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.

自動バグ修正における対応可能バグ数の拡充に向けて

～大規模データセットを用いるアプローチのフィージビリティ調査～

鷲見　創一†　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: †{s-sumi,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　近年，既存ソースコード行の再利用による自動バグ修正手法が注目されている．再利用に基づく自動バグ修正では，バグであると特定された箇所へのソースコード行の挿入において，修正対象プロジェクトからソースコード行を 1 つ選択する．しかし，バグ修正によって追加されたソースコード行の内，修正対象プロジェクトに含まれるものの割合は十分ではない．そこで本研究では，再利用に基づく自動バグ修正手法によって，より多くのバグを修正可能とするための 2 つのアプローチを考える．1 つ目はソースコード再利用候補の探索範囲の拡大である．2 つ目は変数名の正規化である．本研究では，上記した 2 つのアプローチにより，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するか調査する．調査の結果，ソースコード再利用候補の探索範囲の拡大では 37～54%から 43～59%まで増加し，変数名の正規化では 37～54%から 56～64%まで増加することが明らかとなった．ソースコードの再利用候補の範囲拡大に加えて変数名の正規化も行った場合，探索範囲に現れるソースコード行の割合は 37～54%から 64～69%まで増加する．また，正規化した変数のうち周囲から復元できるものの割合は，周囲 5 行の場合で 24～49%，同一ファイル内では 36～78%であった．

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

Toward Increasing the Number of Graftable Bugs in Automated Program Repair

A Feasibility Study of Approach Using Huge Dataset

Soichi SUMI†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871, Japan
E-mail: †{s-sumi,higo,kusumoto}@ist.osaka-u.ac.jp

1. ま え が き

現在，ソフトウェアは開発者が多くの労力をかけることにより作られている．開発者がかける労力の中でも，デバッグは開発工程の半数以上を占めるといわれている [1]．そのため，デバッグの支援を目的とした多くの研究が行われている．

デバッグを行う際には，バグの所在の特定を行い，適切な修正方法を決定し，修正を適用する．デバッグを支援するため，これまでにバグである箇所の特定手法 [2] やバグの理解を支援する手法などが提案されている [3]．しかしこれらの手法を用いたとしても，適切な修正方法の決定やソースコードの変更は必要である．そこで，近年バグの特定と修正の適用を自動的に行う自動バグ修正手法が数多く提案されている．

自動バグ修正手法の 1 つとして，ソースコードの再利用に基づく手法 (以下，再利用に基づく手法と呼ぶ) である GenProg がある [4]．GenProg では，バグを含むプログラムと，失敗テストを含むテストスイートを入力として，全テストを通過するプログラムを出力する．テストスイートを用いてバグの原因であるソースコード行を特定し，プログラム文の挿入や削除，置換などの変更を遺伝的プログラミングに基づいて行うことによりプログラムを修正する．挿入や置換に用いるソースコード行には修正対象プログラムに存在するソースコード行を用いる．また，変更が加えられたプログラムが全てのテストを通過するかどうかによってプログラムの修正が完了したかどうかを判定

する．もし失敗するテストが存在するなら，再度プログラムの変更を行う．

GenProg は 8 つのオープンソースソフトウェア (以下，OSS と呼ぶ) に対して適用され，105 個中 55 個のバグの修正に成功することによってその有効性を示した．GenProg は，修正対象プログラム中に存在するソースコード行によってバグを修正することが可能であるという前提に基づいてプログラムの修正を行っている．この前提が誤りならば，GenProg はバグ修正を完了することができない．Barr らはこの前提について 12 個の OSS を用いて調査を行った [5]．調査の結果，42%の変更において，変更によって追加されたソースコード行の半数以上が同一プロジェクトのソースコードに現れることが明らかとなった．しかしこの結果は，再利用に基づく手法が約半数のバグを修正できないことも示している．そのため，より多くのバグを再利用に基づく手法によって修正可能とするためには工夫が必要である．

そこで，2 つのアプローチを考える．1 つ目はソースコード再利用候補の探索範囲の拡大である．既存研究ではソースコードの変更によって追加されたソースコード行が変更以前のプログラムにどの程度現れるか調査している．本調査では，プログラムの変更によって追加されたソースコード行が変更以前のプログラムだけでなく，大規模データセットに現れるかどうか調べる．これにより変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するかを調査する．

