特別研究報告

題目

再利用に基づく自動プログラム修正における再利用候補の絞り込み手法の実装と評価

指導教員

楠本 真二 教授

報告者

谷門 照斗

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

再利用に基づく自動プログラム修正における再利用候補の絞り込み手法の実装と評価

谷門 照斗

内容梗概

ソフトウェア開発において，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，自動プログラム修正に関する研究が活発に行われている．自動プログラム修正手法として，既存プログラム文の再利用に基づいて修正を行うものがある．自動プログラム修正は，プログラムの変更とテスト実行の 2 段階で構成され，すべてのテストを通過するまでプログラムの変更とテスト実行を繰り返す手法である．プログラムに加える変更としては，プログラム文の挿入や削除，置換がある．再利用に基づく手法におけるプログラム文の挿入では，ソースコード中のすべてのプログラム文を再利用候補とし，再利用候補の中から挿入する文をランダムに選択する．しかし，修正対象のプログラムの規模が大きくなるにつれて，再利用候補となるプログラム文の数が膨大になる．そのため，プログラムの変更とテスト実行の反復回数が増大し，プログラムの修正が完了するまでに非常に長い時間を要する．より短い時間でプログラムを修正するために，プログラムの変更において再利用するプログラム文を絞り込むことが必要である．絞り込みの基準として，更新順と類似度順が先行研究で提案されている．しかし，先行研究では絞り込み基準の提案に留まっており，ツールの実装と評価は行われていない．本研究では，これらの手法を自動プログラム修正ツール jGenProg2 に対しそれぞれ実装し，これらの手法が実際の欠陥修正において有用なのかを評価するため，106 個の欠陥に対してツールを実行した．実験の結果，修正欠陥数と修正時間の面において，類似度順および更新順は jGenProg2 よりも優れているとは言えないことを明らかにした．

主な用語

デバッグ
自動プログラム修正
コード再利用

目次

図目次

表目次

1　まえがき

ソフトウェアの安全性や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．デバッグにおける主要な作業は，ソフトウェアに含まれる故障を検出し，ソフトウェアのソースコードの中から故障を引き起こす原因となった記述（以降，欠陥と呼ぶ）を含む箇所を特定し，特定された欠陥を修正する作業である．

デバッグはソフトウェア開発において多大なコストを要する作業であり，開発者のプログラミング時間の約 50% を占めるといわれている [1]．このように多大なコストを要するデバッグ作業の負荷削減を目指して，デバッグの支援に関する研究が数多く行われている [2, 3, 4, 5, 6]．

デバッグ支援の理想形は，デバッグ作業の完全自動化である．ソフトウェアの故障の検出および欠陥箇所の限局の自動化に関しては，これまでに多数の研究が行われてきた [7, 8]．さらに近年では，欠陥修正の自動化を含めたプログラムの自動修正手法に関する研究も盛んに行われるようになってきており，デバッグ作業の完全自動化に向けて活発に研究が行われている．

自動プログラム修正手法の 1 つに GenProg [9] がある．GenProg は実際のソフトウェア開発において生じた欠陥 105 個のうち 55 個を修正したため，自動プログラム修正手法の中でも有望視されている [10]．GenProg は欠陥を含むプログラム（以降，修正対象プログラムと呼ぶ）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 [11] に基づき，修正対象プログラム中のコード片を用いて欠陥箇所に繰り返し変更を加えることによって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すべてのテストケースを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すべ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．

GenProg は修正対象プログラムに対して繰り返し変更を行うことで欠陥修正を試みる．プログラムに加える変更には，挿入，削除，および置換の 3 種類の操作がある．挿入操作は，修正対象プログラム中に存在するプログラム文をランダムに選択し，それを特定の箇所に挿入するという操作である．ランダムな選択を行うため，プログラムの規模が大きくなるに伴い修正に寄与しないプログラム文を選択する回数が多くなり，欠陥修正に要する時間が増大してしまうという課題がある．

そこで，欠陥の修正に寄与する可能性が高いプログラム文を選択する手法が提案されている．その手法として，類似度順 [12] と更新順 [13] が先行研究において提案されている．

しかし，これらの研究では手法の提案に留まり，ツールとしての実装はまだ行われていない．本研究では，GenProg を Java 向けに再実装したツール [14, 15] に対して，類似度順および更新順を実装した．また，実装したツールを実際のソフトウェア開発の過程で生じた欠陥に対して動作させ，これらの

手法が実際の欠陥修正において有効に働くのかを評価した．

以降，2 章では本研究の前提となる技術および用語の説明を行う．3 章では本研究の目的について述べる．4 章では実装したツールについて説明し，5 章では実装したツールと既存ツールに対する評価実験について述べる．6 章では実験結果に対する考察を行い，7 章では妥当性の脅威について述べる．8 章では本研究に関連する既存研究について説明し，最後に 9 章で本研究のまとめと今後の課題について述べる．

2 準備

本章では本研究の前提となる技術および用語について説明する．

2.1 テストケースを用いたデバッグ

一般的にデバッグは，ソフトウェア中に生じた欠陥の発見から始まり，欠陥の原因となっている箇所を特定して修正し，最後に修正の成否を確認するまでを指す．プログラムの修正を行うためにはまず欠陥が存在する箇所，すなわちソースコードの実装や機能的要求などの誤りを特定する必要がある．

