特別研究報告

題目

開発者と **GenProg** によるソースコード修正の比較調査

指導教員

楠本　真二　教授

報告者

中島　弘貴

平成 28 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 27 年度 特別研究報告

開発者と GenProg によるソースコード修正の比較調査

中島　弘貴

内容梗概

プログラム開発において，デバッグは工数の大部分を占めるといわれている．そのため，デバッグの工数削減を目的として，デバッグの自動化に関する研究が盛んに行われている．それらの研究の中でも，遺伝的プログラミングを用いた自動プログラム修正手法である GenProg が大きな注目を集めている．

GenProg は実社会のプログラムに含まれる欠陥を修正し，その有用性を示した．しかし，GenProg によって修正されるプログラムには，欠陥の修正に直接関係しないコード片の変更や，新たな欠陥を生じさせる変更が加えられる場合があるという問題が指摘されている．この問題は，GenProg のアルゴリズムがテストケースを全て通過するプログラムを出力し，テストケースに記述されていないプログラムの動作を保障しないことに起因する．一方で多くの開発者は，プログラムが満たすべき動作内容をできるだけ損なわないように修正を行っている．そのため，開発者による修正後のプログラムでは，GenProg による修正後のプログラムが抱える問題が発生しにくい．そこで，GenProg による修正プログラムを開発者による修正プログラムに可能な限り近付けることができれば，この問題を解決できる可能性がある．しかし，開発者と GenProg による修正プログラムの差異については，これまで詳しく研究されていない．

本研究では，開発者による修正後のプログラムと GenProg による修正後のプログラムを比較することにより，それらの差異を調査した．調査の結果，開発者はプログラムに対して GenProg より多くの箇所に変更を加えること，特にプログラム文の追加を多く行うことが分かった．

主な用語

デバッグ
自動プログラム修正
コントロールフローグラフ

目 次

図 目 次

表 目 次

1 まえがき

プログラムの信頼性向上のために，デバッグは必要不可欠な作業である．デバッグとは，プログラムに含まれる故障を検出し，故障を引き起こしているプログラム中の欠陥を含む箇所を特定し，特定された欠陥を修正する作業のことである．

デバッグはプログラム開発において多大なコストがかかる作業であり，プログラム開発工程の 50%を占めるといわれている [1]．また，アメリカでは 1 年間に約 3,000 億ドルをデバッグのために費やしているともいわれている [2]．そのため，デバッグの支援を目的とした研究が数多く行われている．

デバッグを支援する方法の 1 つとして，デバッグの自動化が考えられる．デバッグを構成する 3 つの工程のうち，プログラムの故障の検出および欠陥箇所の限局に関しては，これまでに数多くの研究が行われてきている [3, 4, 5, 6, 7]．一方で，デバッグを完全に自動化するためには，欠陥の修正も自動で行う必要がある．そのため，近年ではデバッグにおける全工程の自動化を目的として，欠陥修正の自動化を含めたプログラムの自動修正手法が研究されている．

自動プログラム修正手法では，GenProg が有望視されている [8, 9]．GenProg は欠陥を含むプログラムとテストケースの集合であるテストスイートを入力とし，修正後のプログラムを出力する．GenProg は遺伝的プログラミング [10] に基づき，同一プログラム中のコード片を用いて欠陥箇所に変更を繰り返し加えることで，プログラムの自動修正を行う．出力されるプログラムは，入力として与えられたテストケースを全て通過するプログラムである．テストケースはプログラムが満たすべき動作内容を記述したものであるため，GenProg ではテストケースを全て通過するプログラムを修正が完了したプログラムとしている．しかし，GenProg によって修正されたプログラムは，欠陥の修正に直接関係しない箇所に変更が加えられたり，新たな欠陥が生じたりする場合があるという問題が指摘されている [11, 12]．一方で多くの開発者は，テストケースに記述されていないプログラムの動作を考慮した上でプログラムの修正を行っている [13]．したがって，GenProg が行うプログラム修正を，開発者が行うプログラム修正に可能な限り近付けることにより，GenProg によって修正されたプログラムにおけるこの問題を解決できる可能性がある．

本研究では，開発者が修正を行ったプログラムと GenProg が修正を行ったプログラムについて，修正箇所および修正内容の比較調査を行った．各調査ではコントロールフローグラフ [14] を用いてソースコード上での変更をグラフ上での変更に対応付け，目視で比較を行っている．本調査では，オープンソースソフトウェアに対して開発者が実際に行った修正と，GenProg が行った修正を対象とした．調査の結果，開発者の方が GenProg よりも欠陥を含むプログラムに対して多くの変更を加えること，開発者は GenProg に比べてプログラム文の追加を多く行うことなどが分かった．また，統計的仮説検定を行うことにより，開発者と GenProg におけるこれらの違いが有意なものであることを示した．さらに，GenProg による修正プログラムが抱える問題について，調査結果を用いた GenProg の改善案を提案した．

以降 2 章では，本論文において用いる技術および用語の説明を行う．3 章では本研究における調査

の目的を説明し，4 章では具体的な調査手法について説明する．5 章では調査結果について述べ，6 章では調査結果の考察および妥当性への脅威について述べる．7 章では関連研究について述べ，最後に 8 章で本研究のまとめおよび今後の課題について述べる．

2 準備

本章では，本論文で用いる技術および用語について説明する．

2.1 遺伝的プログラミング

遺伝的プログラミング [10] は，生物の進化の過程を，解の探索を行う問題に応用したアルゴリズムである．遺伝的プログラミングは解の候補となる個体の生成，個体の評価値に基づく生存個体の選択を繰り返し行うことで，解の探索を行う．遺伝的プログラミングでは，まずランダムに個体群の生成を行う．これが第 1 世代となる．次に，任意に設定した評価関数を用いて，第 1 世代に含まれる各個体の適応度を算出する．ここで，適応度の高い個体は解である可能性が高いことを示す．よって，解である可能性が高い個体を生存させるために，算出された適応度に基づいて生存個体の選択を行う．さらに，生存個体を用いて次世代の生成を行う．生存個体の選択および次世代の生成に用いられる方法は，探索を行う問題に応じて異なる．遺伝的プログラミングは，以上のように世代の生成を繰り返すことによって解の探索を行う．このアルゴリズムは解を発見する，もしくはあらかじめ定めた世代数に到達することで停止する．

遺伝的プログラミングは，遺伝的アルゴリズム [15] を拡張したものである．遺伝的アルゴリズムでは個体の表現に配列を用いているが，遺伝的プログラミングでは木構造を用いる．これにより，遺伝的アルゴリズムでは扱えない数式や，プログラムのソースコードなどを扱うことが可能である．

2.2 GenProg

GenProg [8] は，遺伝的プログラミングに基づいてプログラムの修正を自動で行う手法である．GenProg は欠陥を含むプログラムおよびテストケースの集合である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．出力はテストスイートに含まれるテストケースを全て通過するプログラムである．ここで，入力として与える欠陥を含むプログラムを**修正対象プログラム**，出力として得られる修正が完了したプログラムを**修正プログラム**と呼ぶ．また，GenProg は自動プログラム修正を行う前に，入力されたテストケースを用いて修正対象プログラムの欠陥箇所の限局を行う．GenProg では，欠陥箇所の限局を行う部分と自動プログラム修正を行う部分は独立しているため，欠陥箇所の限局に任意の手法を適用することができる．