2 つ目は変数名の正規化である．異なったソフトウェアで似た構文があったとしても，変数名まで同じであるとは考えにくい．変数名を正規化してソースコードを再利用し，バグであると特定された箇所の周囲から変数名を復元することで，再利用可能なソースコード行の割合は増加すると考えられる．そこで，変数名の正規化により，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するかについて調査する．また，変更によって追加されたソースコード行に現れる変数名が，どの程度追加されたソースコード行の周囲に現れるかについても調査する．

以上より本研究では，上述した 2 つのアプローチにより，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するか調査する．

2. 関 連 研 究

近年自動バグ修正に関する研究は数多く行われており，以下のようなものがある．

- 再利用に基づく手法 [4], [6]
- プログラム意味論に基づく手法 [7], [9]
- 修正パターンに基づく手法 [10]

これらのバグ修正手法は修正対象プログラムと失敗テストを含むテストスイートを入力として，修正が完了したプログラムを出力する．バグである箇所の特定と修正完了の判断にはテストスイートを用いている．以降では，それぞれの手法がどのように修正を行うか説明する．

再利用に基づく手法の 1 つである GenProg では，修正対象プログラムのソースコード行を用いてバグであると特定された箇所を変更したプログラム (以降，変異プログラムと呼ぶ) を複数個生成し，評価，選択を行う．これを繰り返し適用することによりバグ修正を行う [4]．図 1 に GenProg の動作の流れを示す．プログラムの変更で行うのは以下の処理のうちの 1 つである．

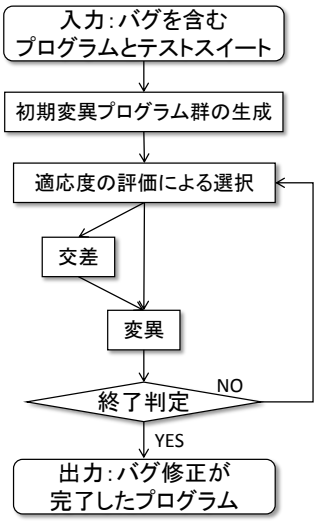

図 1　GenProg の動作の流れ
Fig. 1　Process of automated repair by GenProg

挿入　バグであると特定された行の次の行にソースコード行の挿入を行う処理

削除　バグであると特定された行を削除する処理

置換　削除と挿入を同時に行う処理

Le Goues らはこの手法を 8 つの OSS に対して適用し，105 個のバグのうち 55 個の修正に成功することによりその有用性を示した．しかし GenProg は，変異したプログラムの評価時に与えられた全てのテストを実行するため計算コストが高い．そこで，Qi らは失敗したテストが現れた時点でテストの実行を打ち切って新たに変異プログラムを生成し，プログラムの修正を行う RSRepair を提案した．Qi らは RSRepair を GenProg と同じ 8 つの OSS に対して適用し，GenProg よりも多くのバグを短い時間で修正できたことを報告している．しかし，RSRepair は複数行の変更を必要とするバグを修正できない．

プログラム意味論に基づく手法である SemFix では，テストスイートからバグであると特定された箇所が満たすべき制約を導出し，制約を満たすプログラム文を生成する [7]．Nguyen らはこの手法を 5 つのソフトウェアに対して適用し，GenProg よりも多くのバグを修正できたことを報告している．再利用に基づく手法は既存ソースコードに存在しない記述による修正を行うことができないのに対し，プログラム意味論に基づく手法は過去に存在しない記述を用いた修正が可能であることが特徴である．この手法はテストスイートからバグである箇所が満たすべき論理式を導出し，その論理式を SMT ソルバを用いて解く [8]．SMT 問題は NP-完全の問題であるため，論理式によっては現実的な時間で解くことができない．また，複数行の変更を必要とするバグを修正できない．DirectFix は SemFix を改

表 1 調 査 対 象
Table 1 Research Targets

ソフトウェア	開始リビジョン（日付）		終了リビジョン（日付）		コミット数
Camel	babdb30	(2007/06/04)	390e6c2	(2015/03/26)	1,625
Felix	826b7fc	(2006/03/27)	50d4715	(2015/04/15)	1,472
Hadoop-common	85651f6	(2009/06/23)	0a9f2c5	(2014/08/21)	922
Hive	011a680	(2008/11/13)	7b49691	(2015/04/17)	2,281
OpenJPA	eb96e89	(2006/08/01)	bdaf141	(2015/04/16)	1,348