簡単に欠陥を特定する手段としてテストケースの使用が挙げられる．テストケースとは入力値，期待値，実値の組を指し，ある入力値に対する出力値 (実値) が要求される値 (期待値) と等しいか否かを判定するものである．複数のテストケースを実行した上で，どの入力値を与えた場合に誤った動作となるかを把握することで欠陥を特定し易くなる．また，十分な数のテストケースがあり，すべてのテストケース通過するならば，そのソフトウェアは信頼性が高いと判断できる．理想的なテストケースはソフトウェアの全動作パターンを確認するものであるが，そのようなテストケースの作成や実行は膨大な時間や労力を必要とするため現実的ではない．したがって，実践上では到達可能な実行パスを網羅する程度に留める場合が大半である．しかし，その場合でも十分なテストケースを手動で用意するには多大な労力を要するため，テストケースを自動で生成する手法が提案されている [2, 16]．

以降，本研究ではデバッグ開始時のソフトウェアが通過するテストケースを通過済テストと呼び，通過しないテストケースを未通過テストと呼ぶ．通過済テストは正しい挙動を示し，未通過テストは誤った挙動すなわち欠陥を示す．

2.2 欠陥箇所の限局

デバッグを支援する手法の 1 つに欠陥箇所の限局がある．欠陥箇所の限局手法はソースコードを解析して欠陥の候補を探す手法であり，最も欠陥の可能性が高い箇所を開発者に提示することでデバッグを補助する．欠陥箇所の限局手法には，テストケース毎の実行パスから算出した疑惑値に基づいて順序付けを行う手法が存在する [17]．疑惑値とは，欠陥である可能性の高さを表す値である．このような手法では，通過済みテストのみが実行する文は疑惑値が低くなり，未通過テストのみが実行する文は疑惑値が高くなる．また，GenProg およびその関連手法のうちいくつかはプログラムの変更を行う際，欠陥箇所の限局を行い，疑惑地の高いプログラム文を優先的に変更する．

2.3 遺伝的プログラミング

遺伝的プログラミングとは自然界の生物が環境に適応して進化していく過程を模した探索アルゴリズムである [11]．データ（解の候補）を生物の個体に見立て，生物の進化の過程で行われる事象をモデル化した遺伝的操作と呼ばれるいくつかの操作を繰り返し適用することで，環境に適した，できるだけ最適解に近い個体を生成することを目的とする．

遺伝的プログラミングでは，まず一定数の個体をランダムに生成し，これらを第 1 世代とする．これらの個体に対して遺伝的操作を適用することで次世代の個体群を生成する．同様にして新たな世代の個体群の生成を繰り返す．このような処理を，条件を満たす個体が生成されるか，あらかじめ定められた世代数に到達するまで繰り返す．

遺伝的プログラミングで一般的に行われる遺伝的操作は以下の 3 つである．

選択 生物の自然淘汰をモデル化したもので，環境への適応度に基づいて一定数の個体を取り出す．環境への適応度は解への近さを表す値で，評価関数に基づき算出される．評価関数は実装により異なる．

変異 生物に見られる遺伝子の突然変異をモデル化したもので，個体に対して何かしらの変更を加える．

交叉 生物が交配によって子孫を残すことをモデル化したもので，2 つの個体を混ぜ合わせた新たな個体を生成する．

2.4 GenProg

GenProg は遺伝的プログラミングに基づいて自動的にプログラムの修正を行う手法である [9]．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て受け取り，再利用に基づく自動プログラム修正を行う．出力はテストスイートに含まれるすべてのテストケースを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のこと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のことを修正済みプログラムと呼ぶ．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．

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，欠陥箇所に変異操作を行ったプログラム（以降，個体と呼ぶ）を複数生成し，これらを第 1 世代とする．変異操作では，次の 3 処理のうちいずれか 1 つを行う．

挿入 欠陥を含む行の直後に，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操作

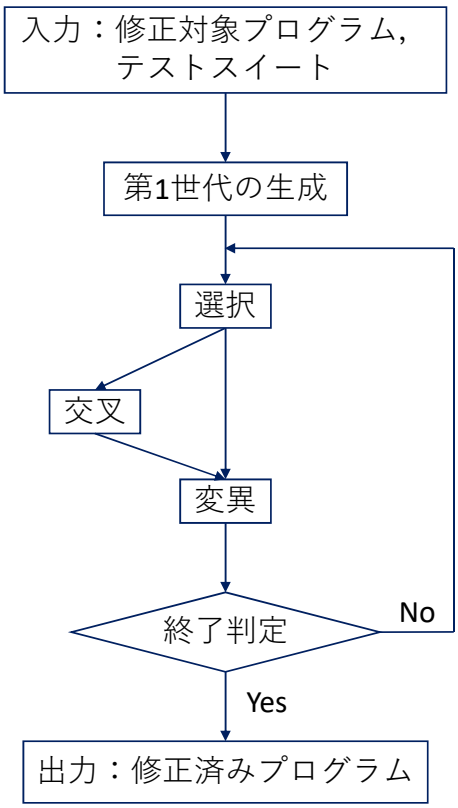

図 1　GenProg の動作の流れ

削除　欠陥を含む行を削除する操作

置換　修正対象プログラムからランダムに選択された行によって，欠陥を含む行を上書きする操作（挿入＋削除）

次に，評価関数に基づいて各個体の適応度の評価を行い，評価値の高いものを一定数残してそれ以外を削除する．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．

次世代の個体群に対してテストを行い，すべてのテストケースを通過する個体があれば，それを修正済みプログラムとして出力する．そのような個体が存在しなければ，選択操作からやり直す．この処理をすべてのテストケースを通過する個体が生成されるか，あらかじめ定められた世代数に到達するまで繰り返す．

