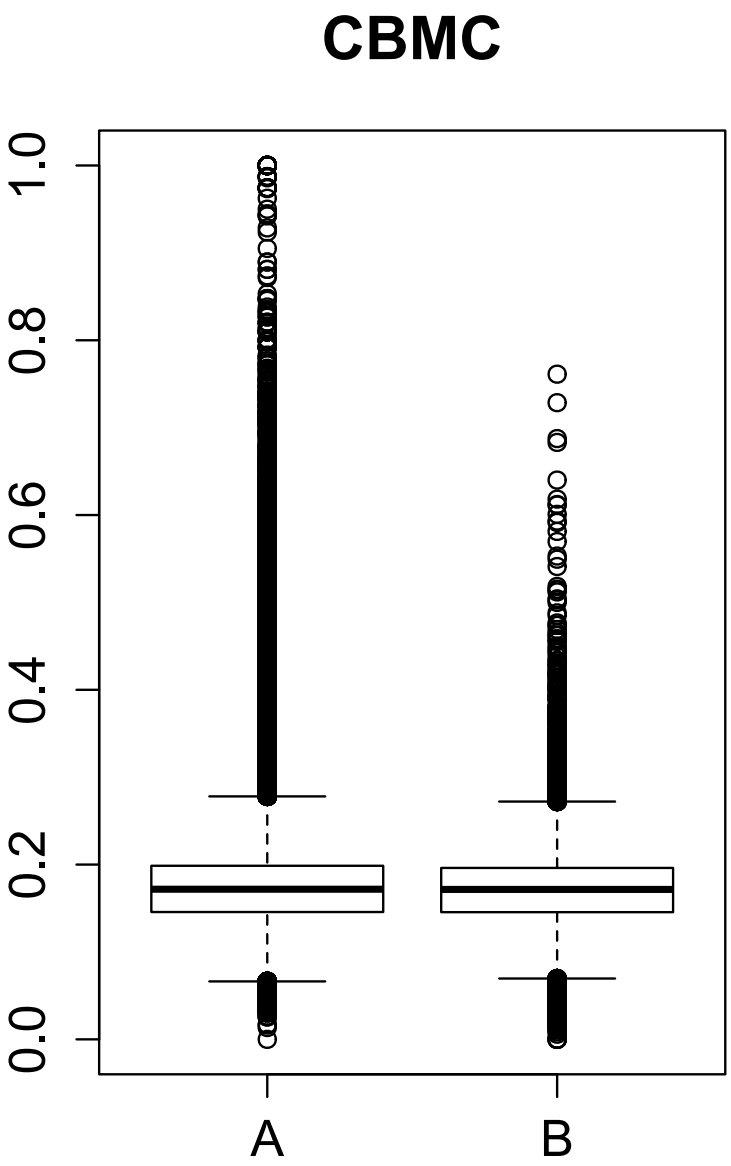

図 3: CMBC の箱ひげ図

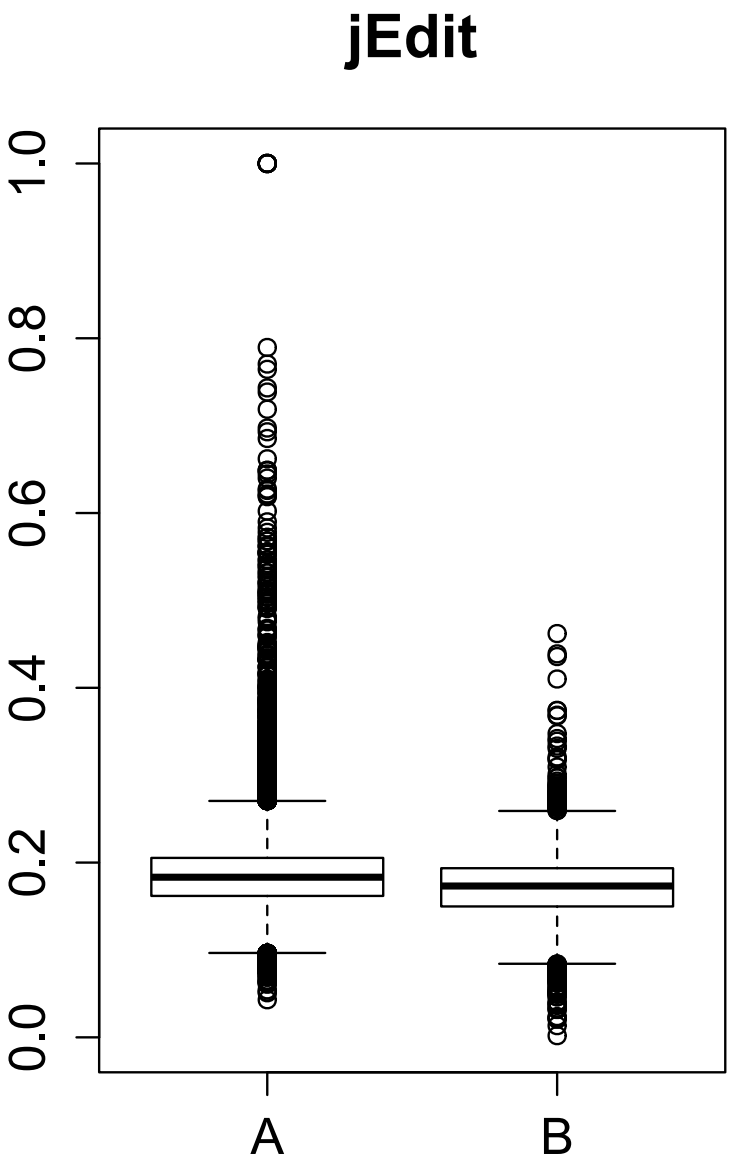

図 4: jEdit の箱ひげ図

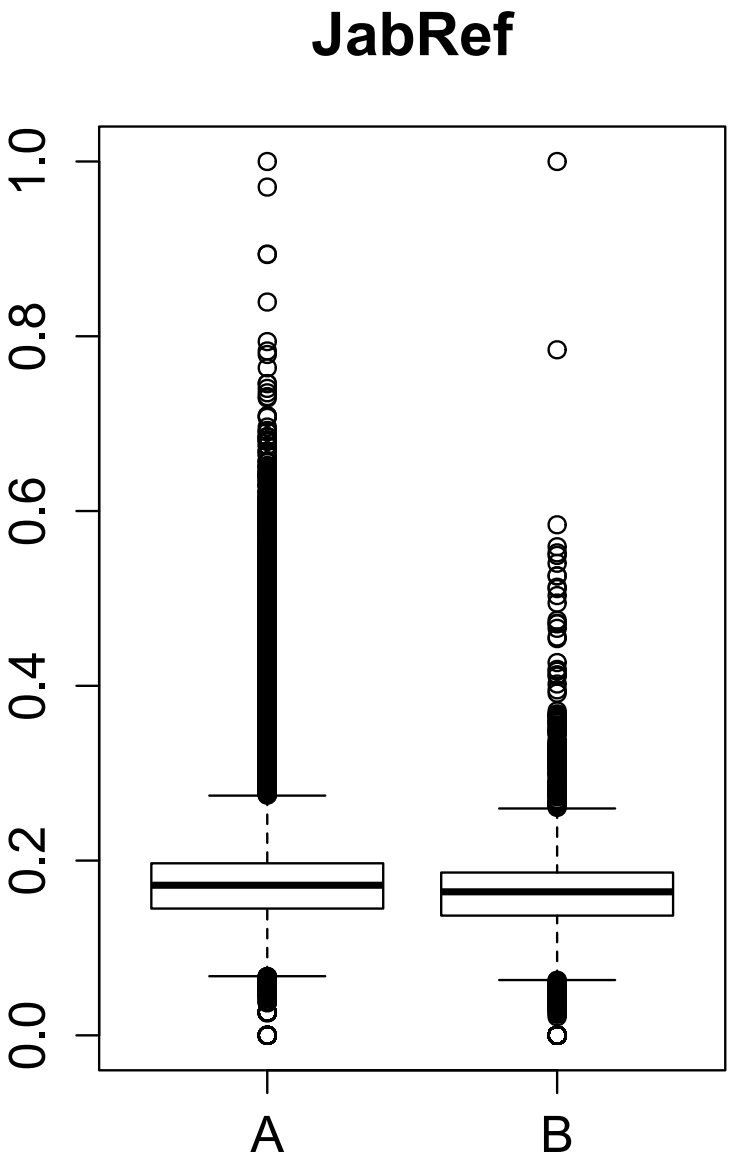

図 5: JabRef の箱ひげ図

6 考察

6.1 調査結果に対する考察

図 2-5 から，4 つのプロジェクトの全ての場合において，中央値は A 群の方が高いことがわかる．しかし，これらの情報のみでは，2 群の母平均に差があるという保証はない．そこで，統計的仮説検定を行った．まず，各プロジェクトの A 群および B 群が正規分布に従うかを検証した．1 標本コルモゴロフ・スミノフ検定により，全データにおいて，p-値が 2.2×10^{-16} 未満となり，有意水準 1 %の下で帰無仮説は棄却された．すなわち，全データに正規性はほぼ無いという結果を得た．続いて，2 群の差の検定として，マン・ホイットニーの U 検定を行った．検定の結果，各 2 群データにおいて，p-値が 4.0×10^{-11} 未満となり，有意水準 1 %の下で帰無仮説は棄却された．すなわち，各 2 群データにおいて，平均に差がないとはほぼ言えないという結果を得た．以上の結果より，A 群の母集団の類似度が B 群のものよりも高いという結論となった．上記の議論より，RQ に対して，Yes と回答することができる．