良した手法である [9]．Mechaev らはこの手法を 5 つのソフトウェアに対して適用し，人が理解しやすい修正を生成できたと報告している．しかし，修正には SemFix よりも長い時間を要する．

修正パターンに基づく手法である PAR では，あらかじめ 10 個の修正パターンを作成し，それらに基づいて修正を行う．Kim らは 6 つの OSS への適用によって GenProg よりも多くのバグを修正することに成功し，理解しやすい修正を生成できたことを報告している．しかし，用意された修正パターンに当てはまらないものは修正できないことや，修正パターンの作成が技術的に難しいものが存在することが課題である [10]．また，実験対象や実験内容に対する批判がある [11]．

本研究では，再利用に基づく手法を対象として調査を行う．この手法は複数行の変更を必要とするバグを修正可能であり，より多くのソースコードを再利用元とすることによって，より多くのバグを修正することが可能である．また，修正に長い時間を要するという課題があるが，修正が完了しやすいソースコード行を効率的に取得しプログラムの変異に用いることなどによって修正時間を短縮することが可能であると考えられる．

3. 調査の目的

本章では再利用に基づく自動バグ修正手法の課題を示し，課題解決に向けた調査動機と調査項目について述べる．

3.1 再利用に基づく手法の課題

再利用に基づく手法の 1 つである GenProg は 8 つの OSS に対して適用され，105 個中 55 個のバグの修正に成功することによってその有効性を示した [4]．しかし，GenProg は「修正対象に存在するソースコードを用いてバグを修正することが可能である」という前提に基づいている．そこでいくつかの調査が行われた．

Barr らは 12 個の OSS の開発履歴を用いて，変更によって追加されたソースコードが同一プロジェクト内のソースコードにどの程度現れるか，行単位の粒度で調査を行った [5]．調査の結果，42%の変更において，変更によって追加されたソースコード行の半数以上が同一プロジェクトのソースコードに現れることが明らかとなった．

Martinez らも同様の調査を字句単位の粒度で行った．調査の結果，実験対象ソフトウェアの 31%〜53%の変更において，変更によって追加されたソースコードの字句すべてが，同一ソフトウェアのソースコードの字句に現れることが明らかとなった [12]．

しかし，これらの結果はソースコードの再利用に基づく自動バグ修正手法が約半数以上のバグを修正できないことも示している．そのため，ソースコードの再利用に基づく自動バグ修正手法によってより多くのバグを修正するためには工夫が必要である．

3.2 調 査 動 機

再利用に基づく手法によってより多くのバグを修正するため，2 つのアプローチを考える．

1 つ目はソースコード再利用候補の探索範囲の拡大である．既存研究ではソースコードの変更によって追加されたソースコード行が変更以前のプログラムにどの程度現れるかを調査した．本調査では，プログラムの変更によって追加されたソースコード行の探索範囲を大規模データセットに拡大することで，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するかを調べる．

2 つ目は変数名の正規化である．他のソフトウェアで似た構文があったとしても，変数名まで同じであるとは考えにくい．そこで，変数名を正規化してソースコードを再利用し，バグであると特定された箇所の周囲から変数名を復元することで，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するか調べる．

3.3 調 査 項 目

本研究の目的は，上述した 2 つのアプローチにより，再利用に基づく手法によって修正可能なバグの数がどの程度変化するか明らかにすることである．そのため，以下の調査項目を設定する．

RQ1 ソースコードの再利用候補の探索範囲を大規模データセットへ拡大することにより，変更によって追加されたソースコード行のうち探索範囲に現れるソースコード行の割合はどの程度変化するか