以下では，GenProg の動作内容について述べる．GenProg の動作の流れを図 1 に示す．GenProg は欠陥箇所の限局を行った後，欠陥箇所に変異の操作を行ったプログラム（以下，個体と呼ぶ）を複数生成する．変異の操作で行うのは，以下に挙げる処理のうちの 1 つである．

挿入 欠陥を含む箇所に，修正対象プログラムに含まれるプログラム文の挿入を行う処理

削除 欠陥を含む箇所を削除する処理

置換 挿入および削除を同時に行う処理

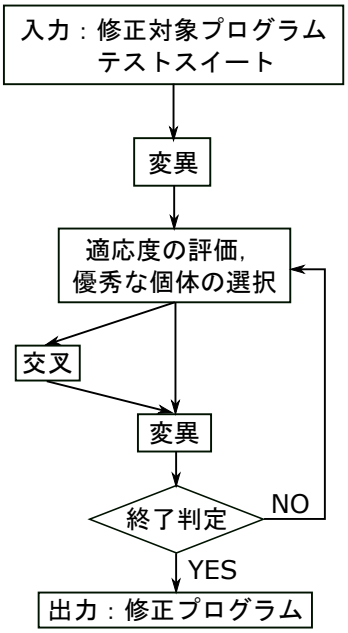


図 1: GenProg の動作の流れ

次に，評価関数に基づいて各個体の適応度の評価を行う．このとき，より多くのテストケースを通過する個体の適応度が高くなる．この適応度が高い個体から，交叉で用いる個体が選択される．選択された個体群のうち，2 つの個体を組み合わせて新たな個体群を生成する．ここで，生存する個体は交叉に用いられた個体および交叉により生成された個体である．次に，生存する個体に変異の操作を行うことで，次世代の個体群を生成する．その後終了判定を行い，修正プログラムが生成されるか，もしくは規定の世代数まで到達すると GenProg は動作を終了する．修正プログラムが生成されなかった場合は，生成された各個体の適応度の評価から以上の動作を繰り返す．

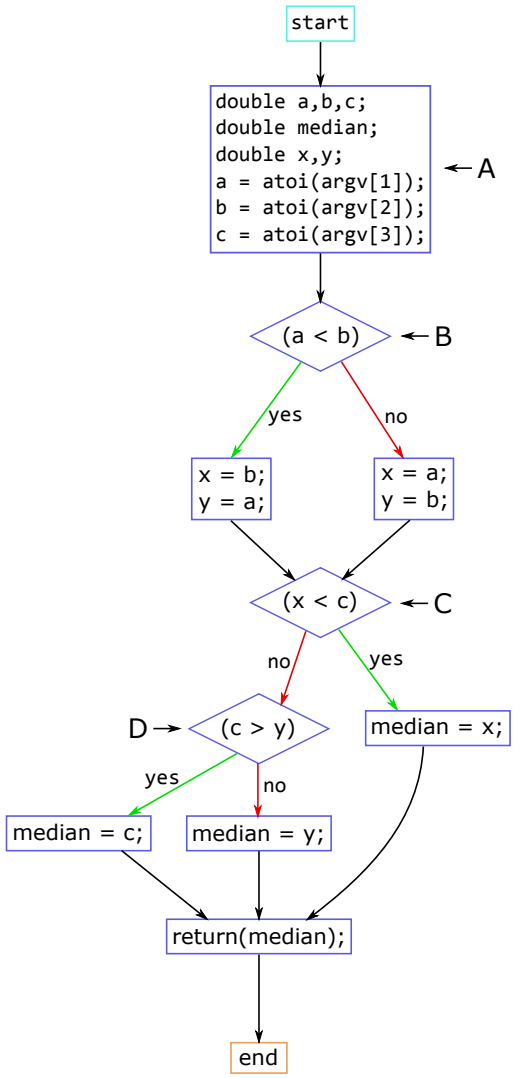
2.3 コントロールフローグラフ

コントロールフローグラフ（以下，CFG と呼ぶ）は，プログラムの制御の流れ（以下，制御構造と呼ぶ）をグラフとして表現したものである [14]．CFG の頂点は，条件文やループ文による分岐や合流を含まないコード片である．あるコード片から別のコード片への分岐あるいは合流は，頂点間の有向辺によって表現される．また，CFG はプログラム中の各関数に対して生成され，CFG の形状はその関数に含まれる分岐やループの位置および数により変化する．

CFG の例として，図 2a に与えられた 3 つの数の中央値を求めるプログラムの main 関数と，図 2b にこの main 関数の CFG を示す．この例では，図 2a の 2 行目から 8 行目までの変数宣言と代入文のみで構成されるコード片が，図 2b のグラフの「A」で示すように，1 つの頂点として表現されている

```
1  int main(int argc, char *argv[]){
2    double a, b, c;
3    double median;
4    double x, y;
5
6    a = atoi(argv[1]);
7    b = atoi(argv[2]);
8    c = atoi(argv[3]);
9
10   if (a < b){
11     x = b;
12     y = a;
13   }
14   else {
15     x = a;
16     y = b;
17   }
18
19   if (x < c)
20     median = x;
21   else if (c > y)
22     median = c;
23   else
24     median = y;
25
26   return (median);
```

(a) 中央値を求めるプログラムの main 関数



(b) 中央値を求めるプログラムの main 関数の CFG

図 2: CFG の例

ことが分かる．また，図 2a の 10 行目は if 文による分岐が含まれるため，図 2b の CFG 上では「B」で示すように，条件を表す (a < b) を 1 つの頂点として，真の場合と偽の場合の分岐を表す有向辺が 2 つ接続されていることが分かる．図 2a の 19 行目および 21 行目の if 文に関しても同様に，図 2b のグラフ上では「C」および「D」で示すように分岐を形成していることが分かる．

```
1  int med(int n1, int n2, int n3) {
2    if ((n1==n2) || (n1==n3) || (n2<n1 && n1<n3) || (n3<n1 && n1<n2))
3      return n1;
4    if ((n2==n3) || (n1<n2 && n2<n3) || (n3<n2 && n2<n1))
5      return n2;
6    if (n1<n3 && n3<n2)
7      return n3;
8  }
```

図 3: 欠陥を含む中央値を求めるプログラム

```
1  int med(int n1, int n2, int n3) {
2    if ((n1==n2) || (n1==n3) || (n3<n1 && n1<n2))
3      return n1;
4    if (n2<n1)
5      return n3;
6    if ((n2==n3) || (n1<n2 && n2<n3) || (n3<n2 && n2<n1))
7      return n2;
8    if (n1<n3 && n3<n2)
9      return n3;
10 }
```

図 4: GenProg による修正プログラム

3 調査目的

本章では，GenProg による修正プログラムの問題，および問題の解決のために必要な調査項目について説明する．

3.1 GenProg による修正プログラムの問題

前章では，自動プログラム修正手法である GenProg について説明した．GenProg は修正対象プログラムに対して，入力として与えられたテストスイートを全て通過するように修正を行う．しかし，GenProg によって出力された修正プログラムは，修正対象プログラムに含まれる欠陥の修正に直接関係しないコード片の変更が含まれることがある [11]．また，修正プログラムが本来満たすべき機能を損なったり，新たな欠陥が生じたりする場合があるという問題も指摘されている [12].

