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.

再利用に基づく自動バグ修正における再利用候補の絞込に向けた調査

横山　晴樹† 　大田　崇史† 　堀田　圭佑† 　肥後　芳樹†

岡野　浩三†† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
†† 信州大学工学部情報工学科，長野市
E-mail: †{y-haruki,t-ohta,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp, ††okano@cs.shinshu-u.ac.jp

あらまし　近年，自動的にバグ修正を行う手法として既存プログラム文の再利用に基づく手法が注目を集めている．自動バグ修正は，ソースコード行の変更とテストの実行の 2 段階で構成され，全てのテストを通過するまでソースコード行の変更からやり直す手法である．ソースコード行の変更は，バグ同定された行に対しプログラム文の挿入，削除，置換のうち 1 つをランダムに実行する操作である．再利用に基づく手法におけるプログラム文の挿入では，挿入する文を同一ソフトウェア内より選択する．しかし，ソフトウェアの大規模化に伴い，選択可能なプログラム文の数が増大すると，バグ修正の反復回数も増大し，バグ修正に時間がかかる恐れがある．より短い時間でバグ修正を行うため，選択可能なプログラム文を絞り込むことが必要である．本研究では，絞り込みに向けた基準を提案し，4 つのオープンソースソフトウェアの過去のバグ修正履歴を対象に調査を行い，提案した基準の妥当性を確認した．

キーワード　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

Investigation for Reducing Reuse Candidates on Reuse-based Automated Program Repair

Haruki YOKOYAMA†, Takafumi OHTA†, Keisuke HOTTA†, Yoshiki HIGO†,

Kozo OKANO††, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871, Japan
†† Department of Computer Science and Engineering Shinshu University, Nagano-shi, 380–8553, Japan
E-mail: †{y-haruki,t-ohta,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp, ††okano@cs.shinshu-u.ac.jp

1. ま え が き

ソフトウェアの信頼性向上のために，デバッグは不可欠な作業である．デバッグにおける主要な作業は，バグの所在の特定とバグの修正である．デバッグはソフトウェア開発において多大なコストを伴う作業であり，ソフトウェア開発者はデバッグ作業にプログラミング時間の半分を費やすとも言われている [1]．そこで，デバッグの支援が必要である．

デバッグを支援する方法の 1 つとして，自動化が挙げられる．これまでの研究では，バグの所在特定の自動化（自動バグ同定）に関する研究が活発に行われてきた [2], [3]．しかし，デバッグを全て自動化する場合，自動バグ同定だけではなく，自動バグ修正も必要である．

自動バグ修正とは，バグ同定されたソースコード行に対し，プログラム文の挿入や削除，置換といった変更を自動で行うことによりバグ修正を行う手法である．これらの変更は，それぞれ次のような処理である．

挿入　バグ同定された行の次の行に挿入を行う処理
削除　バグ同定された行を削除する処理
置換　削除と挿入を連続して行う処理

ソースコード行の変更によるバグ修正の成否はテストの実行により検証される．全てのテストに成功すればバグ修正は完了するが，1 つでもテストに失敗すればソースコードの変更からやり直す．

近年成果を上げている自動バグ修正ツールとして，遺伝的プログラミングを利用した GenProg [4] や，ランダム探索に基づ

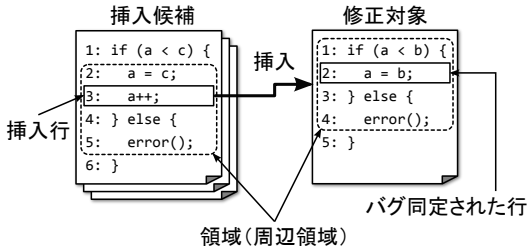

図 1　バグ修正において類似する周辺領域の例
Fig. 1　An Example of Similar Peripheral Regions on a Bug Fix

く RSRepair [5] などが挙げられる．GenProg や RSRepair は，修正の粒度をソースコードの行単位としており，バグ同定された行への挿入を行う際，プログラム文でなく行を挿入する．行の挿入は，同一ソフトウェアからランダムに 1 行を選択し，挿入する処理である．しかし，ソフトウェアの規模が拡大に伴い選択可能な行の数が増大し，反復回数が増えるため，時間がかかる恐れがある．したがって，挿入する行を選択する際に工夫が必要である．