GenProg では，ソースコード中に存在する行を用いて欠陥を修正できると仮定している．Barr ら

は実際に行われた変更を基に仮定の正しさを検証するために調査を行った [18]．調査の結果，ソースコード中の行を用いることで，10% の変更において追加された行のすべての行を記述することができ，42% の変更において追加された半数以上の行を記述できることが分かった．

2.5 再利用に基づく自動プログラム修正手法の課題点

挿入操作および置換操作で挿入するプログラム文は，修正対象プログラム中からランダムに選択する．そのため，プログラムの規模が大きくなれば，修正に寄与するプログラム文を選択するまでに行う変異操作の数が増大し，欠陥修正に膨大な時間を必要とする．また再利用に基づく自動プログラム修正手法では，修正候補の探索を一定の時間もしくは世代数で打ち切る．したがって，探索回数や修正時間が増大することによって，多くの探索や修正時間を必要とする欠陥を修正できなくなってしまう．

したがって，修正時間の増大を抑えて効率的に修正に寄与するプログラム文を選択できるような手法が求められる．

3　研究目的

本章では，本研究に至った背景および本研究の目的について述べる．

3.1　これまでの取り組み

本研究に至った背景として，本研究グループの横山らと山本らが行った研究がある．本節では，これらの研究について説明する．

3.1.1　横山らの研究

本小節では，既存の再利用に基づく自動プログラム修正手法が抱える課題の解決に向けて調査した横山らの研究 [12] について説明する．はじめに横山らの研究で用いられた用語を説明する．

挿入候補　変異処理において挿入の対象となりうる行の集合
挿入行　挿入候補のうち，実際に挿入された行
周辺領域　ある行を中心とした前後数行

これらの用語に加えて以降，中心行という用語を用いる．これは，周辺領域の中心となる行のことである．横山らは先述した再利用に基づく自動プログラム修正手法が抱える課題を改善するために，ソースコードの行に優先度付けを行い，優先度順に挿入することを提案した．

横山らは，実際に行われた欠陥の修正の例を基に，挿入候補のうち，その周辺領域と欠陥が存在している行の周辺領域が類似している場合は，欠陥が修正できる可能性が高いと考えた．欠陥箇所と類似した周辺領域の中心行によって欠陥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に領域間の類似度を用いることが有用となる．横山らは，領域間の類似度を用いて優先度付けを行うことが有用であるかを確認するために以下のような調査項目を設定した．

RQ:　挿入行を含む領域と欠陥箇所の周辺領域の類似度は，挿入行を含まない領域と欠陥箇所の周辺領域の類似度よりも高くなる傾向はあるか．

調査では実際に行われた欠陥修正の情報を用いて類似度の比較を行った．実際に行われた欠陥修正の情報はオープンソースソフトウェアのソースコードリポジトリから取得した．欠陥の修正が行われたかどうかについては，コミットログから判断する．修正に関するキーワードである，“fix”, “fixed”, “fixing” のいずれかがコミットログに含まれているもの（以降，修正コミットと呼ぶ）を調査の対象とした．さらに，修正コミットにおいて追加された行を挿入行とした．横山らの研究において，修正コミットで追加されたすべての行は欠陥の修正に用いられていたものとして仮定している．

類似度の算出方法について述べる．横山らの調査では類似度をレーベンシュタイン距離を用いて算出している．レーベンシュタイン距離とは，2 つの文字列に対して片方の文字列を他方の文字列へ編集するのにかかる最小手数を表す．文字列の編集には文字の挿入，削除，および置換があり，各操作は 1 回につき 1 手として手数をカウントする．横山らの研究では，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，トークン単位の編集距離を求める．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大きくなる．そこで，様々な長さの文字列（トークン列）に対する類似度を公平に比較するため，値の正規化を行う．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．これらのことを考慮して 2 つのトークン列 t_1, t_2 に対する類似度順 $s(t_1, t_2)$ を以下のように定める．

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

ここで，$dist(t_1, t_2)$ は t_1 と t_2 のレーベンシュタイン距離，$len(t_1), len(t_2)$ はそれぞれ t_1, t_2 のトークン列の長さを表す．$dist(t_1, t_2)$ の範囲は，$0 \leq dist(t_1, t_2) \leq max\{len(t_1), len(t_2)\}$ であるため，$s(t_1, t_2)$ の範囲は $0 \leq s(t_1, t_2) \leq 1$ となる．2 つのトークン列のレーベンシュタイン距離が小さくなるほど，つまり 2 つのトークン列が類似しているほど $s(t_1, t_2)$ は 1 に近づくことになる．

RQ に対する調査は表 1 に示す 4 つのオープンソースソフトウェアのソースコードリポジトリを対象に行われた．調査において周辺領域の行数は中心行の前後 5 行とされた．類似度順を算出した結果，Apache httpd, CBMC, JabRef において明確な差が見られた．jEdit に関しては統計検定を用いて有意差を確認した．横山らは調査の結果から RQ に対して Yes と回答した．

さらに横山らは，類似度順が挿入候補を選択する評価基準として有用であるか確認するために追加調査を実施した．追加調査では，挿入行のカバレッジを算出した．カバレッジの定義を以下に示す．

$$\text{カバレッジ} = \frac{\text{挿入行の類似度順位}}{\text{ソースコードの総行数}}$$

ただし挿入行の類似度順位とは，プログラム中のすべての行を類似度が高い順に並べたときに，その挿入行が何番目に位置するかという値である．カバレッジを算出することで，評価基準に従って挿入候