図 3 は IntroClass Benchmark [16] に含まれる，欠陥を含む中央値を求めるプログラムである．また，表 1 はこのプログラムの故障を検出するためのテストスイートである．2 つのテストスイートのうち，white-box tests はこのプログラムを GenProg によって修正する際に入力として与えるテストスイートであり，black-box tests は white-box tests とは独立した，GenProg に入力として与えないテストスイートである．図 3 のプログラムは white-box tests のうち `med(2,0,1)`，black-box tests のうち

`med(8,2,6)` において値が出力されず，正しく動作しない．

図 4 は IntroClass Benchmark に含まれる，white-box tests を用いて GenProg が修正を行ったプログラムである．このプログラムにおける修正内容は，図 3 の 2 行目にある if 文の条件式を 1 つ削除し，3 行目と 4 行目の間に if 文および return 文を追加したことである．このプログラムは white-box tests を全て通過するものの，black-box tests については `med(6,2,8)` が `8`，`med(8,6,2)` が `2` と出力され，正しく動作しない．この例から，GenProg による修正プログラムは，修正対象プログラムに含まれる欠陥の修正には成功したが，修正対象プログラムがもともと正しく行っていた動作を損なっていることが分かる．

3.2 調査項目

前節では，GenProg による修正プログラムにおける問題について説明した．一方で多くの開発者は，テストでは表現しきれないプログラムの動作を考慮した上で修正を行っている [13]．そのため開発者による修正プログラムは，GenProg による修正プログラムにおける前節で述べた問題を含む可能性が低いと考えられる．そこで本研究では，修正対象プログラムに対して開発者が行った実際の修正と，GenProg を用いて行った修正の差異を調査する．この調査結果を用いて GenProg による修正プログラムを開発者による修正プログラムに可能な限り近付けることにより，前節で述べた問題を解決できる可能性がある．

本調査を行うにあたり，以下に挙げる 2 つの研究課題を設定した．

RQ1 開発者と GenProg による各修正プログラムで，修正箇所に違いはあるか

RQ2 開発者と GenProg による各修正プログラムで，修正内容に違いはあるか

RQ1 では，開発者と GenProg が修正対象プログラムに対して行った修正の箇所についての調査を行う．もし修正を行った箇所に違いがあれば，GenProg の欠陥箇所の限局手法に改良が必要であると

表 1: テストスイート

black-box tests	white-box tests
med (2, 6, 8) = 6	med (0, 0, 0) = 0
med (2, 8, 6) = 6	med (2, 0, 1) = 1
med (6, 2, 8) = 6	med (0, 0, 1) = 0
med (6, 8, 2) = 6	med (0, 1, 0) = 0
med (8, 2, 6) = 6	med (0, 2, 1) = 1
med (8, 6, 2) = 6	med (0, 2, 3) = 2
med (9, 9, 9) = 9	

言える．もし修正を行った箇所に違いが無ければ，GenProg は欠陥箇所の限局にほぼ成功していると言える．

RQ2 では，開発者と GenProg が修正対象プログラムに対して行った修正の内容について調査を行う．もし修正の内容に違いがあれば，GenProg の自動プログラム修正のアルゴリズムに改良が必要であると言える．もし修正の内容に違いが無ければ，GenProg が行うプログラム修正と開発者が行うプログラム修正はほぼ同じであると言える．

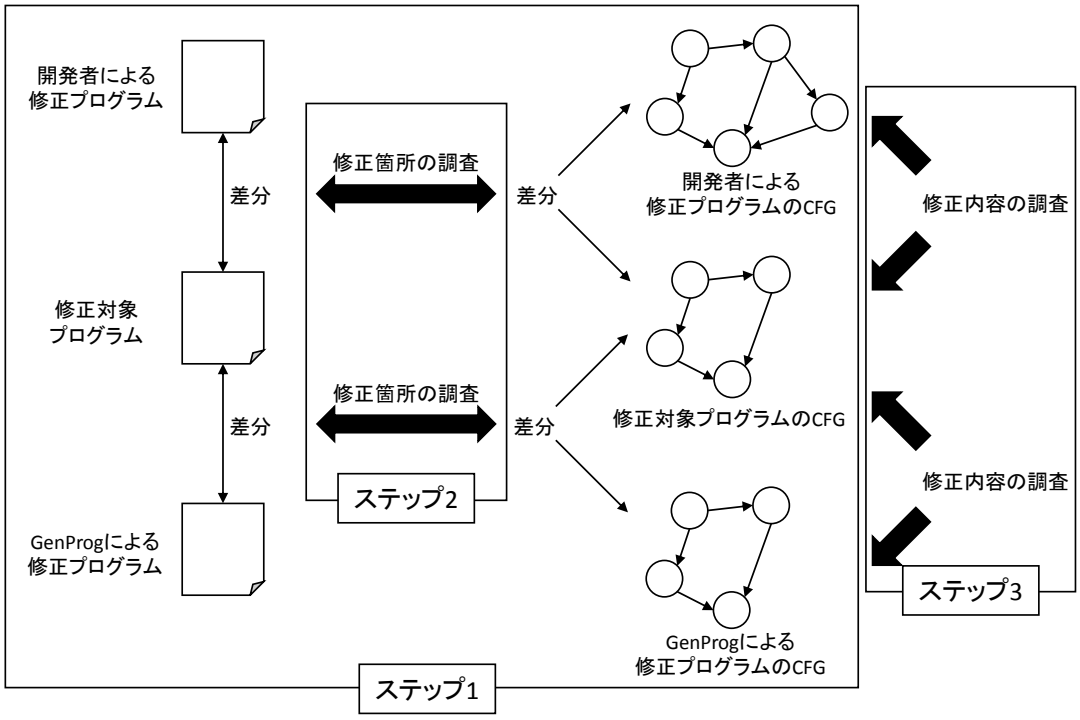


図 5: 調査の流れ

4 調査方法

本章では，前章で挙げた RQ に回答するために行った調査の方法について説明する．

4.1 調査手順

本研究では，ソースコード上での変更が関数の制御構造に与える影響について調査を行う．そこで，ソースコード上での変更を CFG 上での変更に対応付け，CFG 上で修正箇所および修正内容の調査を行う．これにより，ソースコード上での変更が関数の制御構造に与える影響を，視覚的に判断することが可能になる．また，修正内容をグラフ上での変化の種類で分別して記録することが可能になり，調査や考察を行いやすくなる．

本研究では開発者と GenProg による各欠陥修正について，調査を以下の 3 ステップで行う．

ステップ 1 ソースコード上での修正箇所の特定および CFG の作成

ステップ 2 CFG 上での修正箇所の調査

ステップ 3 CFG 上での修正内容の調査

図 5 に，本研究における調査方法の概要を示す．以下では，それぞれのステップについて詳細に述べる．

ステップ 1　ソースコード上での修正箇所の特定および CFG の作成

まず修正対象プログラムと修正プログラムのコメントを除去し，修正対象プログラムと開発者による修正プログラムについて，ソースコードの差分を抽出する．次に，ソースコード上で差分が存在する関数について，修正対象プログラムおよび開発者による修正プログラムの CFG を作成する．修正対象プログラムと GenProg による修正プログラムについても，同様にして CFG を作成する．本研究では，ソースコード解析ツールである Understand [17] を用いて CFG の作成を行った．