行を選択する際に行う工夫として，同一ソフトウェアの各行に優先度付けを行い，優先度順に挿入する行を選択することが考えられる．優先度付けには，優先度を算出する基準が必要である．そのため，本研究では優先度の基準を提案し，4 つのオープンソースソフトウェアにおける過去のバグ修正履歴を対象に調査を行い，提案した基準の妥当性を確認した．本研究における貢献は，自動バグ修正において挿入する行をランダムに選択する手法に対する，代替案となる優先度付けの基準の提案と，提案した基準の妥当性の部分的検証である．

2. 調査の目的

本節では用語の定義を行った後，既存手法の課題を提示し，課題の解決に向けた調査動機と調査項目について述べる．

2.1 用語の定義

本小節では，本研究で用いる用語を定義する．

挿入候補　プログラムの変更処理において行の挿入を行う際に選択されうる行の集合

挿入行　プログラムの変更処理において行の挿入を行う際に挿入される行

領域　ソースコード中の連続する数行

行の周辺領域　特定の行を中心とする領域．i 行目の行の周辺領域は，領域の行数が n のとき，開始行が式 (1)，終了行が式 (2) となる領域である．

$$i - \lfloor n/2 \rfloor + 1 \tag{1}$$

$$i + \lceil n/2 \rceil \tag{2}$$

図 1 を例として，定義した用語を用いた説明を行う．図 1 は，共通する処理の片方でバグが発生しているため，抜けている行の挿入を行っている例である．この例では，挿入候補のうちあるソースファイルの 3 行目が挿入行として選択され，修正対象のバグ同定された行の次の行である 3 行目に挿入される．このとき，挿入行の周辺領域とバグ同定された行の周辺領域が類似していることが分かる．

2.2 再利用に基づく手法の課題

再利用に基づく手法とは，挿入候補を同一ソフトウェア内の各行に限定している自動バグ修正手法である．

GenProg [4] と RSRepair [5] に共通する処理の 1 つに，変異プログラムの生成がある．変異プログラムは，バグ同定された行に対して変更を加えることで生成できる．加える変更には，行の挿入，削除，置換があり，それぞれ次のような処理である．

挿入　バグ同定された行の次の行に挿入を行う処理

削除　バグ同定された行を削除する処理

置換　削除と挿入を連続して行う処理

一般に，行の挿入はプログラミング言語の構文が許容する限り無限に挿入候補が存在するという点で困難な処理である．再利用に基づく手法では，挿入候補を同一ソフトウェア内の各文とすることで，候補数を限定している．

GenProg と RSRepair は再利用に基づく手法を実装しているツールである．GenProg と RSRepair は，行の挿入の際に挿入行を挿入候補からランダムに 1 つ選択している．しかし，対象とするソフトウェアの規模の増大に伴い挿入候補の数も増大し，バグを修正できる行を発見できる確率が下がるため，挿入行をランダムに選択する手法は非効率である．

自動バグ修正の時間を短縮するためには，バグを修正できる行を短時間で見つける必要がある．そのため，ある行がバグ修正を実現できる見込みを優先度として，優先度順に挿入を行うことが考えられる．優先度を用いるためには，あらかじめ各行に対して優先度を算出しておく必要がある．また，優先度を算出するためには基準が必要である．

2.3 調 査 動 機

挿入候補の優先度付けを行うための基準を提案するため，図 1 のようなバグ修正例に注目する．これは，共通する処理の片方でバグが発生しているため，抜けている行の挿入を行っている例である．この例では，挿入行の周辺領域とバグ同定された行の周辺領域が類似していることが分かる．

この例から，自身の周辺領域とバグ同定された行の周辺領域が類似している行は，バグを修正できる行である可能性が高いと考えられる．そのような行において実際にバグを修正できる傾向があれば，挿入候補の優先度付けを行うための基準として，領域間の類似度を用いることを提案することができる．つまり，挿入候補の各行の周辺領域とバグ同定された行の周辺領域との類似度の順に優先度付けを行うことが自動バグ修正の時間短縮になる可能性がある．

2.4 調 査 項 目

本研究では，自身の周辺領域とバグ同定された行の周辺領域が類似するような行にバグを修正できる傾向があるかどうかを明らかにすることを目指す．そのため，次の研究課題を立てた．