6.2 調査の妥当性について

本調査では，類似度の計算方法として，文字列のレーベンシュタイン距離を用いた．しかし，変数名が異なるだけで，機能が類似している場合に対し，高い類似度を出すことができないという欠点がある．そこで，文字列のレーベンシュタイン距離を用いるのではなく，ソースコードを字句解析し，トークン単位のレーベンシュタイン距離を求めるのが有効ではないかと考えている．

また，そもそも類似度の指標としてレーベンシュタイン距離を用いることが妥当であるかどうかも疑問である．今後の研究では，集合の類似度やレーベンシュタイン距離以外の文字列間の距離概念について調査を行い，妥当な類似度の表を明らかにしたいと考えている．

さらに，今回はコミットログに対してバグ修正キーワードを基にリビジョンを抽出しているが，この仮定が本当に有効なものであるか疑問である．コミットログに対してキーワード検索を行うよりも，バグ管理システムの情報を用いてバグ修正情報を得る方が正確ではないかと考えている．ただし，バグ管理システムを採用している OSS プロジェクトがどれだけ存在するかが疑問であり，さらなる調査が必要である．

7　あとがき

本研究では，コード片の挿入元と挿入先の周辺領域の類似性について調査を行った．

調査では，コード片挿入先周辺の領域と挿入元周辺の領域の類似度は，挿入先周辺の領域と挿入元と無関係な領域の類似度より高くなる傾向があるか，という RQ を立て，実際に 4 つのプロジェクトに対して類似度の算出と比較を行った．

調査の結果，類似度が挿入候補限定に有用であることを仮説検定により示した．

今後の課題は，研究の妥当性の詳細な検証と，類似度による限定手法を GenProg に実装し，評価することである．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました 楠本 真二 教授に心より感謝申し上げます．

本研究の全過程において，常に暖かく見守り，的確なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます．

本研究に関して，研究の方向性や有用性など，的確なご助言ご指導を頂きました 井垣 宏 特任准教授に心より感謝申し上げます．

本研究を行うにあたり，議論の中で熱いご助言ご指導を頂きました 肥後 芳樹 助教に心より感謝申し上げます．

本研究を通して，研究生活を応援して頂き，時に技術的ご指導を頂きました 堀田 圭佑 研究員，櫻井 浩子 特任研究員に心より感謝申し上げます．

本研究を通して，日常の中で相談に乗って頂き，ご助言をを頂きました 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後期課程 2 年の 村上 寛明 氏，同 楊 嘉晨 氏に深く感謝申し上げます．

本報告を行うにあたり，日常の中で声をかけて頂き，研究の基礎を一から指導して頂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 2 年の 大田 崇史 氏，同 楠 野明 氏，同 藤田 悠矢 氏，同 今里 文香 氏，同 切貫 弘之 氏，同 溝曽路 貴雅 氏に深く感謝申し上げます．

また研究室生活の中で相談に乗って頂き，また励まして頂きました 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 年の江川 翔太 氏，同 大谷 明央 氏，同 高 良多朗 氏に心より感謝申し上げます．

その他の楠本研究室の皆様のご協力に心より感謝致します．

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] J Baker. Experts battle $192bn loss to computer bugs. http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html, Feb 2012. Accessed 2015-01-27.

[2] T Britton, L Jeng, GT Carver, P Cheak, and T Katzenellenbogen. Reversible debugging software. Technical report, Univercity of Cambridge, Judge Buisiness School, 2013.

[3] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 215–222, Sept 1976.

[4] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference onSoftware Engineering, 2007.* (ICSE 2007), pp. 75–84, May 2007.

[5] Carlos Pacheco. *Directed Random Testing.* Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 2009.

[6] R. Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering, 2007.* (ICSE 2007), pp. 416–426, May 2007.

[7] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24rd International Conference on Software Engineering, 2002.* (ICSE 2002)., pp. 467–477, May 2002.

[8] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 45–55, Washington, DC, USA, 2009. IEEE Computer Society.

[9] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89–98, Sept 2007.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[12] 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*, ICSE '12, pp. 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[13] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 306–317, New York, NY, USA, 2014. ACM.

[14] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[15] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pp. 1–10, New York, NY, USA, 2002. ACM.

[16] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 467–477, New York, NY, USA, 2002. ACM.

[17] A Eiben and J Smith. *Introduction to Evolutionary computing*. Springer, 2003.

[18] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pp. 965–972, New York, NY, USA, 2010. ACM.

[19] L Miller.B and D Goldberg E. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol Comput*, Vol. 4, No. 2, pp. 113–131, 1996.

[20] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[21] Yoshiki Higo. Yoshikihigo commentremover github. https://github.com/YoshikiHigo/CommentRemover, Feb 2015. Accessed 2015-02-10.