ステップ 2　CFG 上での修正箇所の調査

ステップ 1 で作成した各 CFG に対して，修正対象プログラムと修正プログラムのソースコード上での差分と，グラフ上で頂点数や頂点内のコード片が変化した箇所を目視により対応付ける．ソースコード上での各差分がグラフ上のどの変化に対応するかについては，自動で判別する手段が存在しないため，本調査ではこれらの対応付けを目視で行っている．

ステップ 3　CFG 上での修正内容の調査

開発者または GenProg による修正前後の関数における CFG の変化について，以下の各調査を行う．

頂点数の変化に関する調査　追加または削除された頂点の数を記録する．また，頂点数の変化の原因となるソースコード上での変更の数を，変更の種類別に記録する．頂点数の変化の原因は分岐やループに加えられる変更であるため，本調査では以下の 5 種類の変更について記録する．

- if 文の追加または削除
- while 文の追加または削除
- goto 文の追加または削除
- switch 文の追加または削除
- case 文の追加または削除

頂点内のコード片の変更に関する調査　コード片が変更された頂点の数，およびコード片の追加と削除が行われた行数を記録する．

CFG の形状の変化に関する調査　CFG の形状の変化とは，頂点数の増減，有向辺数の増減，有向辺の接続元または接続先の変化のことである．CFG 上でこれらの変化が見られた場合，その関数は制御構造が変化するコード片の変更が行われたと判断する．本調査では，これらの変化が見られた関数の数を記録する．

これらの調査を行った後，開発者と GenProg で得られた対応する各結果についてマン・ホイットニーの U 検定を行い，2 群の間に有意差があるかを検証する．

4.2　調査対象

本調査では，ManyBugs Benchmark [16] に含まれる C 言語で記述された 9 個のオープンソースソフトウェアを対象とした．各ソフトウェアの概要を表 2 に示す．ただし，kLOC はソースコードの長さを 1,000 行単位で表しており，欠陥数の中の括弧は，Le Goues らによって行われた実験 [16] において，GenProg による修正が成功した欠陥の数を表している．本調査では，GenProg による修正が成功した 83 個の欠陥について調査を行う．

表 2: 調査対象

ソフトウェア	kLOC	欠陥数	種類
fbc	97	3 (1)	コンパイラ
gmp	145	2 (0)	数学ライブラリ
gzip	491	5 (1)	データ圧縮プログラム
libtiff	77	24 (17)	グラフィックスライブラリ
lighttpd	62	9 (4)	Web サーバ
php	1,099	104 (51)	Web プログラミング言語
python	407	15 (2)	汎用プログラミング言語
valgrind	793	15 (3)	動的デバッグツール
wireshark	2,814	8 (4)	ネットワークアナライザ
合計	5,985	185 (83)	

5 調査結果

本研究では，前章で述べた 83 個の欠陥を対象として調査を行った．その結果，開発者または GenProg による修正が行われた関数は 222 個であることが分かった．そのうち，開発者が修正を行った関数の数は 143 個，GenProg が修正を行った関数の数は 111 個であった．また，開発者および GenProg が共に修正を行った関数は 32 個であった．以下では，これらの関数に対して行われた具体的な修正内容について説明する．また，表 3 に今回の調査結果の概略をまとめる．

表 3: 調査結果の概略

調査内容	頂点の追加数	頂点の削除数
調査結果	開発者の方が多い	有意差は無い
調査内容	if 文の追加数	if 文の削除数
調査結果	開発者の方が多い	GenProg の方が多い
調査内容	while 文の追加数	while 文の削除数
調査結果	開発者の方が多い	有意差は無い
調査内容	goto 文の追加数	goto 文の削除数
調査結果	開発者の方が多い	有意差は無い
調査内容	switch 文の追加数	switch 文の削除数
調査結果	開発者の方が多い	開発者の方が多い
調査内容	case 文の追加数	case 文の削除数
調査結果	開発者の方が多い	開発者の方が多い
調査内容	形状が変化した CFG の数	コード片の変更数
調査結果	開発者の方が多い	開発者の方が多い
調査内容	コード片の追加行数	コード片の削除行数
調査結果	開発者の方が多い	開発者の方が多い

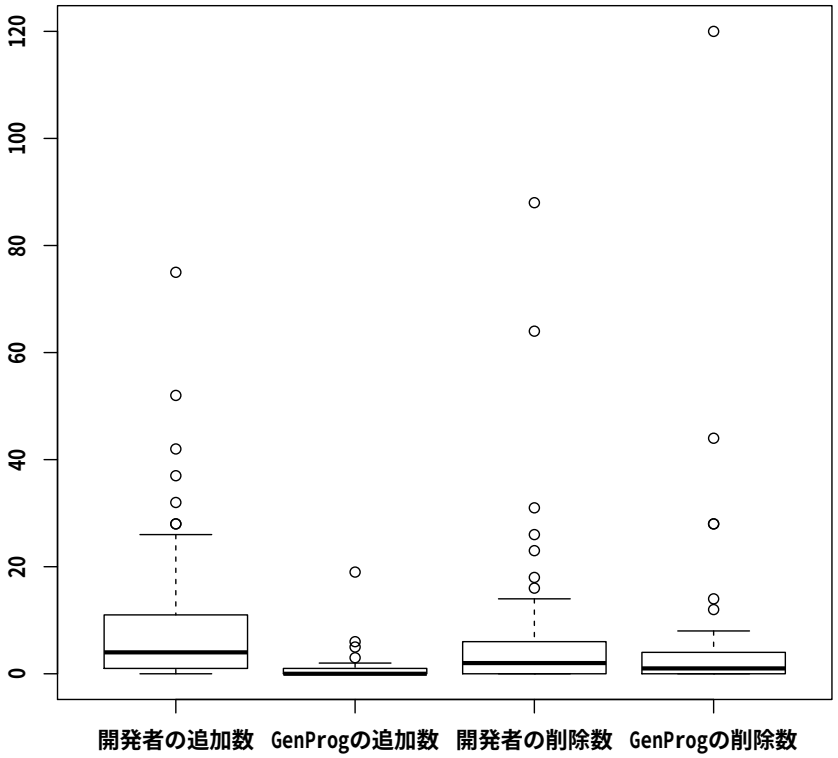


図 6: 頂点の追加数および削除数

CFG の頂点数の変化について　図 6 は，開発者および GenProg が 1 つの CFG に対して頂点の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 105 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 59 個についてこれらの変化が見られた．

- 頂点の追加数について，中央値は開発者の方が大きいという結果が得られた．また，検定を行った結果，p 値が 3.203×10^{-10} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．
- 頂点の削除数について，中央値は開発者の方が大きいという結果が得られた．また，検定を行った結果，p 値が 0.8823 となり，有意水準 5%の下で帰無仮説は棄却されなかった．すなわち，この 2 群の平均には差が無いという結果を得た．