RQ:　挿入行を含む領域とバグ同定された行の周辺領域の類似度は挿入行を含まない領域とバグ同定された行の周辺領域の類似度よりも高くなる傾向はあるか．

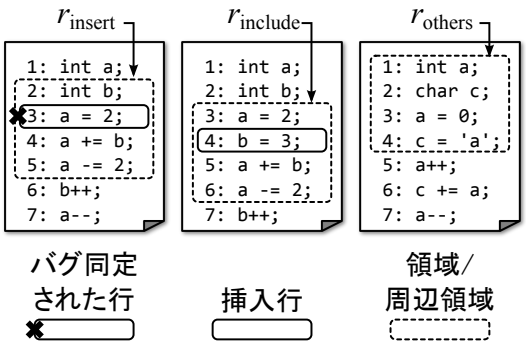

図 2　類似度算出に用いる領域
Fig. 2　Regions for Calculating Similarities

この RQ に対し Yes と回答できるならば，類似度が高い順に優先度付けをすることが有用であるといえる．No と回答できる場合は，類似度が低い順に優先度付けをすれば，同様に有用な手法であるといえる．類似度の差が見られない場合，類似度を用いた優先度付けは有用な手法ではないという結論を得る．

3. 調査の方法

調査では，解析対象のリポジトリ，バグ修正キーワード，領域の行数を入力とし，類似度を算出する．本節では，類似度の算出対象，指標，算出手順を述べる．

3.1 類似度の算出対象

本調査において算出する類似度は，挿入行を含む領域とバグ同定された行の周辺領域の類似度（S_{include}[(注1)]）および，挿入行を含まない領域とバグ同定された行の周辺領域の類似度（S_{others}）である．これらの類似度の算出に必要な情報は，次の 3 つの領域である．

- バグ同定された行の周辺領域（r_{insert}[(注2)]）
- 挿入行を含む領域（r_{include}）
- 挿入行を含まない領域（r_{others}）

これらの領域は図 2 のように表される．r_{insert} は，バグ同定された行が i 行目であり，領域の行数が n のとき，開始行が式 (1)，終了行が式 (2) となる領域である．r_{include} と r_{others} は r_{insert} と同じ行数の領域であるが，r_{include} は挿入行を含み，r_{others} は挿入行を含まない領域である．

また，領域の群に関して，挿入行を含む領域群を R_{include}，挿入行を含まない領域群を R_{others} と定義する．全ての領域群を R_{all} としたとき，R_{include} と R_{others} は式 (3) および式 (4) を満たす．

$$R_{\text{include}} \cup R_{\text{others}} = R_{\text{all}} \tag{3}$$

$$R_{\text{include}} \cap R_{\text{others}} = \emptyset \tag{4}$$

以上の領域の定義と 3.2 で述べる領域の類似度 (7) を用いて類似度 S_{include} を式 (5)，S_{others} を式 (6) のように定義する．

$$S_{\text{include}} = \{r_{\text{include}} \in R_{\text{include}} \mid s(r_{\text{include}}, r_{\text{insert}})\} \tag{5}$$

(注1)：S は Similarity を表す．
(注2)：r は Region を表す．

$$S_{\text{others}} = \{r_{\text{others}} \in R_{\text{others}} \mid s(r_{\text{others}}, r_{\text{insert}})\} \tag{6}$$

3.2 類似度の指標

類似度の指標にはレーベンシュタイン距離 [6] を基にした値を用いる．レーベンシュタイン距離とは，2 つの文字列に対する片方からもう片方への編集にかかる最小手数である．編集において，文字列への操作には文字の挿入，削除，置換があり，各操作は 1 回につき 1 手で行うものとして，手数を算出する．本調査では，文字列ではなくトークン列のレーベンシュタイン距離を考え，文字の代わりにトークンを用いる．ただし，レーベンシュタイン距離は対象とする文字列（トークン列）が長くなるにつれて最大値が大きくなる．様々な文字列（トークン列）における類似度を公平に比較するため，値の正規化が必要である．また，レーベンシュタイン距離が小さいほど類似度は高くなるべきなので，レーベンシュタイン距離を正規化した値の大小を逆転させる必要がある．

本調査では，2 つのトークン列 t_1，t_2 に対するレーベンシュタイン距離に基づく類似度 $s(t_1, t_2)$[(注3)] を式 (7) ように定める．

$$s(t_1, t_2) = 1 - \frac{\text{dist}(t_1, t_2)}{\max\{\text{len}(t_1), \text{len}(t_2)\}} \tag{7}$$

