社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 開発者と自動プログラム修正手法による修正の差異の調査

中島　弘貴† 　横山　晴樹† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{h-nakajm,y-haruki,higo,kusumoto}@ist.osaka-u.ac.jp

**あらまし**　近年，プログラム中の欠陥の特定および修正を自動的に行う手法が数多く提案されている．自動プログラム修正手法は，欠陥を含むプログラムとテストスイートを入力として，プログラムの修正とテストの実行を繰り返し，全てのテストを通過するプログラムを出力する手法である．しかし，これらの手法は欠陥修正に直接関係しないコード片の変更や，新たな欠陥を発生させてしまう変更を行う可能性がある．一方で，多くの開発者は既存の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．したがって，自動プログラム修正手法における上記の問題を改善するためには，プログラムに対して開発者が実際に行った修正と，自動プログラム修正手法を用いて行った修正の差異を調査する必要がある．本研究では，コントロールフローグラフを用いたプログラムの修正箇所および修正方法の比較手法を提案し，9 個のオープンソースソフトウェアの欠陥修正履歴を対象に，これらの手法を用いて調査を行った．調査の結果，開発者は自動プログラム修正手法に比べてプログラムの制御構造を変化させる修正を多く行うこと，開発者は自動プログラム修正手法よりもプログラム文の追加を多く行うことなどが明らかになった．
**キーワード**　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

## 1. ま え が き

ソフトウェア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．デバッグはソフトウェア開発において多大なコストがかかる作業であり，ソフトウェア開発工程の 50%を占めると言われている [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言われている [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．これまでの研究では，欠陥箇所の限局を自動で行う手法が数多く提案されてきた [3], [4]．しかし，デバッグを完全に自動化するためには，欠陥箇所の限局だけでなく，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自動プログラム修正手法が研究されている．

自動プログラム修正手法の 1 つとして，遺伝的プログラミングに基づく手法である GenProg がある [5]．GenProg は入力として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与えると，欠陥を修正したプログラムを出力する．GenProg では欠陥を含むプログラムが与えられると，まずテストスイートを用いて欠陥箇所の限局が行われる．次に，欠陥を含む可能性の高いソースコード行に対して，プログラム文の挿入，削除，置換といった変更を加える．これらの変更は，次のような処理である．

**挿入**　欠陥を含む可能性がある行の次の行に，プログラム文の挿入を行う操作
**削除**　欠陥を含む可能性がある行を削除する操作
**置換**　挿入および削除を同時に行う操作

これらの変更を加えられたプログラムは，テストケースを実行することにより修正が完了したかどうか検証される．全てのテストケースを通過すれば GenProg による修正は完了するが，テストケースを 1 つでも通過しない場合は，修正が完了するまで上記の変更を繰り返し行う．

GenProg 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を修正プログラムとして出力する．しかし，テストケースに記述されていないプログラムの動作について，修正後のプログラムが正しく動作する保障はない．つまり，たとえ GenProg による修正が成功したとしても，修正後のプログラムには新たな欠陥が含まれている可能性がある．実際，GenProg による修正プログラムには，本来満たすべき機能が損なわれる場合や，新たな欠陥が発生する場合があるという指摘もある [6]. 一方で，多くの開発者はテストケースで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 [7]．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上記の問題を含む可能性が低いと考えられる．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により，上記の問題を改善でき

る可能性がある．この改善を行うためには，開発者が行う修正と GenProg が行う修正の差異を調査する必要がある．

本研究では，開発者と GenProg による各修正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．調査の結果，開発者は自動プログラム修正手法に比べてプログラムの制御構造を変化させる修正を多く行うこと，開発者は自動プログラム修正手法よりもプログラム文の追加を多く行うことなどが明らかになった．

## 2. 調 査 目 的

本章では GenProg における課題を示し，課題解決に向けた調査動機および調査項目について説明する．

### 2.1 GenProg の課題

GenProg は 8 つのオープンソースソフトウェアに対して適用され，105 個中 55 個の欠陥の修正に成功したとして，その有用性が示された [8]．しかし，GenProg は与えられた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みなしている．すなわち，GenProg による修正後のプログラムでは，ソースコードの可読性や保守性などが考慮されていない．実際，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と指摘されている [9]．また，修正プログラムがテストスイートに過剰に適応することにより，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ことも指摘されている [6]．

### 2.2 調 査 動 機

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，多くの開発者は既存のテストケース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 [7]．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上記の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，欠陥を含むプログラムに対して開発者が行った修正と，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，上記の問題を解決できる可能性がある．