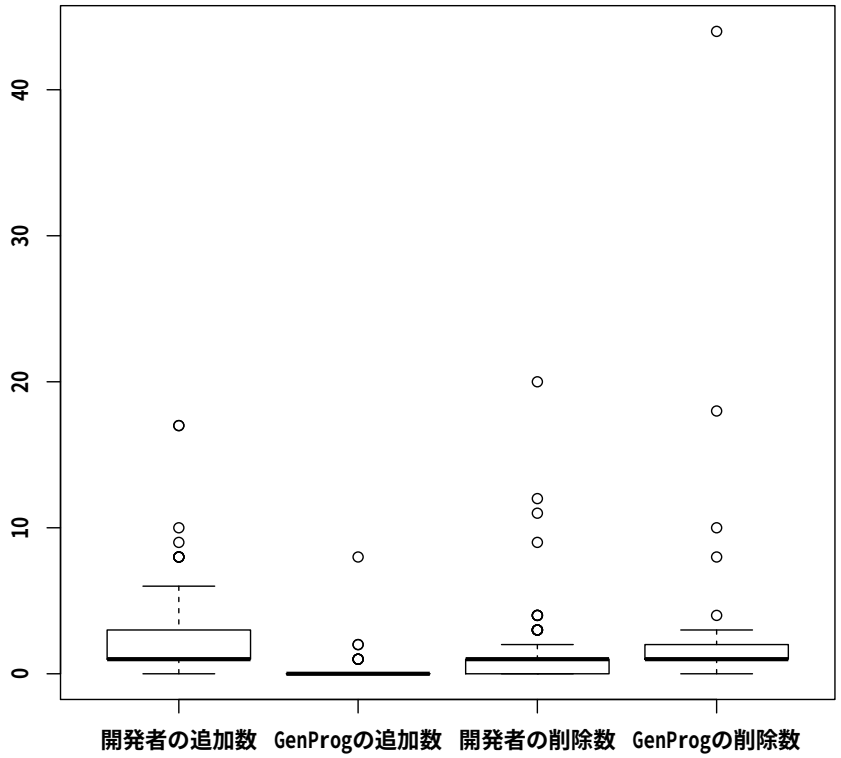


図 7: if 文の追加数および削除数

if 文の追加数および削除数について　図 7 は，開発者および GenProg が 1 つの CFG に対して if 文の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 90 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 33 個についてこれらの変化が見られた．

- if 文の追加数について，中央値は開発者の方が大きいという結果が得られた．また，検定を行った結果，p 値が 3.742×10^{-7} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．
- if 文の削除数について，中央値は開発者と GenProg で同じであるという結果が得られた．また，検定を行った結果，p 値が 0.03851 となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．

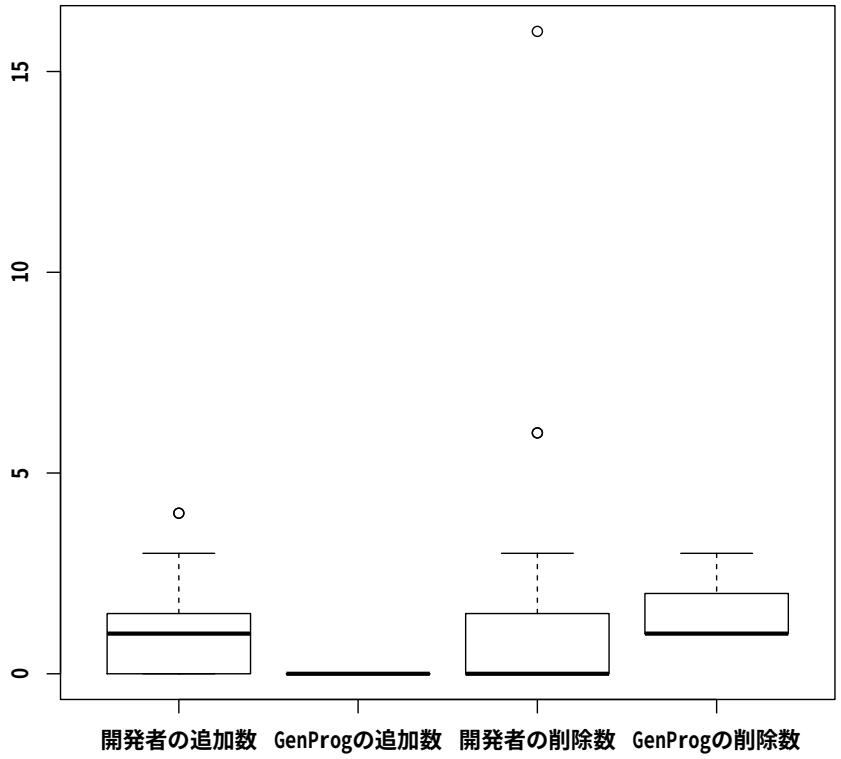


図 8: while 文の追加数および削除数

while 文の追加数および削除数について　図 8 は，開発者および GenProg が 1 つの CFG に対して while 文の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 23 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 9 個についてこれらの変化が見られた．

- while 文の追加数について，中央値は開発者の方が大きいという結果が得られた．また，検定を行った結果，p 値が 1.871×10^{-3} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．
- while 文の削除数について，中央値は GenProg の方が大きいという結果が得られた．また，検定を行った結果，p 値が 0.06233 となり，有意水準 5%の下で帰無仮説は棄却されなかった．すなわち，この 2 群の平均には差が無いという結果を得た．

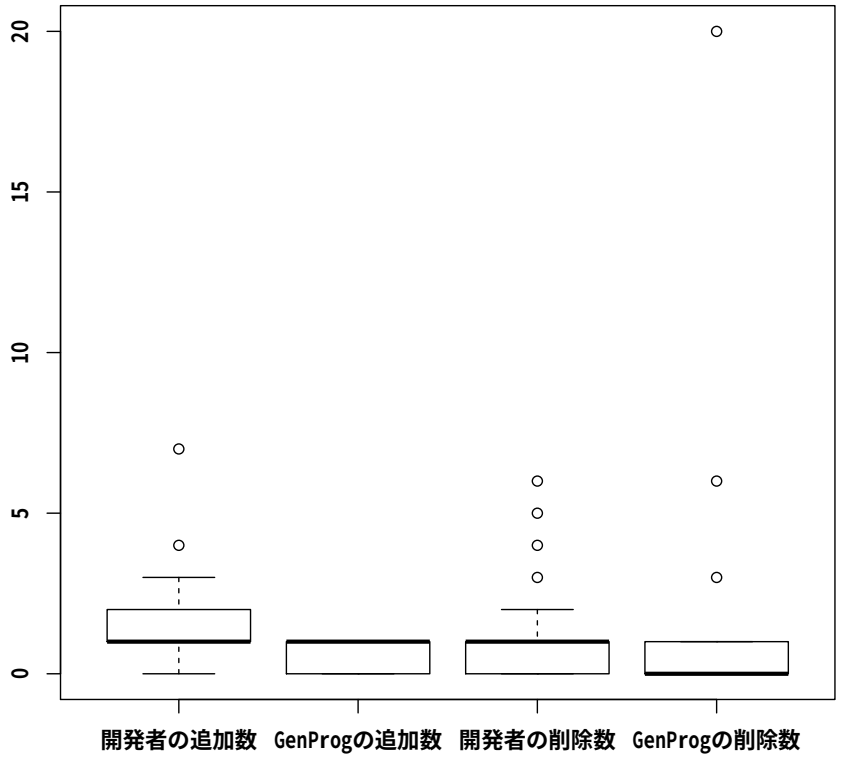


図 9: goto 文の追加数および削除数

goto 文の追加数および削除数について　図 9 は，開発者および GenProg が 1 つの CFG に対して goto 文の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 40 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 30 個についてこれらの変化が見られた．