ここで，$\text{dist}(t_1, t_2)$ はレーベンシュタイン距離[(注4)]，$\text{len}(t_1)$[(注5)]，$\text{len}(t_2)$ はトークン列の長さを表す．$\text{dist}(t_1, t_2)$ を $\max\{\text{len}(t_1), \text{len}(t_2)\}$ で割ることで，値を正規化することができる．正規化により，$s(t_1, t_2)$ の範囲は $0 \leqq s(t_1, t_2) \leqq 1$ となる．また，$\text{dist}(t_1, t_2)/\max\{\text{len}(t_1), \text{len}(t_2)\}$ の値を 1 から引くことで，類似度が高くなるにつれて $s(t_1, t_2)$ が 1 に近づく．領域 r_1，r_2 に対する類似度 $s(r_1, r_2)$ は，r_1，r_2 をトークン化し，トークン列として扱うことで計算することができる．

3.3 調査手順

調査は図 3 のように以下の 6 ステップで行う．

ステップ 1　バグ修正コミットの特定
ステップ 2　バグ修正前後リビジョンの抽出
ステップ 3　挿入行の抽出
ステップ 4　ソースコードの正規化とトークン化
ステップ 5　類似度算出の準備
ステップ 6　類似度の算出

以下，上記の 6 ステップで行う処理を詳細に述べる．

ステップ 1 では，リポジトリのコミットログを解析し，バグ修正コミットを特定する．コミットがバグ修正を含んでいるか否かは，コミットログより判断する．本調査では，コミットログのコメントにバグ修正キーワードが含まれているコミットをバグ修正コミットと定める．バグ修正キーワードは “fix” や “fixed” などの，バグ修正を表す単語である．