### 2.3 調 査 項 目

本研究では，開発者による修正プログラムと GenProg による修正プログラムの差異を明らかにすることを目指す．そのため，以下の調査項目を設定した．

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

**RQ2** 開発者と GenProg による修正プログラムで，修正方法に違いはあるか

## 3. 調 査 方 法

本章では調査の方法および調査に用いる技術について説明する．

### 3.1 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）とは，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．CFG の頂点は，分岐や合流を 1 つも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．

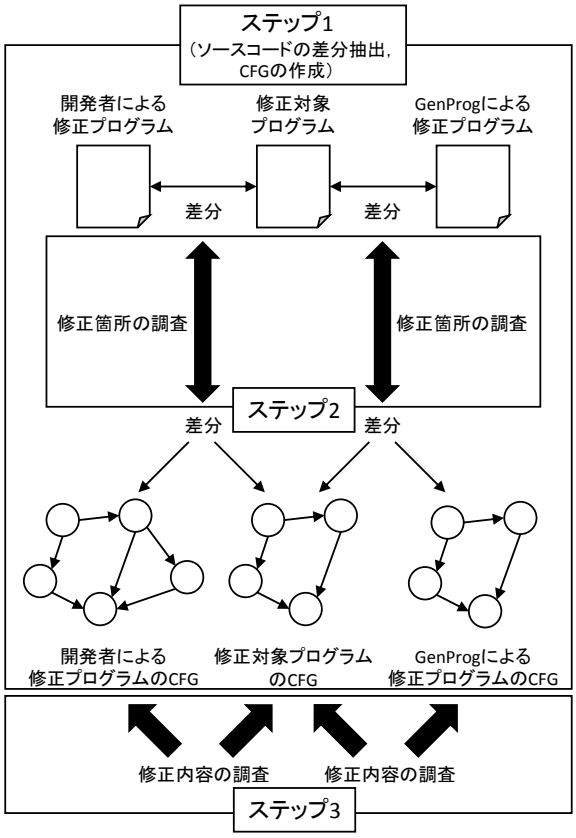


図 1 調査の流れ

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．

### 3.2 調 査 手 順

本研究では開発者と GenProg による各欠陥修正に対して，調査を図 1 のように以下の 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

以下，上記の 3 ステップで行う処理を詳細に述べる．

ステップ 1 では，まず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それぞれで CFG

表 1　調 査 結 果

| 調査内容 | 頂点の追加数 | 頂点の削除数 | コード片の変更数 | 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 | 有意差は無い | 開発者の方が多い | 開発者の方が多い | 開発者の方が多い |
| 調査内容 | if 文の追加数 | if 文の削除数 | while 文の追加数 | while 文の削除数 | CFG の形状変化数 |
| 調査結果 | 開発者の方が多い | GenProg の方が多い | 開発者の方が多い | 有意差は無い | 開発者の方が多い |