表 1　横山らの研究の調査対象ソフトウェア

ソフトウェア	言語	開始リビジョン (日付)	終了リビジョン
Apache httpd	C	1,410,755 (2012/11/18)	1,421,851 (2012/12/14)
CBMC	C++	3,602 (2014/02/22)	3,719 (2014/04/14)
JabRef	Java	3,349 (2010/11/01)	3,474 (2011/03/18)
jEdit	Java	19,812 (2011/08/19)	20,292 (2011/11/11)

補を選択していったときに，挿入行を選択するまでに挿入候補全体の上位何 % を選択すればよいかが分かる．追加調査の対象は，RQ の調査で対象にしたソフトウェアにおける変更のうち，実際に欠陥の修正が行われたことを目視により確認したコミットにおける挿入行である．

追加調査の結果，類似度順を用いることで，調査した挿入行のうち 75% が，挿入候補の 10% の行を選択するまでに選択されていることが分かった．このことより，修正に寄与する行の大半は類似度順で効率的に選択できると考えられる．

3.1.2 山本らの研究

本小節では，類似度順が抱える課題点を受けて，挿入候補の優先度付けの新たな基準の提案および調査を行った山本らの研究 [13] に関して説明する．

まず類似度順が抱える課題点について説明する．類似度順により挿入行を選択することが効果的なのは，欠陥が存在している行の周辺領域と類似したコード片にその欠陥の修正に寄与する行が存在している場合である．欠陥の修正に寄与する行が類似したコード片に存在していない場合は，類似度順は効果的ではない．

このような課題点があるため，別の基準によるアプローチを行う必要がある．そこで，山本らは新たな基準として更新順を提案した．更新順とは，挿入候補をその最終更新日時がより最近のものから順に選択するという評価基準である．

更新順による挿入候補の選択手順について説明する．更新順は入力として，修正対象プログラムとテストスイートに加えて，修正対象プログラムの開発履歴を受け取る．開発履歴はバージョン管理システムにより取得する．入力された開発履歴を基に，修正対象プログラム中のすべての行の最終更新日時を取得する．そして，この最終更新日時がより最近のものから順に挿入候補を選択する．

山本らは，オープンソースソフトウェアで実際に行われた欠陥の修正の情報をもとに更新順の有用性を調査した．調査対象のソフトウェアを表 2 に示す．これらのプロジェクトにおいて実際に行われた欠陥修正に関する情報の取得には，横山らの調査と同様の方法を採用した．

表 2　山本らの研究の調査対象ソフトウェア

ソフトウェア	言語	開始リビジョン (日付)	終了リビジョン
Apache httpd	C	76,295 (2004/11/19)	1,722,377 (2015/12/31)
CBMC	C++	1 (2011/05/08)	6,211 (2015/12/30)
JabRef	Java	1 (2003/10/15)	3,718 (2011/11/11)
jEdit	Java	1 (2006/07/01)	24,280 (2015/12/31)

調査では，各修正コミットにおける挿入行のカバレッジを算出した．カバレッジは，横山らの追加調査と同様に以下のように定める．

$$\text{カバレッジ} = \frac{\text{挿入行の更新順位}}{\text{ソースコードの総行数}}$$

ただし挿入行の更新順位とは，プログラム中のすべての行を更新順に並べた際に，その挿入行が何番目に位置するかという値である．調査の対象は，表 2 に示した開始リビジョンと終了リビジョン間に含まれるすべての挿入行である．調査の結果，挿入行の約 6 割は上位 10% に位置することが分かった．さらに，同じ対象に対して類似度順を適用して比較を行ったところ，類似度順では優先的に選択されないが更新順では優先的に選択される行が多数存在することが分かった．また，更新順と類似度順の上位 10% を組み合わせることで挿入行の約 70% を見つけることができ，上位 20% を組み合わせることで挿入行の約 80% を見つけることができるということも分かった．これらのことより，更新順と類似度順を組み合わせて実行することで効率的に修正に寄与する行を選択できるのではないかと考えられる．

3.2 研究動機

前節で類似度順および更新順に関する本研究グループの取り組みについて述べたが，これらの手法はどちらもツールとして実装されていない．したがって実際の欠陥修正において，これらの手法が効率よく動作するのかということまでは分かっていない．そこで本研究ではまず，類似度順と更新順をツールとして実装する．そして，実装したツールを実際の開発現場で生じた欠陥に対して動作させることで，これらの手法が実社会で生じた欠陥を効率的に修正できるのかを評価する．

3.3 研究課題

本研究では，実社会で生じた欠陥の自動修正に対して類似度順と更新順がどの程度有効なのかを明らかにすることを目的とする．そのため，以下の 4 つの研究課題を設定した．

RQ1　類似度順はランダムに挿入候補を選択する手法よりも多くの欠陥を修正できるか．
RQ2　更新順はランダムに挿入候補を選択する手法よりも多くの欠陥を修正できるか．
RQ3　類似度順はランダムに挿入候補を選択する手法よりも高速に欠陥を修正できるか．
RQ4　更新順はランダムに挿入候補を選択する手法よりも高速に欠陥を修正できるか．

4 実装

本章では，どのように類似度順および更新順を実装したのかについて述べる．

4.1 Astor

類似度順と更新順の実装は， Astor [14, 15] という既存の再利用に基づく自動プログラム修正ツールに対して両手法を組み込むことで行った．本節では，この Astor の概要について説明する．