ステップ 2 では，ステップ 1 で特定したバグ修正コミットを用いて，リポジトリからバグ修正を行う前後のリビジョンを得る．

(注3)：s は Similarity を表す．
(注4)：dist は Distance を表す．
(注5)：len は Length を表す．

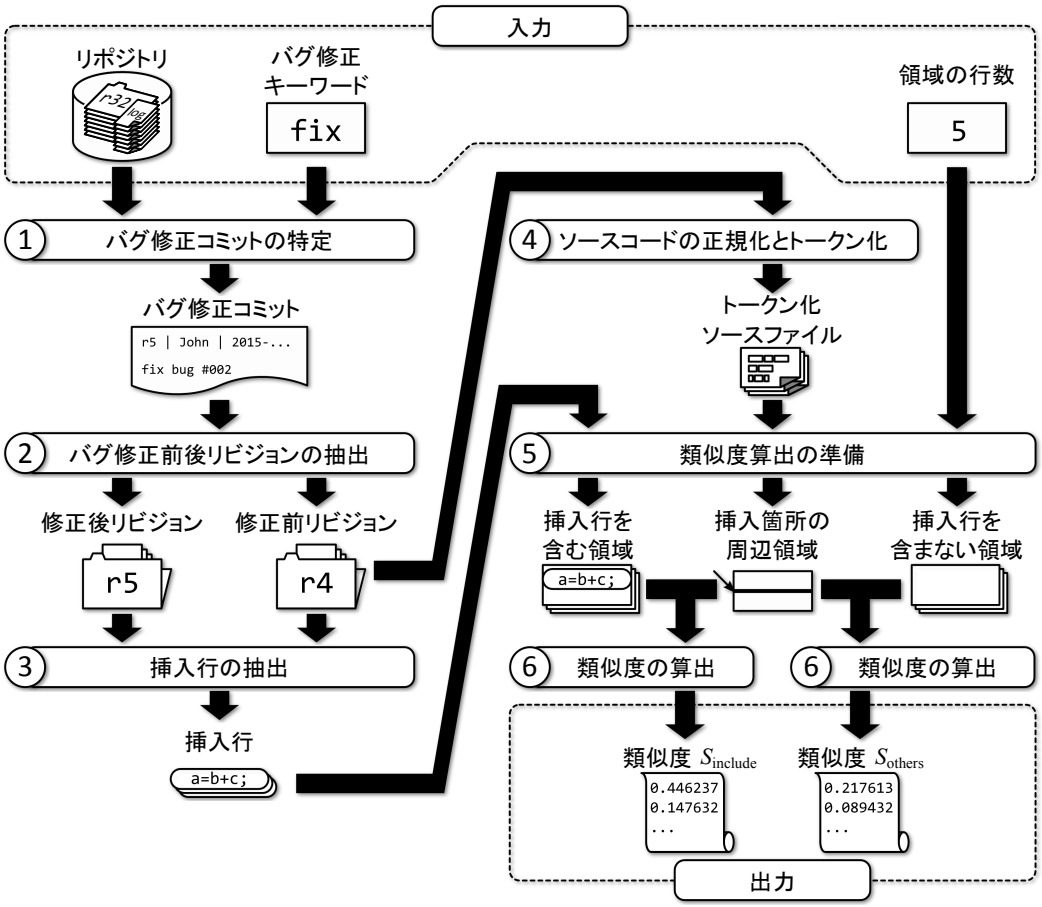

図 3　調査の流れ

Fig. 3　Research Methods Overview

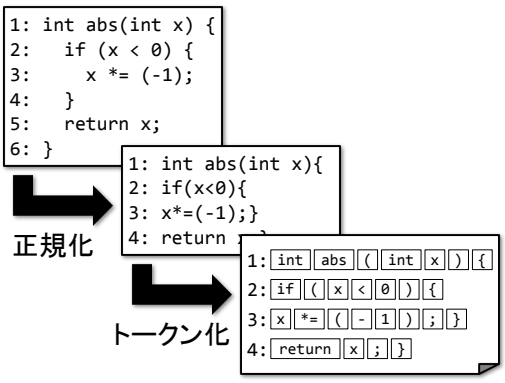

図 4　正規化とトークン化

Fig. 4　Normalize and Tokenize

ステップ 3 では，バグ修正を行う前後のリビジョンより，挿入行を抽出する．まず，バグ修正を行う前後のリビジョンより，各ソースファイルの差分を得る．差分は Unix の `diff` コマンドの考え方 [7] に基づいて抽出する．

ステップ 4 では，ステップ 2 で得たバグ修正前のリビジョンに含まれる各ソースファイルに対し字句解析を行う．字句解析では，図 4 のようにソースファイルを正規化（コメント，不要な空白，空行の除去）し，トークン化を行う．これらの処理を行うのは，類似度 S_{include}，S_{others} の値にプログラムの動作に関係のない要素（コメント，不要な空白，空行）や文字列の長短による影響を与えないためである．

ステップ 5 では，入力された領域の行数，ステップ 3 で得た挿入行の集合，ステップ 4 で得たトークン化ソースファイル群より，挿入行を含む領域群 R_{include}，挿入行を含まない領域群 R_{others}，バグ同定された行の周辺領域 r_{insert} を得る．まず，トークン化ソースファイル群の各行に対する周辺領域 R_{all} を求める．次に，各挿入行に対して，その挿入行を含むか否かで R_{all} を 2 群 R_{include}，R_{others} に分ける．また，挿入行に一対一対応する r_{insert} を求める．

ステップ 6 では，類似度の算出対象と指標を定め，準備した各領域より類似度 S_{include}，S_{others} の算出を行う．類似度の算出対象については 3.1，類似度の指標については 3.2 で述べた．

4. 調　　査

本節では，調査方法を実装したツールと調査対象，調査結果について述べる．

4.1 調査方法の実装

本研究では，3. で述べた調査方法を調査ツール **CalcSim** として実装し，調査を行った．CalcSim は Java 言語で記述されており，Subversion で管理された C，C++，Java 言語のプロジェクトに対して調査を行うことができる．

4.2 調 査 対 象

本研究では，4 つのオープンソースソフトウェアに対し調査を行う．各ソフトウェアの概要を表 1 に示す．調査において必要となる入力は，リポジトリ，バグ修正キーワード，領域の行数の 3 つである．本調査では，表 1 の 4 つのソフトウェアの各リポジトリに対し，バグ修正キーワードを “fix”，“fixed”，“fixing” の 3 つ，領域の行数を 5 として，CalcSim に入力する．

4.3 調 査 結 果

表 1 の 4 つのソフトウェアに対し調査を行った結果，各ソフトウェアに対してそれぞれ 3.1 で述べた類似度 S_{include} と S_{others} を得た．調査によって得た類似度を箱ひげ図として図 5 に示す．図 5 より，Apache httpd，CBMC，JabRef，において，S_{include} の中央値が S_{others} の中央値よりも大きいことが分かった．