- goto 文の追加数について，中央値は開発者と GenProg で同じであるという結果が得られた．また，検定を行った結果，p 値が 1.600×10^{-3} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．
- goto 文の削除数について，中央値は開発者の方が大きいという結果が得られた．また，検定を行った結果，p 値が 0.3566 となり，有意水準 5%の下で帰無仮説は棄却されなかった．すなわち，この 2 群の平均には差が無いという結果を得た．

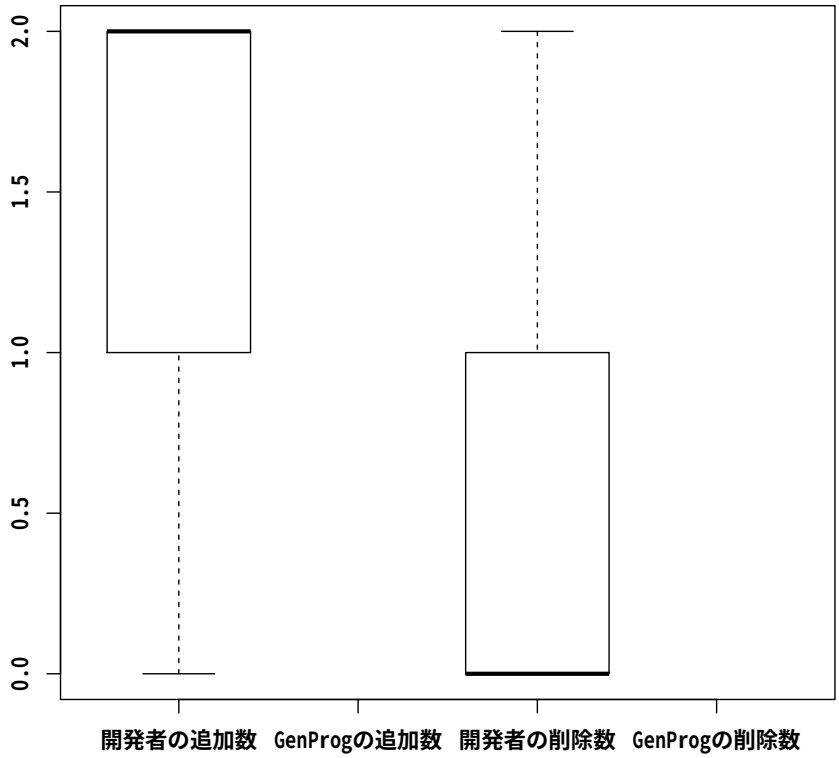


図 10: switch 文の追加数および削除数

switch 文の追加数および削除数について　図 10 は，開発者および GenProg が 1 つの CFG に対して switch 文の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 3 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行ったが，switch 文に関する変更は見られなかった．また，GenProg は switch 文の追加および削除を行っていなかったため，開発者および GenProg の結果を用いて検定を行うことはできなかった．

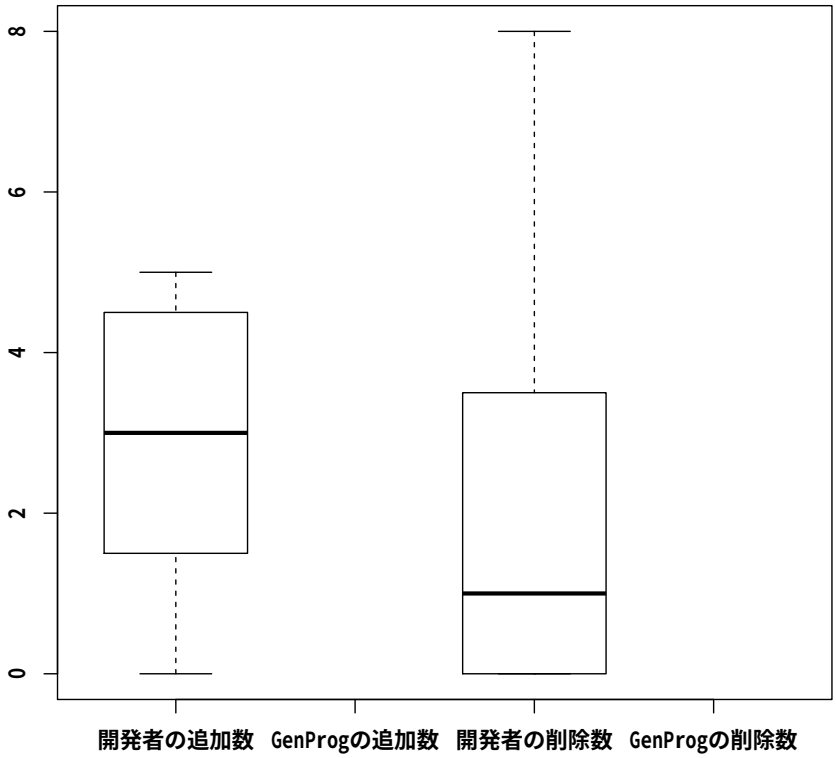


図 11: case 文の追加数および削除数

case 文の追加数および削除数について　図 11 は，開発者および GenProg が 1 つの CFG に対して case 文の追加および削除をそれぞれいくつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 7 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行ったが，case 文に関する変更は見られなかった．また，GenProg は case 文の追加および削除を行っていなかったため，開発者および GenProg の結果を用いて検定を行うことはできなかった．