Astor は次のような背景の下開発された．近年，自動プログラム修正に関する研究が盛んに行われるようになり，多くの自動欠陥修正手法が提案され，それらの手法を実現するツールも開発されてきた．だが，それらのツールの一部は公開されておらず，その研究論文中で述べられている実験の再現や，新たに開発した自動修正手法との比較実験の実施が困難なことがあった．そこで，Astor はオープンソースで開発が進められ，誰もが使用できるよう公開されている．さらに，Astor は自動プログラム修正手法のプラットフォームとなることも目的としてる．Astor は拡張が行いやすいように設計されており，Astor に対して独自の拡張を行うことにより新たな自動プログラム修正手法の実装を容易に行えるようになっている．

Astor には GenProg [10], Kali [19] および MutRepair [14, 20] の 3 つの自動プログラム修正手法が実装されており，どれか 1 つを選んで実行することができる．これらの自動プログラム修正手法を実現するツールはすでに存在していたが，これらのツールが修正対象とするプログラムの記述言語は C 言語であった．一方，Astor は Java を対象としている．Astor における GenProg, Kali および MutRepair の Java 向け実装はそれぞれ jGenProg2, jKali および jMutRepair と呼ばれている．本研究では jGenProg2 に対して拡張を行うことで類似度順と更新順を実装した．

jGenProg2 には独自の手法の実装を容易に行えるように IngredientSearchStrategy という抽象クラスが用意されている．これは変更操作に用いるプログラム文の選択アルゴリズムを表現するための抽象クラスである．この抽象クラスには抽象メソッドが 1 つ定義されており，このメソッドの実装によって，変更操作に用いるプログラム文をどのようなアルゴリズムに基づいて選択するかを規定することができる．このメソッドは，どこにどんな変更操作を行うのかという情報を受け取り，その変更操作に用いるプログラム文を返すメソッドである．各変更操作は，変更を行う箇所（欠陥限局箇所）と変更操作の種類の組によって一意に識別できる．以降，この組のことを変更クエリと呼ぶ．jGenProg2 では，どのような変更クエリに対しても，修正対象プログラム中からランダムに選択してきたプログラム文を返している．ただし，同一のクエリに対して一度適用したプログラム文は再度返さないようになっている．

類似度順と更新順のそれぞれの絞り込み基準に従ってプログラム文を選択するような IngredientSearchStrategy の子クラスを作成することで，類似度順と更新順を実現できる．以降，類似度順

と更新順を実現する IngredientSearchStragety の子クラスをそれぞれ SimilaritySearch, ChronologicalSearch と呼ぶ.

4.2 類似度順の実装

SimilaritySearch は類似度順位表を持つ．類似度順位表とは，修正対象プログラムの各行を，その行の周辺領域と欠陥限局箇所の周辺領域との類似度が高い順に並べたものである．欠陥限局箇所は複数存在する可能性があるので，欠陥限局箇所ごとに類似度順位表を作成する．なお，類似度順位表の作成は修正対象プログラムへの変異処理を開始する前に 1 度だけ行う．SimilaritySearch は変更クエリを受け取るごとに，変更箇所に対応する類似度順位表の上から順にプログラム文を返す．

4.3 更新順の実装

ChronologicalSearch は修正対象プログラムの更新履歴を入力として受け取る．この更新履歴を基に更新順位表を作成する．更新順位表とは，修正対象プログラムの各行を，その行の最終更新日時が最近のものから順に並べたものである．ChronologicalSearch は変更クエリごとに，更新順位表の上から順にプログラム文を返す．

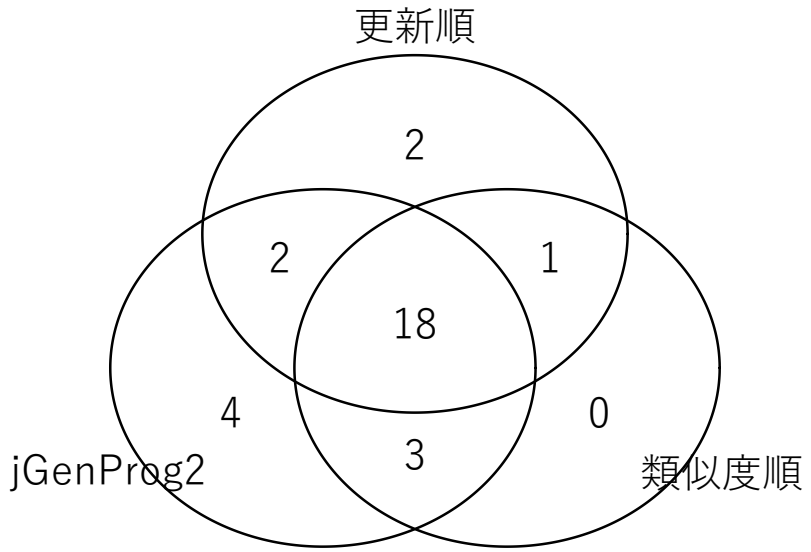

図 2　各種法によって修正できた欠陥の個数

5　評価実験

前章では，類似度順と更新順をどのように実装したのかについて述べた．本研究ではこの実装したツールを用いて，類似度順および更新順が実際の欠陥修正において有効に動作するのかを評価するために実験を行った．本章では，この評価実験について述べる．

5.1　実験設定

類似度順と更新順の有用性を評価するために，類似度順，更新順および jGenProg2 を実際の欠陥に対して動作させ，その結果を比較した．評価項目は以下の 2 つである．