一方，図 5 だけでは jEdit に関して明らかな情報を得られない．そこで，統計的手法により jEdit における S_{include} と S_{others} の比較を行った．まず，データの正規性を検証するため，1 標本コルモゴロフ・スミノフ検定を行った．検定の結果，S_{include} に対する p-値は 2.2×10^{-16} 未満，S_{others} に対する p-値も 2.2×10^{-16} 未満となり，2 群共に正規性がほぼ無いことが分かった．次に，正規性のない独立 2 群の有意差を検証するため，マン・ホイットニの U 検定を行った．検定の結果，p-値は 2.2×10^{-16} 未満となり，ほぼ確実に有意差があることが分かった．

以上の結果より，RQ に対して **Yes** と回答することができる．

5. 議　　論

本節では，調査結果の考察と調査の妥当性について議論する．

5.1 考　　察

調査結果より，自身の周辺領域とバグ同定された行の周辺領域が類似している行は，バグを修正できる行である可能性が高くなることを支持する知見を得た．この知見に基づき，挿入候補の各行の周辺領域とバグ同定された行の周辺領域との類似度の順に優先度付けを行うことが自動バグ修正の時間短縮になる可能性がある．

ただし，優先度付けを行う手法と既存のランダムに選択する手法では挿入候補は同じなので，修正可能なバグの種類は変わらない．また，優先度付けの処理にかかる時間が長くなる場合，優先度付けによって少ない反復回数で修正ができても，自動バグ修正全体の時間が伸びる恐れがある．

5.2 妥当性への脅威

本研究では，バグ修正コミットを特定する際，バグ修正キーワードを含むコミットを全てバグ修正コミットとして扱っている．この場合，バグ修正キーワード含むが実際にはバグを修正していないコミットまでバグ修正コミットとして扱っている可能性がある．例えば，ソースファイル内のコメントのタイプミスを “fix” した場合などもバグ修正コミットに含まれるという問題がある．

また，本調査では，領域の行数が 5 の場合しか調査をしていない．領域の行数を変化させることによって，類似度の算出時間や，類似度の値が変化する可能性がある．

6. 関 連 研 究

近年の自動バグ修正に関する研究には以下のものがある．

- 再利用に基づく確率的な手法 [4], [5]
- 再利用に基づく手法の適用範囲に関する調査 [8]
- プログラム意味論に基づく手法 [9]
- パターンに基づく手法 [10]

GenProg [4] は，修正対象プログラムとテスト集合の入力に対し，全てのテストを成功する修正プログラムを出力する．内部では，変異プログラムの複数生成，評価，選択を，全てのテストを成功するプログラムを生成するまで繰り返す．変異プログラムとは，修正対象プログラムの中でバグ同定された行に変更を加えたプログラムのことである．GenProg はこの手法により，実在するソフトウェアのバグを修正する成果を上げている．しかし，各変異プログラムに全てのテストを実行するため，計算コストが大きくなるという問題がある．

これに対して RSRepair [5] は，変異プログラムを 1 つだけ生成し，全てのテストを成功しなければ再度生成を行う手法を用いている．この手法では，1 つでもテストの失敗があれば，他のテストの適用を打ち切れるため計算コストを削減することができる．ただし，GenProg が複数のバグを含むプログラムの修正が可能であるのに対し，RSRepair は 1 つのバグを含むプログラムの修正しかできないという制限がある．

GenProg と RSRepair に共通する処理の 1 つに，プログラム変更時における行の挿入がある．GenProg と RSRepair が行の挿入を行う際，同一ソフトウェア内の行をランダムに 1 つ再利用する．再利用に基づく手法の妥当性に関する Barr らの研究 [8] は，実在するソフトウェア開発履歴のコミットにおいて，挿入される行の約 43%は同一ソフトウェアの直前のコミットより再利用可能であることを明らかにした．

SemFix [9] は，プログラム意味論に基づく手法であり，プログラムが満たすべき性質を表す論理式を基に修正を行う．SemFix はいくつかのプログラムに対して，GenProg よりも多くのバグを修正する成果を上げている．この手法では，論理式を SMT ソルバ [11] を用いて解く必要がある．SMT 問題は NP-完全であるため，論理式が複雑な場合，現在の CPU の処理能力では現実的な時間で解を求めることが難しくなる．