RQ2 変数名を正規化することで，変更によって追加されたソースコード行のうち探索範囲に現れるソースコード行の割合はどの程度変化するか

RQ3 変更によって追加されたソースコード行に含まれる変数は，追加先の周辺にどの程度現れるか

4. 調 査

本章では調査の方法について述べる．

4.1 データの収集

本調査は 5 つの OSS に対して行う．調査対象プロジェクトを表 1 に示す．調査では，OSS の開発履歴のコミットと大規

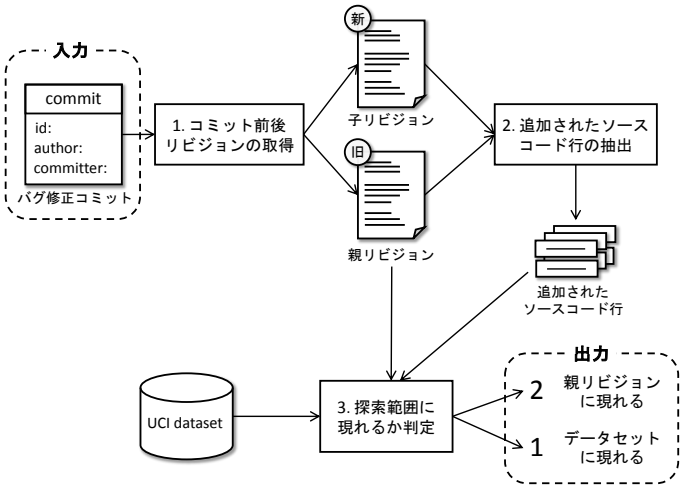

図 2　実験 1，実験 2 の流れ
Fig. 2　Process of experiment 1 and experiment 2

模データセットのソースコード行の集合を入力として，変更によって追加されたソースコード行や変数のうち探索範囲に現れるものを出力する．大規模データセットには，”UCI source code data sets”[13](以降，UCI データセットと呼ぶ) を用いる．UCI データセットの詳細を表 2 に示す．UCI データセットに含まれるソースコードは正規化して改行記号で行ごとに分割し，ソースコード行の集合として取得する．正規化については 4. 2 節で説明する．

開発履歴に含まれるコミットにはバグ修正には関係のないものが含まれるため，コミットコメントと課題管理ツールである JIRA を用いてバグ修正に関するコミットを特定する．実験対象である Hadoop-common プロジェクトのコミットには多くのファイルが移動されたコミット (コミット ID：9a9fcf8) が存在する．本研究ではバグ修正に関するソースコードの変更について調査することが目的であるため，調査対象から除外する．

4. 2　調 査 方 法

本調査では各 RQ に回答するために 3 つの実験を行った．各実験の概要は以下の通りである．

実験 1　コミットにより追加されたソースコード行が探索範囲にどの程度現れるか調査する．探索範囲は親リビジョンの全てのソースコードの場合と親リビジョンの全てのソースコードに UCI データセットを加えた場合で比較する．

実験 2　変数名の正規化を行い，コミットによって追加されたソースコード行が探索範囲にどの程度現れるか，実験 1 と同様に調査する．

実験 3　コミットにより追加されたソースコード行に含まれる

表 2　UCI データセットの詳細 (収集日：2010/04)
Table 2　Details of UCI dataset(archived : April 2010)

ソフトウェア数	13,193
ソースファイル数	2,072,490
総行数	361,663,992
全容量	30.6GByte

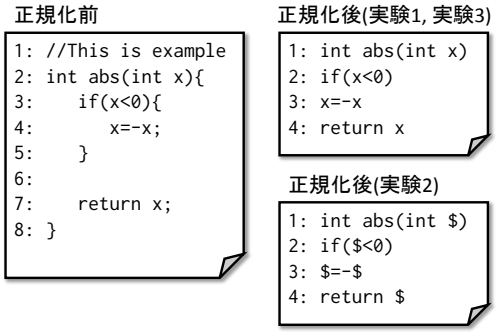

図 3　正規化の流れ
Fig. 3　An example of normalization

変数名が追加された位置の周囲に存在するか調査する．この実験では，挿入行の前後 5 行，10 行，20 行や同一ファイル内に変数名が存在するか調査する．

まず，実験 1 について説明する．実験 1 では，コミットにより追加されたソースコード行が探索範囲に現れるかどうかを調査する．自動バグ修正におけるソースコード行の削除にはソースコードの再利用を必要としないため，本調査ではコミットによって追加されたソースコード行のみを対象とする．実験 1 の概略を図 2 に示す．調査は以下の 3 ステップで行う．