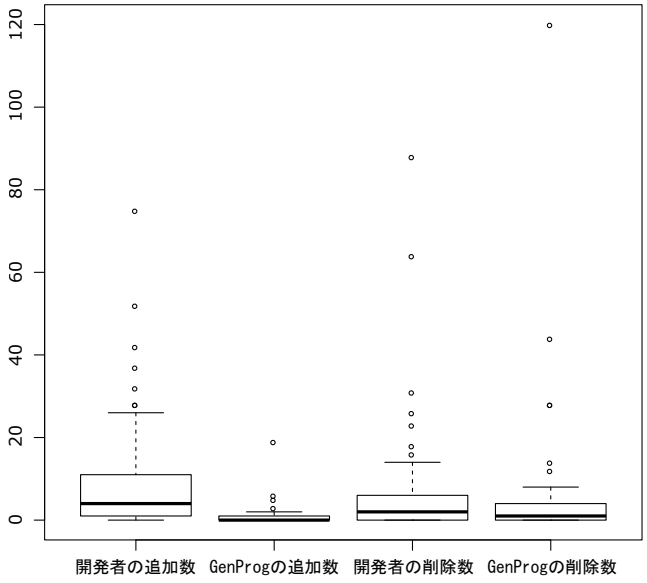

図 2　頂点の追加・削除数

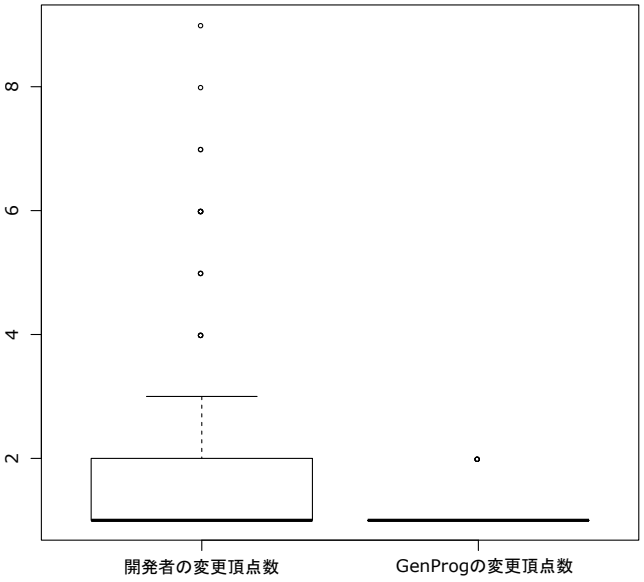

図 3　コード片が変更された頂点数

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．

ステップ 2 では，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で対応付ける．ソースコード上での各差分がグラフ上で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 では，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．

a） 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 2 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

b） 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．

c） 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ような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 の対応するデータ同士について，マン・ホイットニーの U 検定を用いて有意差の有無を確認する．

## 4. 調　　査

本章では，調査対象および調査結果について説明する．

### 4.1 調 査 対 象

本調査では，ManyBugs Benchmark [10] 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．それぞれのソフトウェアの概要は表 2 の通りである．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，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) | |

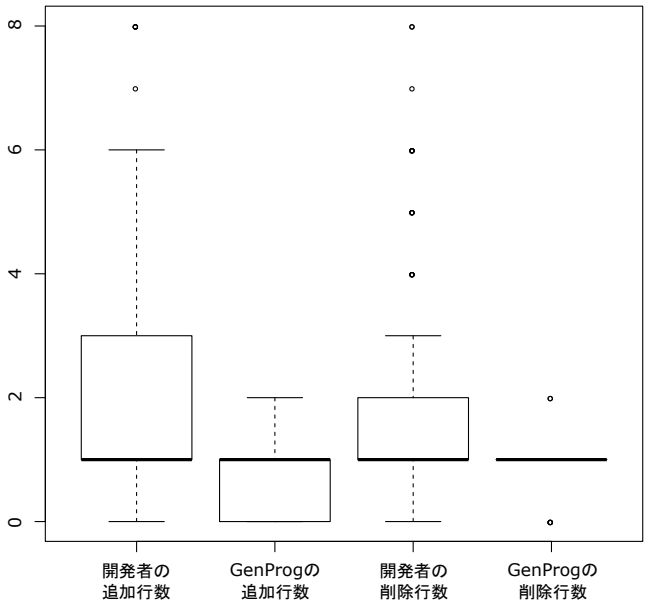

図 4　コード片の変更行数

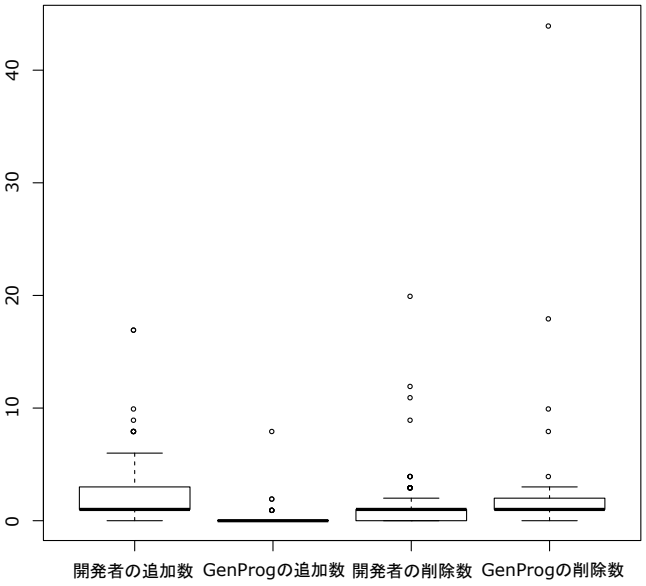

図 5　if 文の追加・削除数

### 4.2 調 査 結 果

表 2 の 83 個の欠陥に対し調査を行った結果，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は 143 個，GenProg が修正を行った関数は 111 個，開発者と GenProg が共に修正を行った関数は 32 個であった．表 1 に，本調査の結果についてまとめる．以下では，これらの関数に対して行われた修正について具体的に説明する．

**頂点の追加・削除数について**

図 2 は，開発者および GenProg が 1 つの関数の 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す箱ひげ図である．データ数は開発者が 105，GenProg が 59 である．