PAR [10] は，パターンに基づく手法であり，人が書いたパッ

表 1　調 査 対 象
Table 1　Research Targets

ソフトウェア	言語	ディレクトリ	開始リビジョン（日付）		終了リビジョン（日付）		行数
Apache httpd	C	/trunk	1,410,755	(2012/11/18)	1,421,851	(2012/12/14)	137,954
CBMC	C++	/trunk/src	3,602	(2014/02/22)	3,719	(2014/04/14)	116,506
jEdit	Java	/trunk/org	19,812	(2011/08/19)	20,292	(2011/11/11)	74,399
JabRef	Java	/trunk/jabref/src	3,349	(2010/11/01)	3,474	(2011/03/18)	62,780

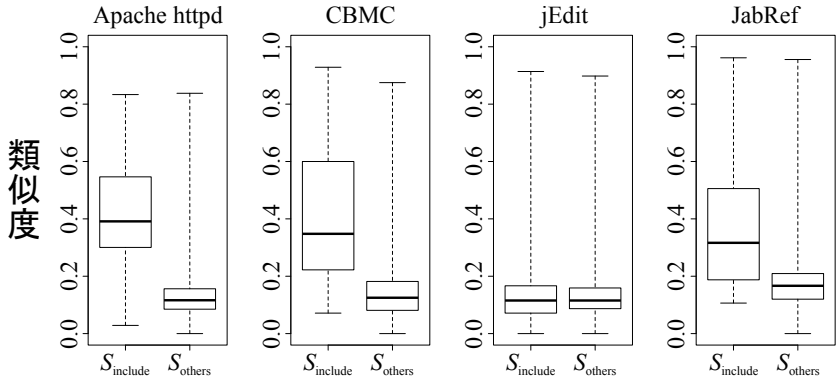

図 5　調査結果の箱ひげ図
Fig. 5　Boxplots of the Research Result

チを基に自動でパッチを生成する手法である．PAR はいくつかのプログラムに対して，GenProg よりも多くのバグを修正する成果を上げているが，その実験内容や実験対象に関する批判的議論も行われている [12]．

7. あ と が き

本研究における貢献は，自動バグ修正において挿入する行をランダムに選択する手法に対する，代替案となる優先度付けの基準の提案と，提案した基準の妥当性の部分的検証である．4 つのオープンソースソフトウェアにおける過去のバグ修正履歴を対象にとした調査と，箱ひげ図や統計的仮説検定による比較によって，提案した基準の妥当性を確認した．

今後の課題は，提案した基準に基づく優先度付け手法を挿入する行をランダムに選択している GenProg や RSRepair に実装し，手法の有用性を評価することである．

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) の助成を得て行われた．

文　　献

[1] J. Baker, “Experts battle £192bn loss to computer bugs,” Feb. 2012. Accessed 2015-04-11. http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html

[2] J.A. Jones and M.J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” ASE ’05, pp.273–282, ACM, New York, NY, USA, 2005. http://doi.acm.org/10.1145/1101908.1101949

[3] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang, “Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization,” ICSE ’09, pp.45–55, IEEE Computer Society, Washington, DC, USA, 2009. http://dx.doi.org/10.1109/ICSE.2009.5070507

[4] 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,” ICSE ’12, pp.3–13, IEEE Press, Piscataway, NJ, USA, 2012. http://dl.acm.org/citation.cfm?id=2337223.2337225

[5] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” ICSE ’14, pp.254–265, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2568225.2568254

[6] R.A. Wagner and M.J. Fischer, “The string-to-string correction problem,” J. ACM, vol.21, no.1, pp.168–173, Jan. 1974. http://doi.acm.org/10.1145/321796.321811

[7] E.W. Myers, “An O(ND) difference algorithm and its variations,” Algorithmica, vol.1, no.2, pp.251–266, 1986. http://dx.doi.org/10.1007/BF01840446

[8] E.T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, “The plastic surgery hypothesis,” FSE ’14, pp.306–317, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2635868.2635898

[9] H.D.T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” ICSE ’13, pp.772–781, IEEE Press, Piscataway, NJ, USA, 2013. http://dl.acm.org/citation.cfm?id=2486788.2486890

[10] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” ICSE ’13, pp.802–811, IEEE Press, Piscataway, NJ, USA, 2013. http://dl.acm.org/citation.cfm?id=2486788.2486893

[11] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” TACAS’08/ETAPS’08, pp.337–340, Springer-Verlag, Berlin, Heidelberg, 2008. http://dl.acm.org/citation.cfm?id=1792734.1792766

[12] 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,” ICSE ’14, pp.234–242, ACM, New York, NY, USA, 2014. http://doi.acm.org/10.1145/2568225.2568324