STEP-A1　コミット前後のリビジョンの取得
STEP-A2　コミットにより追加されたソースコード行の抽出
STEP-A3　探索範囲に現れるかどうかの判定

以下では，各ステップで行う処理の詳細について述べる．

STEP-A1 では，コミット前後のリビジョンを取得する．親リビジョンについては，STEP-A3 で用いるため全てのソースファイルを取得する．

STEP-A2 では，コミットにより追加されたソースコード行を取得する．追加されたソースコード行の抽出は，コミット前後のリビジョンの全てのソースファイルを正規化し，それぞれ差分を抽出することにより行う．コミットにより新たにファイルが追加された場合には，追加されたファイルのソースコード行全てが挿入されたソースコード行として扱う．差分の抽出には Myers の diff アルゴリズム [14] を使用した．

STEP-A3 では，挿入されたソースコード行のうち，親リビジョンやデータセットに現れるものを取得する．探索範囲に現れるソースコード行の取得では，まず STEP-A1 で取得した親リビジョンの全てのソースコードを正規化しソースコード行の集合を取得する．次に追加されたソースコード行のうち，探索範囲のソースコード行の集合から見つかったものを探索範囲に現れるソースコード行とする．

ソースコードの正規化では，以下の 3 つを行った．図 3 にソースコードの正規化の例を示す．

- コメント，空行，インデントの除去
- 中括弧 ({ , }) の除去
- 変数名の正規化 (実験 2 のみ)

次に実験 2 について説明する．実験 2 では実験 1 と同様の実験を行う．ソースコードの正規化時に変数名の正規化も行うこ

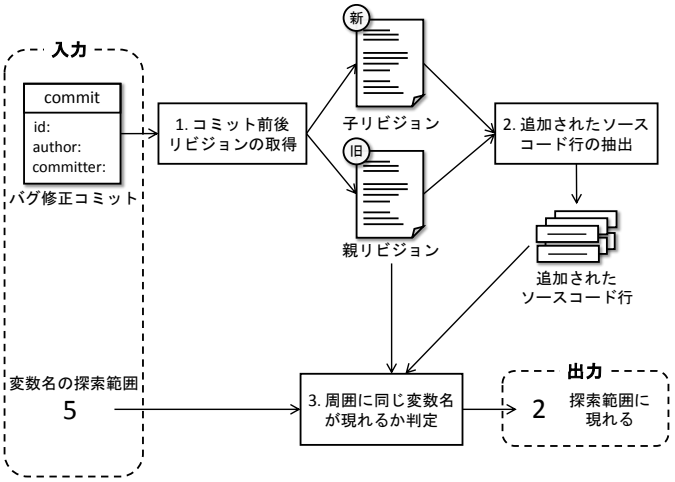

図 4　実験 3 の流れ
Fig. 4　Process of experiment 3

とが実験 2 と実験 1 との違いである．

最後に実験 3 について説明する．実験 3 の概略を図 4 に示す．実験 3 は以下の 3 ステップで行う．

STEP-B1　コミット前後のリビジョンの取得

STEP-B2　コミットにより追加されたソースコード行の抽出

STEP-B3　周囲に同じ変数名が現れるかどうかの判定

STEP-B1，B2 については STEP-A1，A2 と同様である．

STEP-B3 では，コミットにより追加されたソースコード行に含まれる全ての変数を取得し，追加されたソースコード行の周辺に現れるものを特定する．追加されたソースコード行の周辺とは，たとえば周囲 5 行の場合，追加されたソースコード行の前後 5 行の領域を指す．図 5 に追加されたソースコード行の周辺の例を示す．ソースコード行に含まれる変数の取得には，Eclipse JDT を用いる．

4.3 調 査 結 果

実験 1，実験 2 の実験結果を表 3 に示す．PARENT は，コミットによって追加されたソースコードの探索範囲が親リビジョンのみの場合の実験結果である．また，LARGESET は探索範囲を UCI データセットに拡大した場合の実験結果である．括弧内の数値は，実験 1 の PARENT の場合と比較してどれだけ値が増加したかを示している．

実験 1 の結果について説明する．コミットによって追加されたソースコード行のうち探索範囲に現れるものの割合は，探索範囲の拡大により 37〜54%から 43〜59%に増加した．各プロジェクトでは 5〜8%の増加で，増加分の中央値は 6%であった．