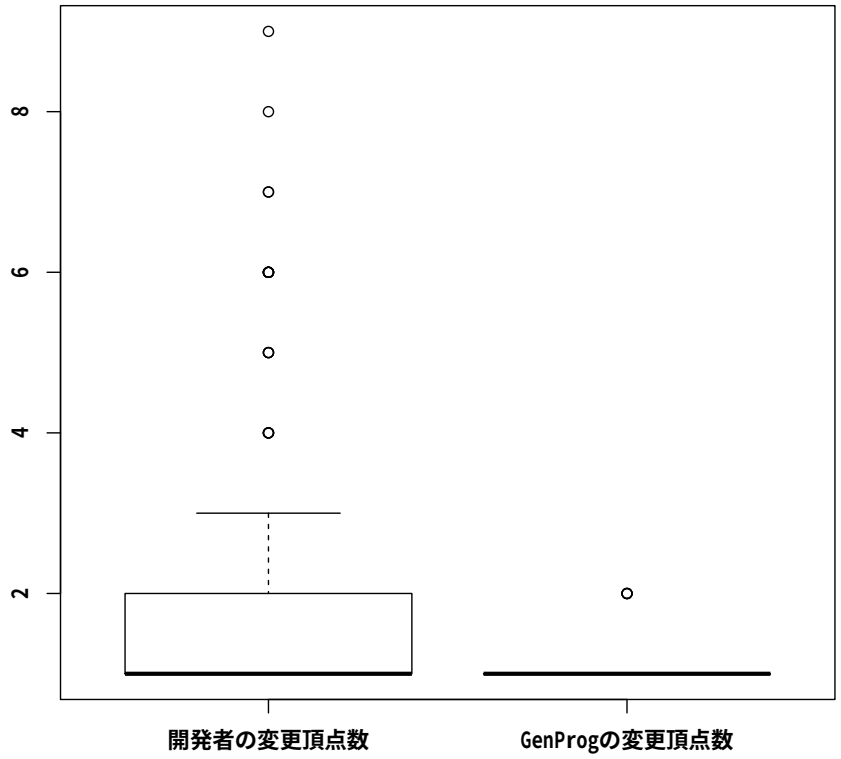


図 12: コード片が変更された頂点数

コード片が変更された頂点数について　図 12 は，開発者および GenProg が 1 つの CFG に対して，コード片の変更を行った頂点がいくつあるかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 116 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 60 個についてこれらの変化が見られた．中央値に関しては，開発者と GenProg で同じであるという結果が得られた．また，検定を行った結果，p 値が 1.093×10^{-7} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．

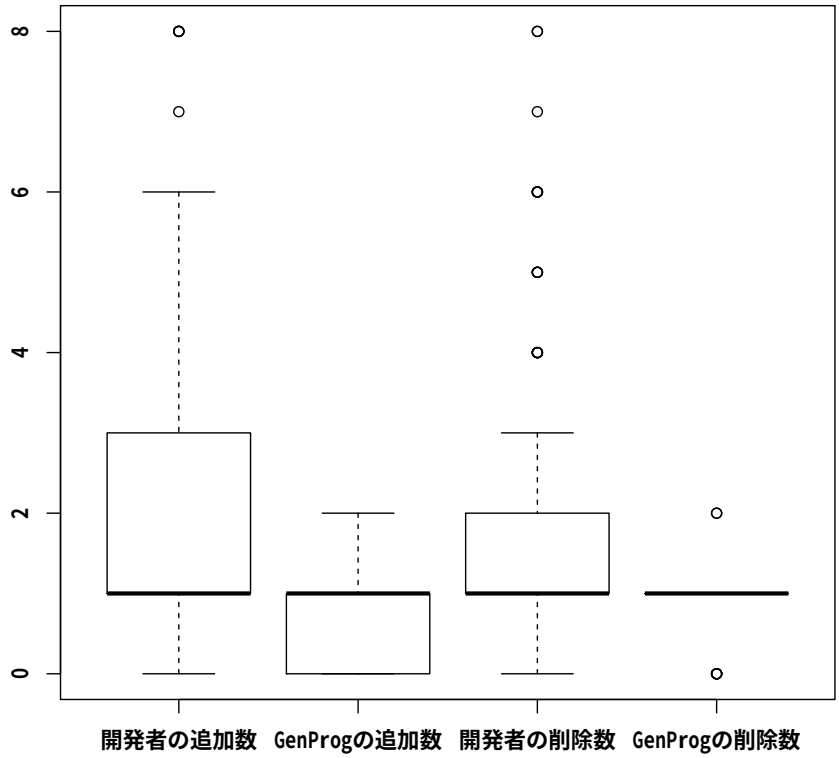


図 13: コード片の追加行数および削除行数

コード片の追加行数および削除行数について　図 13 は，開発者および GenProg が 1 つの CFG に対してコード片の追加および削除をそれぞれ何行ずつ行ったかを示した箱ひげ図である．開発者は 143 個の関数に修正を行い，そのうち 116 個についてこれらの変化が見られた．一方で GenProg は 111 個の関数に修正を行い，そのうち 60 個についてこれらの変化が見られた．

- コード片の追加行数について，中央値は開発者と GenProg で同じであるという結果が得られた．また，検定を行った結果，p 値が 1.854×10^{-10} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．
- コード片の削除行数について，中央値は開発者と GenProg で同じであるという結果が得られた．また，検定を行った結果，p 値が 1.373×10^{-4} となり，有意水準 5%の下で帰無仮説は棄却された．すなわち，この 2 群の平均に差が無いとは言えないという結果を得た．

形状が変化した CFG の数について　開発者が修正を行った 143 個の関数のうち，CFG の形状が変化した関数の数は 105 個であった．これは約 73.4%である．一方で GenProg が修正を行った 111 個の関数のうち，CFG の形状が変化した関数の数は 64 個であった．これは約 57.7%である．これらの結果から，開発者の方が GenProg よりも関数の制御構造を変化させる修正を多く行うことが分かった．

6 議論

本章では，調査結果に対する考察および妥当性への脅威について説明する．

6.1 調査結果の考察

図 6 および図 12 から，開発者は GenProg に比べて頂点の追加を多く行い，さらに頂点内のコード片の変更も多く行うことが分かる．これは，開発者は欠陥の修正を行う際に多数の箇所に変更を加えるが，GenProg は欠陥箇所として限局された少数の箇所に変更を加えていることを示している．また，頂点の削除数に関しては，開発者と GenProg の間で有意な差は見られなかった．これは，開発者と GenProg がプログラムの修正を行う際に，ほぼ同じ頻度で頂点の削除に関する変更を加えていることを示している．

図 7-11，図 13 および表 3 から，頂点の削除数，while 文の削除数，goto 文の削除数の 3 項目を除いて，開発者と GenProg で差があることが分かる．また，追加数に関しては，開発者の方が GenProg よりも多いことが分かる．これは，GenProg はプログラム文の挿入候補が同一プログラム中の既存のプログラム文に限定される一方で，開発者は同一プログラム中に存在しない新たなプログラム文の挿入が可能であるためだと考えられる．また，各項目の削除数に関しては，GenProg の方が開発者よりも多い，あるいは開発者と GenProg の間に有意差が無いという結果がいくつか見られる．これは GenProg がプログラム文の追加よりも削除を多く行うことで，欠陥の修正を行おうとする傾向があることを示している．GenProg では，欠陥の修正に必要なプログラム文が同一プログラム中に存在しない場合，修正を行うことが困難であるため，このような傾向が見られるのではないかと考えている．

以上の結果から，3.1 節で述べた GenProg の修正プログラムにおける問題の改善案として，以下の 2 つが挙げられる．

修正箇所の候補を増やす　上述のように，開発者は GenProg に比べてプログラム中の多くの箇所に修正を行っている．一方で，GenProg は修正を行う際に，一度の変更でプログラムの一箇所のみに変更を加える．そのため，GenProg による修正を行う際に一度の変更で複数箇所の変更を行うよう設定することにより，GenProg による修正プログラムの問題を改善できる可能性があると考えられる．また，この改善を行うことにより，欠陥を含む箇所を修正できる確率が上がり，GenProg による修正がより成功するようになると考えられる．