修正欠陥数　修正することのできた欠陥の個数
修正時間　修正済みプログラムが得られるまでに要した時間

今回の実験の対象は Apache Commons Math [21] というオープンソースで開発されているソフトウェアの開発過程で生じた 106 個の欠陥である．これらの欠陥は Defects4j [22] という欠陥のデータセットより取得した．

5.2 実験結果

5.2.1 修正欠陥数

各手法によって修正できた欠陥の個数を図 2 に示す．図から分かる通り，jGenProg2 には修正できなかったが類所度順または更新順では修正することができた欠陥が存在する．つまり，jGenProg2 に加えて類似度順や更新順を用いることで修正できる欠陥の数を増やすことができる．

ただし，以下に示すとおり各手法によって修正できた欠陥の個数を比較すると，類似度順，更新順ともに jGenProg2 よりも多くの欠陥を修正することはできなかったことが分かる．

jGenProg2　27 個
類似度順　22 個
更新順　23 個

このことから，RQ1, RQ2 への回答はともに No である．

5.2.2 修正時間

jGenProg2，類似度順，更新順のすべてが修正できた欠陥に関して，各手法の修正時間の分布を図 3 に示す．この結果を基に，類似度順および更新順が jGenProg2 よりも高速に欠陥を修正できるのかを調べるため，統計的手法により比較を行った．

まず，データの正規性を検証するためコルモゴロフ・スミノフ検定を行った．その結果，jGenProg2, 類似度順，更新順に対する p-値はそれぞれ約 1.8×10^{-2}，約 3.0×10^{-3}，約 6.0×10^{-2} となり，jGenProg2 と類似度順には正規性がないことが分かった．そこで，正規性のない対応のある 2 群の有意差を検証するため，ウィルコクソン符号順位検定を jGenProg2 と類似度順，jGenProg2 と更新順の 2 組に対してそれぞれ行った．その結果，jGenProg2 と類似度順の組に対する p-値は約 8.2×10^{-1}，jGenProg2 と更新順の組に対する p-値は約 3.3×10^{-1} となり，有意差がないことが分かった．ただし，ウィルコクソン符号順位検定で有意差なしと判定されても t 検定で有意差が検出されることがある．そこで，この 2 組に対してそれぞれ対応のある t 検定も行った．その結果，jGenProg2 と類似度順の組に対する p-値は約 7.4×10^{-1}，jGenProg2 と更新順の組に対する p-値は約 1.9×10^{-1} となり，2 組ともほぼ確実に有意差がないことが分かった．

以上の結果より，RQ3, RQ4 への回答はともに No である．

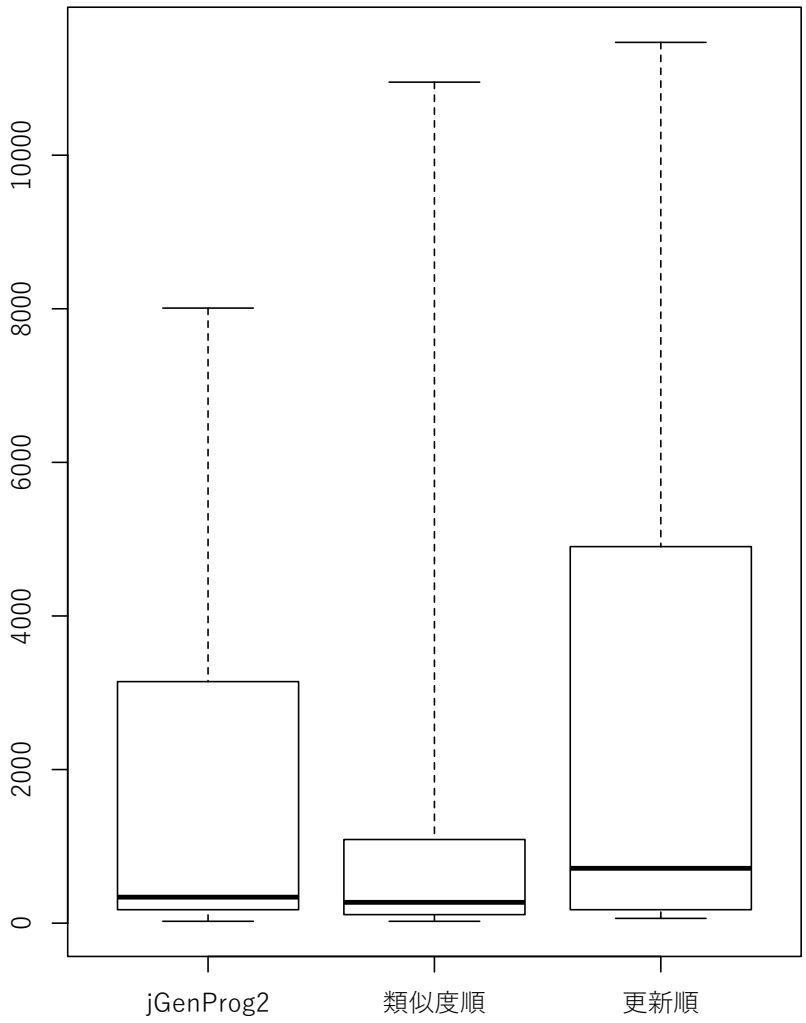

図 3　3 手法が修正した欠陥の修正時間の箱ひげ図

6　考察

実験結果より，類似度順および更新順は jGenProg2 に比べて修正欠陥数に関しては勝っているとは言えず，修正時間に関しては有意な差が見られなかった．つまり，類似度順および更新順は jGenProg2 よりも有用性が高いとは言いない．しかし，類似度順と更新順が有用となる場合もある．修正欠陥数に関しては，図 2 に示した通り，jGenProg2 には修正できなかったが類似度順もしくは更新順では修正できる欠陥も存在する．また修正時間に関しても，jGenProg2 では修正に要する時間が比較的長いが類似度順もしくは更新順では比較的短時間で修正できる欠陥も存在する．このように，類似度順や更新順で効率よく修正できる欠陥に対しては，ランダムに挿入候補を選択する手法よりも有用である．