- 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，有意水準 5%の下で頂点の追加数については有意差があるということが分かった．
- 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，有意水準 5%の下で頂点の削除数については有意差がないということが分かった．

**コード片が変更された頂点数について**

図 3 は，開発者および GenProg が 1 つの関数の CFG に対して，CFG の形状変化を伴わないコード片の変更をいくつの頂点に対して行ったかを示す箱ひげ図である．データ数は開発者が 116，GenProg が 60 である．コード片が変更された頂点数の中央値は，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，有意水準 5%の下でコード片が変更された頂点数については有意差があるということが分かった．

**コード片の変更行数について**

図 4 は，開発者および GenProg が 1 つの関数の CFG に対して，CFG の形状変化を伴わないコード片の追加および削除を何行ずつ行ったかを示す箱ひげ図である．データ数は，コード片が変更された頂点数についてのデータ数と同じである．

- 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，有意水準 5%の下でコード片の追加行数については有意差があるということが分かった．
- 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，有意水準 5%の下でコード片の削除行数については有意差があるということが分かった．

**if 文の追加・削除数について**

図 5 は，開発者および GenProg が 1 つの関数の 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す箱ひげ図である．データ数は開発者が 90，GenProg が 33 である．

- 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，有意水準 5%の下で if 文の追加数については有意差があるということが分かった．
- 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，有意水準 5%の下で if 文の削除数については有意差があるということが分かった．

**while 文の追加・削除数について**

図 6 は，開発者および GenProg が 1 つの関数の CFG に対して，whlie 文の追加および削除をそれぞれいくつ行ったかを示す箱ひげ図である．データ数は開発者が 23，GenProg が 9 である．

- 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，有意水準 5%の下で while 文の追加数については有意差があるというこ

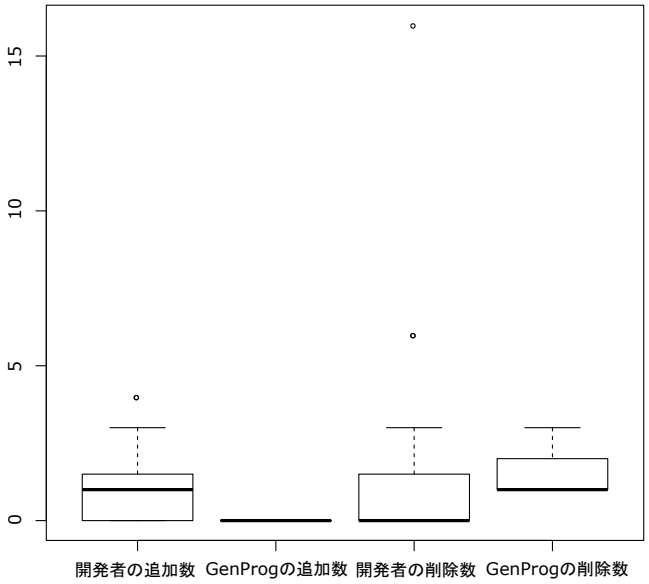


図 6　while 文の追加・削除数

とが分かった．

- 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，有意水準 5%の下で while 文の削除数については有意差がないということが分かった．

**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 個の関数の内，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 個の関数の内，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．

## 5. 議　　論

本章では，調査結果の考察および妥当性への脅威について述べる．

### 5.1 調査結果の考察

調査結果から，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．

頂点の削除数，while 文の削除数の 2 項目を除いて，開発者と GenProg で有意差が見られることが分かる．特に追加数に関しては，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，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．

以上の結果から，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．

a） 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が抱える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．

b） 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における問題を改善できる可能性があると考えられる．

### 5.2 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．

## 6. 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．

### 6.1 再利用に基づく手法

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 [8]．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 [11]．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．

### 6.2 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 [12]．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 つのオープン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 [13]．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．

### 6.3 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 [9] を提案した．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 [14]．

## 7. 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，特にプログラム文の追加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　本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：25220003），および文部科学省研究費補助金若手研究（A）（課題番号：24680002）の助成を得て行われた．

## 文　献

[1] Jennie 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] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[3] James A. Jones and Mary Jean 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.

[4] 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*, pp. 45–55, 2009.

[5] 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*, pp. 364–374, 2009.

[6] 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 *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

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

[8] 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.

[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 802–811, 2013.

[10] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. In *Transactions on Software Engineering*, pp. 1236–1256, 2015.

[11] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 772–781, 2013.

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

[14] Martin 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.