プログラム文の追加を多く行うようにする　調査結果から，開発者は GenProg に比べてプログラム文の追加を多く行っていることが明らかになった．また，特に if 文の追加および頂点内のコード片の変更を多く行っていることが分かった．そのため，GenProg による修正を行う際にこれらを優先的に行うようにすることで，GenProg による修正プログラムの問題を改善できる可能性があると考えられる．

6.2 妥当性への脅威

本調査では，調査手順のすべてのステップにおいて目視による調査が含まれている．そのため，調査結果には集中力の低下による誤った結果が含まれる可能性や，調査漏れが存在する可能性がある．本研究では各関数の調査を行う際に十分な時間を確保し，細心の注意を払って調査を行った．本調査ではコントロールフローグラフを用いることで，ソースコード上での修正がプログラムの制御構造に与える影響について調査を行った．しかし，制御構造以外の影響については調査を行っていない．データ依存関係の変化など，その他の影響に関する調査は今後の課題である．また，本調査では ManyBugs Benchmark に含まれるオープンソースソフトウェアの 83 個の欠陥に対して調査を行った．しかし，調査対象が異なれば，調査結果が変わる可能性がある．

7 関連研究

本章では，本研究に関連する既存研究について説明する．

7.1 再利用に基づく手法

Weimer らは，コード片の再利用に基づいて自動プログラム修正を行う GenProg を提案した [8]．Le Goues らは GenProg を 8 つのオープンソースソフトウェアに対して適用し，105 個の欠陥のうち 55 個の修正に成功したとして，その有用性を示した [9]．しかし，GenProg は変異により生成したプログラムの評価時に，与えられたテストケースを全て実行するため時間がかかるという問題がある．Qi らはこの問題に対して，1 つのテストケースに失敗した時点で評価を打ち切り，新たなプログラムの生成を行う RSRepair を提案した [18]．Qi らは GenProg と同じ 8 つのオープソースソフトウェアに対して RSRepair を適用し，GenProg よりも多くの欠陥を短い時間で修正できたと報告している．しかし，RSRepair は GenProg と異なり，複数のプログラム文の変更を必要とする欠陥を修正できない．

7.2 プログラム意味論に基づく手法

Nguyen らはプログラム意味論に基づいて自動プログラム修正を行う SemFix を提案した [19]．SemFix はテストスイートを用いて欠陥を含む箇所が満たすべき制約を導出し，その制約を満たすプログラム文を生成する手法である．Nguyen らは SemFix を 5 つのオープソースソフトウェアに適用し，GenProg よりも多くの欠陥を修正できたと報告している．プログラム意味論に基づく手法は修正対象プログラム中に存在しないプログラム文を生成することが可能であるが，制約を満たすプログラム文を生成する問題は NP-完全の問題であるため，制約の内容によっては現実的な時間で解けない可能性がある．

Mechtaev らはプログラム意味論に基づく手法を改良した DirectFix を提案した [20]．DirectFix は，欠陥箇所の限局と修正プログラムの生成を同時に行うことで，修正内容ができるだけ簡潔になるようにした手法である．Mechtaev らは DirectFix を SemFix と同じ 5 つのオープンソースソフトウェアに対して適用し，人間にとって理解しやすい修正プログラムを生成できたと報告している．

7.3 修正パターンに基づく手法

Kim らは修正パターンに基づいて自動プログラム修正を行う PAR を提案した [11]．PAR で用いられている 10 個の修正パターンは，開発者によって実際に行われた修正を元にして作成されたものである．Kim らは PAR を 6 つのオープンソースソフトウェアに対して適用し，GenProg よりも多くの欠陥を修正できたこと，および GenProg よりも人間にとって理解しやすい修正プログラムを生成できたことを報告している．しかし，実験対象や実験内容については批判的な意見も存在する [21]．

7.4 修正プログラムの質

Smith らは，自動プログラム修正手法による修正プログラムがテストスイートに過剰に適応する問題について調査を行った [12]．調査の結果，この問題の解決策として自動プログラム修正手法を適切な場面で用いること，テストスイートの質を測り適切に改良することなどが示された．また，現在の自動プログラム修正手法が，いくつかの場面においてプログラミング初心者より高い水準でプログラム修正を行えることが示された．

Fry らは，自動プログラム修正手法によって生成された修正プログラムの質について，保守性の観点から調査を行った [22]．この調査では，自動プログラム修正手法による修正プログラムは開発者による修正プログラムに比べて保守性が低いという結果が得られたが，適切な説明を掲載することで開発者による修正プログラムよりも保守性が高くなることが示されている．

8 あとがき

本研究では，開発者と GenProg による修正プログラムについて，コントロールフローグラフを用いて修正箇所および修正内容の比較調査を行った．修正箇所については，開発者の方が GenProg よりも多いという結果が得られた．また，修正内容については，開発者は GenProg に比べて制御構造を変化させる修正を多く行い，特に if 文の追加などを多く行うということが分かった．

今後の課題としては，今回の調査結果を用いた GenProg の改良およびその評価が考えられる．たとえば，GenProg が修正対象プログラムに対してプログラム文の追加を多く行うようにする，修正を行う箇所の候補を増やすなどである．また，本研究では GenProg に関してのみ調査を行ったが，他の自動プログラム修正手法に対する調査も今後の課題である．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に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，終始熱心かつ丁寧なご指導を頂きました肥後 芳樹 准教授に深く感謝申し上げます．

本研究に関して，有益かつ的確なご助言ご指導を頂きました柗本 真佑 助教に心より感謝申し上げます．

本研究を行うにあたり，日常の中で声をかけて頂き，ご指導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 3 年の村上 寛明 氏，同 楊 嘉晨 氏に深く感謝申し上げます．

本研究を通して，日常の中で声をかけて頂き，的確なご助言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前期課程 2 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．

本研究を進めるにあたり，日常の中で相談に乗って頂き，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 1 年の小倉 直徒 氏，同 佐飛 祐介 氏，同 鷲見 創一 氏，同 古田 雄基 氏，同 幸 佑亮 氏，同 横山 晴樹 氏に深く感謝申し上げます．

本研究の全過程において，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に心より感謝申し上げます．

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，この場を借りて心より感謝申し上げます．

参考文献

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

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

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, pp. 215–222, 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 on Software Engineering*, pp. 75–84, 2007.

[5] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 273–282, 2005.

[6] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 45–55, 2009.

[7] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. Mimic: Locating and understanding bugs by analyzing mimicked executions. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pp. 815–826, 2014.

[8] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 364–374, 2009.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and 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*, pp. 3–13, 2012.

[10] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

[11] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 802–811, 2013.

[12] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[13] Yalin Ke, Kathryn T. S., C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *Proceedings of the 30th International Conference on Automated Software Engineering*, pp. 295–306, 2015.

[14] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, pp. 1–19, 1970.

[15] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[16] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. In *IEEE Transactions on Software Engineering*, pp. 1236–1256, 2015.

[17] Understand. https://www.techmatrix.co.jp/quality/understand/.

[18] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 772–781, 2013.

[20] S. Mechtaev, Jooyong Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, pp. 448–458, 2015.

[21] M. Monperrus. A critical review of ”automatic patch generation learned from human-written patches”: Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 234–242, 2014.

[22] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 10th International Symposium on Software Testing and Analysis*, pp. 177–187, 2012.