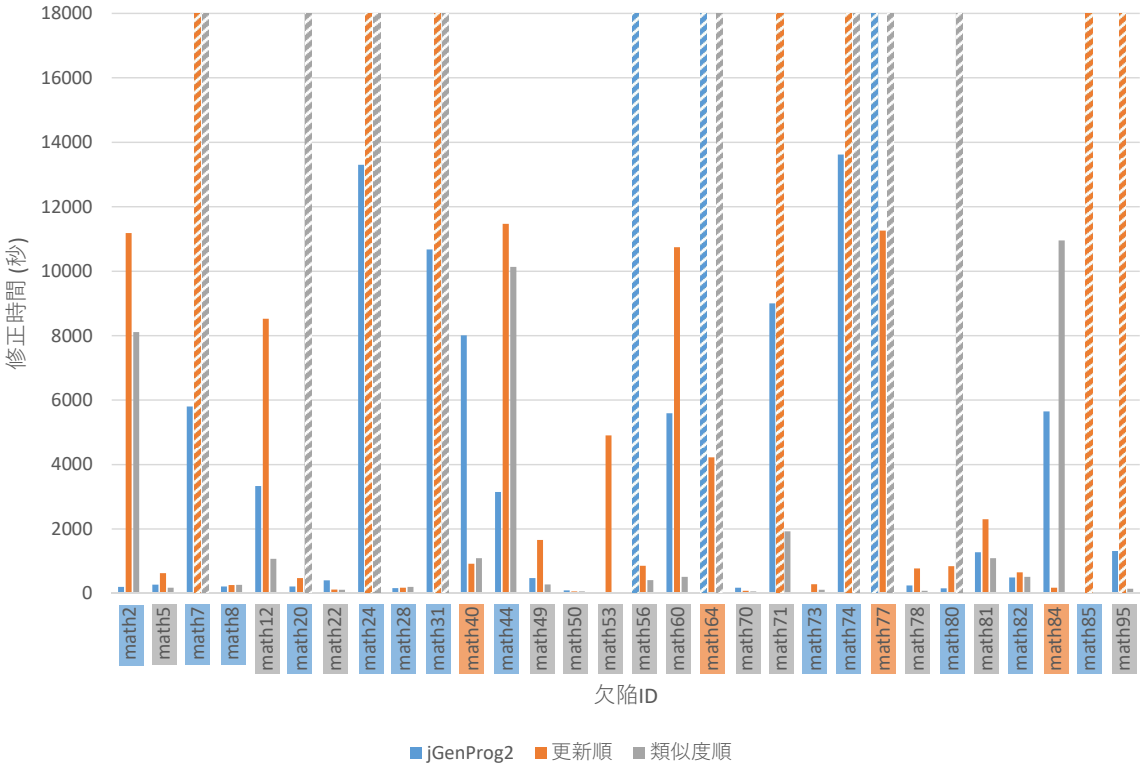

図 4　修正時間

修正時間の評価実験では，対応のある 2 群の検定を行った．これは，同一の欠陥に対して jGenProg2 と類似度順（もしくは更新順）を動作させるという試行を多数行った際に，これらの手法間で修正時間に差があるかを確かめるものである．この結果，有意な差は見られなかった．そこで，3 手法のうち少なくとも 1 つの手法によって修正できた欠陥 30 個について，欠陥ごとにそれぞれの手法がどの程度の時間でその欠陥を修正できたのかを比較した．その結果を図 4 に示す．ただし，グラフ中の網掛けが施されているものは，その欠陥をその手法によって修正できなかったことを表している．修正できなかった欠陥に関しては，タイムアウト時間に到達して終了したものと，修正途中で例外が発生して終了したものとがある．また，横軸の各欠陥 ID には色が着けてあるが，この色はその欠陥を最も早く修正した手法の色に対応している．

欠陥ごとに 3 手法の修正時間を比べてみると，3 手法の修正時間の傾向によってこれらの欠陥を大きく 2 つのグループに分けることができる．1 つは，3 手法ともに数分から数十分程度と比較的短時間で修正できているグループ，もう 1 つは，修正に比較的長時間を要している（もしくは，修正できていない）手法がある一方，比較的短時間で修正を終えている手法があるグループである．

後者の欠陥群について注目してみると，jGenProg2 の数分の 1 程度の時間で修正できているものが類似度順，更新順の両方に対して存在する．例えば，math60 については，類似度順は jGenProg2 の

10 分の 1 程度の時間で修正できており，math84 については，更新順は jGenProg2 の 30 分の 1 程度の時間で修正できている．このように欠陥によっては，1 つの手法では修正に時間がかかっても他の手法では比較的高速に修正できることがある．また，ある手法では修正できなくても別の手法では修正できる欠陥もある．つまり，jGenProg2，類似度順および更新順はある程度相補的な関係にあると言える．したがって，修正したい欠陥に応じて，その欠陥を効率よく修正できる手法を選択して実行できれば望ましい．しかし，欠陥を効率よく修正できる手法を事前に判断するのは困難だと考えられる．そこで，jGenProg2, 類似度順および更新順を一定時間ごとに切り替えながら実行することで修正時間を平均的に短縮することができるのではないかと考えられる．