表 3　実験 1，実験 2 結果

Software	実験 1 正規化なし		実験 2 正規化あり	
	PARENT	LARGESET	PARENT	LARGESET
Camel	53.7%	58.8%	63.9%	69.3%
	–	(+5.1%)	(+10.2%)	(+15.5%)
Felix	43.4%	51.0%	59.9%	68.8%
	–	(+7.6%)	(+16.5%)	(+25.4%)
Hadoop-common	43.7%	50.0%	59.3%	66.2%
	–	(+6.3%)	(+15.6%)	(+22.5%)
Hive	37.3%	42.9%	56.3%	64.1%
	–	(+5.6%)	(+19.0%)	(+26.8%)
OpenJPA	42.7%	49.2%	57.6%	65.0%
	–	(+6.5%)	(+14.9%)	(+22.3%)

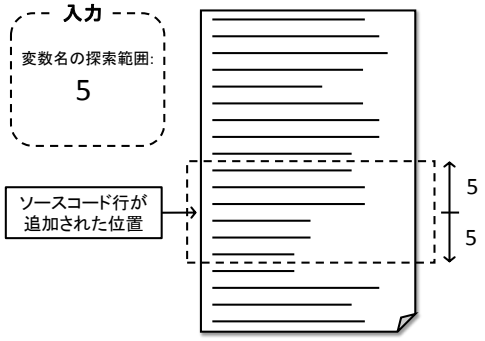

図 5　追加されたソースコード行の周囲
Fig. 5　Surrounging code line

実験 2 の実験結果について説明する．表 3 から，変数の正規化によって，コミットにより追加されたソースコード行のうち探索範囲に現れるものの割合は 37〜54%から 56〜64%へ増加したことが分かる．各プロジェクトでは 10〜19%の増加で増加分の中央値は 15%と，変数名の正規化により，探索範囲に現れるソースコード行の割合は大きく向上した．また，探索範囲を UCI データセットに拡大して変数名の正規化を行った場合，探索範囲に現れるソースコード行の割合は 37〜54%から 64〜69%まで増加する．各プロジェクトでは 15〜27%の増加で，中央値は 22%であった．

実験 3 の実験結果を表 4 に示す．コミットにより追加されたソースコード行に含まれる変数のうち，追加された行の周囲から引用できるものの割合は周囲 5 行で 24〜49%で, 中央値は 44%であった．追加されたソースコード行に含まれる変数のうち同一ファイル内に現れるものの割合は 36〜78%で，中央値は 68%であった．

以上より各 RQ への回答は次の通りである．

RQ1 への回答　変更によって追加されたソースコード行のうち探索範囲に現れるソースコード行の割合は 37〜54%から 43〜59%へ増加する．各プロジェクトでは 5〜8%の増加で，増加分の中央値は 6%である．

RQ2 への回答　変更によって追加されたソースコード行のうち探索範囲に現れるソースコード行の割合は 37〜54%から 56〜64%へ増加する．各プロジェクトでは 10〜19%の増加で増加分

表 4　実験 3 結果
Table 4　Result of experiment 3

プロジェクト	周囲 5 行	周囲 10 行	周囲 20 行	ファイル内
Camel	49.0%	55.5%	60.6%	67.8%
Felix	46.4%	54.1%	60.5%	72.6%
Hadoop	44.0%	51.1%	58.2%	70.8%
Hive	34.5%	41.0%	47.7%	77.8%
OpenJPA	23.6%	27.4%	30.2%	35.5%

の中央値は 15%である．また，変数名の正規化に加えてソースコードの再利用候補の範囲拡大も行った場合，探索範囲に現れるソースコード行の割合は 37〜54%から 64〜69%まで増加する．

RQ3 への回答　変更によって追加されたソースコード行に含まれる変数のうち，追加先の周辺に現れるものの割合は，周囲 5 行の場合は 24〜49%で，中央値は 44%である．また，変更が行われたファイル内から復元する場合は 36〜78%で，中央値は 68%である．

5. 妥当性の脅威

本研究は，課題管理システムである JIRA の情報を利用してバグに関係したコミットについて調査しており，バグ修正以外のコミットも含まれる可能性がある [15], [16]．そのため，バグ修正のために挿入されたソースコード行のみを調査した場合，今回とは異なった結果が得られる可能性がある．特に引用可能な変数の割合は向上すると考えられる．