例えば，jGenProg2，類似度順，および更新順がそれぞれ 4 時間，2 時間，および 20 分で修正できる欠陥に対して 3 手法を 10 分ごとにローテーションしながら実行することを考える．まず，jGenProg2 を 10 分実行するが欠陥は修正できないので，jGenProg2 を中断し，類似度順を 10 分実行する．ここでも，欠陥は修正できないので更新順を 10 分実行するということを繰り返していく．すると，2 巡目に更新順を実行したところで欠陥が修正できる．このときの修正時間は 1 時間となり，もともと修正時間が最短であった更新順の 20 分よりは修正時間が長くなってしまうが，jGenProg2 および類似度順と比べると修正時間をそれぞれ 4 分の 1 および 2 分の 1 に短縮できる．このように，3 手法をローテーションさせることで修正時間を平均的に短縮できる欠陥としては，math2, math60, math84 などがある．

また，1 つの手法で変異処理に用いた挿入候補を他の手法で再度用いないようにすれば重複してテストを実行する必要がなくなるため，その分さらに修正時間を短縮できる．

しかし，最短で修正を終える手法の修正時間がタイムアウト時間の 3 分の 1 程度よりも長ければ，ローテーションすることによってタイムアウト時間を超えてしまい，その欠陥を修正できなくなってしまうことも考えられる．

7　妥当性への脅威

本研究の実験対象プロジェクトは Apache Commons Math のみである．プロジェクトが変われば発生する欠陥の傾向も変わる可能性が考えられるため，その他のプロジェクトで生じた欠陥に対しても実験を行うべきである．

本研究では，既存の再利用に基づく自動プログラム修正ツールである Astor に対して拡張を行うことで類似度順と更新順の実装を行った．Astor は実行時引数として乱数のシード値を指定する．今回はこのシード値を 0 で固定して実験を行ったが，シード値を変えることで実験結果が変わる可能性がある．

8 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．

8.1 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 [23]．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 [10]．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，修正に要する時間が長いという問題がある．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 [24]．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．

8.2 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 [25]．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 [26]．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．

8.3 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 [27]．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基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 [28].

9　あとがき

本研究では，類似度順と更新順の実装を行い，これらの手法の有用性を評価するために実験を行った．欠陥修正数に関しては，類似度順と更新順ともにランダムに修正候補を選択する手法よりも多くの欠陥を修正することはできないことが分かった．しかし，ランダムな手法では修正できないが類似度順または更新順では修正できる欠陥が存在することが分かった．修正時間に関しては，更新順と類似度順ともにランダムに挿入候補を選択する手法との有意な差は見られなかった．また，ランダムな手法，類似度順および更新順をローテーションさせることで修正時間を短縮できる場合があることを示した．

今後の課題は，類似度順，更新順およびランダムに挿入候補を選択する手法をローテーションさせながら実行する手法を実装し，欠陥修正に要する時間を短縮できるかどうか調査することである．また，本研究で用いた Astor 以外の再利用に基づく手法へ，類似度順および更新順のアイディアを組み込めるか検討することも必要である．

謝辞

本研究を行うにあたり，親身なご指導を賜り，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．

本研究の全過程を通し，研究に関する考え方や方向性など，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およびご指導を頂きました，柗本 真佑 助教に深く感謝申し上げます．

本研究を進めるにあたり，適切なご助言および多大なるご助力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 年の鷲見 創一 氏，同 横山 晴樹 氏に心より感謝申し上げます．

本研究を進めるにあたり，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] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[2] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, No. 3, pp. 215–222, 1976.

[3] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, pp. 75–84, 2007.

[4] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282, 2005.

[5] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the International Conference on Software Engineering*, pp. 45–55, 2009.

[6] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. Mimic: Locating and understanding bugs by analyzing mimicked executions. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 815–826, 2014.

[7] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pp. 467–477, 2002.

[8] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pp. 89–98, 2007.

[9] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[10] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of aatomated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the International Conference on Software Engineering*, pp. 3–13, 2012.

[11] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural*

Selection, Vol. 1. MIT press, 1992.

[12] 横山晴樹, 大田崇史, 堀田圭佑. 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 (ソフトウェアサイエンス). 電子情報通信学会技術研究報告, Vol. 115, No. 20, pp. 47–52, 2015.

[13] 山本将弘, 横山晴樹, 肥後芳樹, 楠本真二. 再利用に基づく自動プログラム修正における更新順を用いた挿入候補の絞込の提案 (ソフトウェアサイエンス). 電子情報通信学会技術研究報告, Vol. 115, No. 508, pp. 79–84, 2016.

[14] Matias Martinez and Martin Monperrus. ASTOR: Evolutionary Automatic Software Repair for Java. Technical report, Inria, 2014.

[15] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 441–444, 2016.

[16] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, pp. 416–426, 2007.

[17] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[18] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 306–317, 2014.

[19] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 24–36, 2015.

[20] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 65–74, 2010.

[21] Commons Math Developers. Apache commons math. *Forest Hill, MD, USA: The Apache Software Foundation*, 2012.

[22] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[23] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference*

on Software Engineering, pp. 364–374, 2009.

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

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

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

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

[28] 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 International Conference on Software Engineering*, pp. 234–242, 2014.