また今回の調査では，5 つの Java で記述された OSS に対して調査を行った．多くの実験対象やほかの言語で記述されたソフトウェアに対して同様の調査を行った場合，異った結果が得られる可能性がある．

本研究では，2010 年 4 月時点の Apache を含む OSS の集合である UCI データセットを大規模データセットに用いている．実験対象は Apache のソフトウェアであり，開発履歴の一部は UCI データセットの収集日より過去の変更が含まれている．そのため，それらの変更で追加されたソースコード行が探索範囲に現れる割合は不当に高い可能性がある．しかし筆者らが実験結果を確認したところでは，そのような傾向は見当たらなかった．

6. あ と が き

本研究では，再利用に基づく手法によってより多くのバグを修正するため，2 つのアプローチを考えた．1 つ目はソースコード再利用候補の探索範囲の拡大，2 つ目は変数名の正規化である．そしてこれらのアプローチにより，変更によって追加されたソースコード行のうち探索範囲に現れるものの割合がどの程度変化するか調査を行った．調査の結果，2 つのアプローチにより，変更によって追加されたソースコード行のうち探索範囲に現れるソースコード行の割合は 37〜54%から 64〜69%まで大きく増加することが明らかとなった．また，正規化した変数のうち周囲から復元できるものの割合は，周囲 5 行の場合で 24〜49%，同一ファイル内で 36〜78%であった．

今後は，上記した 2 つのアプローチを GenProg に実装し，その有効性について検証を行う予定である．上記した 2 つのアプローチを実装する場合，再利用候補が増えることによる修正時間の増加が考えられる．そのため，修正が完了しやすいソースコード行の効率的な取得方法の考案が今後の課題である．

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) の助成を得て行われた．

文　献

[1] J. Baker, "Experts battle £192bn loss to computer bugs," Feb. 2012. Acessed 2015-06-09. http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html

[2] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault localization for data-centric programs," in Foundations of Software Engineering, 2011.

[3] D. Zuddas, W. Jin, F. Pastore. "MIMIC : Locating and Understanding Bugs by Analyzing Mimicked Executions," ASE'14, Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp.815-826, New York, NY, USA, 2014, ACM ,http://www.lta.disco.unimib.it/lta/uploads/papers/Xie-UnitTestOperationalViolation-ASE-2006.pdf

[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, pp3-13, IEEE Press, Piscataway, NJ, USA, 2012. http://dl.acm.org/citation.cfm?id=2337223.2337225

[5] 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

[6] 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

[7] H.D.T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," ICSE'13, pp.802-811, IEEE Press, Piscataway, NJ, USA, 2013. http://dl.acm.org/citation.cfm?id=2486788.2486890

[8] L. De. Moura and N. Børner, "Z3: An efficient smt solver," TACAS'08/ETAPS'08, pp337-340, Springer-Verlag, Berlin, Heidelberg, 200. http://dl.acm.org/citation.cfm?id=1792734.1792766

[9] S. Mechtaev. J. Yi and A. Roychoudhury. "DirectFix: Looking for Simple Program Repairs," In Proceedings of the 2015 International Conference on Software Engineering, pp.448-458, 2015.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," ICSE'13 pp.772-781, IEEE Press, Piscataway, NJ, USA, 2013. http://dl.acm.org/citation.cfm?id=2486788.2486893

[11] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and evaluation of automatic software repair," ICSE'14, pp.234-242, ACM.

[12] M. Martinez, W. Weimer, and M. Monperrus. "Do the fix ingredients already exist? An empirical inquiry into the redundancy assumpions of program repair approaches," In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pp492-495, New York, NY, USA, 2014. ACM.

[13] C. Lopes, S. Bajracharya, J. Ossher, and P.Baldi. Uci source code data sets. http://www.ics.uci.edu/ lopes/datasets/

[14] E.W. Myers, "An O(ND) difference algorithm and its variations," Algorithmica, vol.1, no.2, pp.251-266, 1986. http://dblp.uni-trier.de/rec/bib/journals/algorithmica/Meyers86

[15] K. Herzig and A. Zeller. The impact of tangled code changes. In MSR'13, pp.121-130.

[16] H. Kirinuki, Y.Higo, K. Hotta, and S. Kusumotno. "Hey! are you commiting tangled changes?," In Proceedings of the 22nd International Conference on Program Comprehension, pp.262-265, 2014.