特別研究報告

題目

自動プログラム修正における更新履歴データを用いた挿入候補選択手法

指導教員
楠本 真二 教授

報告者
山本 将弘

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

自動プログラム修正における更新履歴データを用いた挿入候補選択手法

山本 将弘

内容梗概

ソフトウェア開発において，プログラムの信頼性を向上させるためにデバッグを行う必要がある．デバッグは作業コストがかかるため，支援するためにプログラム修正を自動化することが望ましい．自動プログラム修正の研究において，既存のソースコード中の行を再利用して欠陥の修正を行う研究が行われ，様々な手法が提案されている．そのような再利用に基づく自動プログラム修正手法では，挿入する行を修正対象プログラムのソースコード全体からランダムに選択する．ランダムな選択では，プログラムの規模が拡大するにつれて欠陥の修正に寄与しない行が選択される回数が多くなり，修正が完了するのに時間がかかってしまう可能性がある．そのため，欠陥の修正に寄与する行を優先して選択するような基準を提案する必要がある．既存研究では類似度順という基準が提案されている．類似度順とは，ソースコード中の各行を，その行の周辺コードと欠陥が存在する箇所の周辺コードの類似度が高い順で選択する基準である．既存研究における調査の結果，類似度順が有用であることがわかっている．しかし，欠陥の修正に寄与する行であるにも関わらず，類似度順ではその行の優先度が低くなってしまう場合があることが確認されている．そのため，類似度順とは別の基準を提案する必要がある．

そこで本研究では，更新順という基準を提案する．更新順とは，ソースコード中の各行を更新日時の新しい順で選択する基準である．各行の更新日時はバージョン管理システムより取得する．本研究では，オープンソースソフトウェアで過去に行われた欠陥の修正において挿入された行を用いて，更新順の有用性を調査した．調査では，欠陥の修正において挿入された行のうち，行の再利用により選択できる行を対象とした．調査の結果，対象の行のうち約 6 割の行が更新順で挿入候補の 10%の行を選択するまでに選択されていることがわかった．また，同じ調査対象を用いて更新順と類似度順の比較を行い，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わかった．

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

1　まえがき **1**

2　関連研究 **3**

2.1　GenProg 3

2.2　RSRepair 4

2.3　横山らの研究 5

2.4　類似度順による選択の課題点 7

3　アプローチ **9**

4　調査 **10**

4.1　調査目的 10

4.2　調査手順 10

4.3　調査対象 12

5　結果と考察 **13**

5.1　更新順の有用性の評価 13

5.2　更新順と類似度順の比較 14

6　妥当性への脅威 **20**

7　あとがき **21**

謝辞 **22**

参考文献 **23**

図 目 次

1 GenProg の動作の流れ 3
2 類似度順で挿入する行を選択するのが適さない欠陥の修正の例 8
3 修正コミットごとの処理 11
4 更新順のカバレッジの分布 14
5 更新順と類似度順の関係（Apache httpd） 15
6 各基準でカバレッジの小さい挿入行の集合によるベン図（Apache httpd） 15
7 更新順と類似度順の関係（CBMC） 16
8 各基準でカバレッジの小さい挿入行の集合によるベン図（CBMC） 16
9 更新順と類似度順の関係（JabRef） 17
10 各基準でカバレッジの小さい挿入行の集合によるベン図（JabRef） 17
11 更新順と類似度順の関係（jEdit） 18
12 各基準でカバレッジの小さい挿入行の集合によるベン図（jEdit） 18

表 目 次

1 横山らの研究の調査対象ソフトウェア 6
2 本研究の調査対象ソフトウェア 12
3 各ソフトウェアにおける挿入行の数 13
4 更新順と類似度順の組み合わせによる挿入行の割合 19

1　まえがき

ソフトウェア開発において，プログラムの信頼性を向上させるためには，プログラムの欠陥を取り除くデバッグを行うことが必要である．デバッグには，テストによるプログラムの欠陥の検出，欠陥箇所の特定，欠陥箇所の修正といった工程が含まれる．ソフトウェア開発の過程において欠陥は頻繁に発生するため，デバッグは必要不可欠な作業である．デバッグは多大なコストを必要とし，文献 [1] ではソフトウェア開発者は開発時間の半分をデバッグに費やすと報告されている．デバッグは必要不可欠な作業であるにも関わらず，多大なコストを要するため，コストを削減するような支援が求められる．

デバッグの支援を行うため，デバッグの工程の自動化に関する様々な研究が行われている．テストを実施する工程において，ソフトウェア開発者は欠陥の検出にテストケースを利用している．テストケースには開発者がプログラムに期待する動作の入出力の組み合わせが記述されている．プログラムがテストケースを通過するということは，そのプログラムが開発者の期待する動作を行うことを示す．また，プログラムがテストケースを通過しないということは，そのプログラムが開発者の期待する動作をしないことを示す．プログラムが開発者の期待する動作をしないということは，プログラムが欠陥を含んでいるということである．プログラムの欠陥の検出を十分に行うためには，莫大な数のテストケースを作成する必要があり，これを手動で行うと多大なコストがかかる．そのため，テストケースを自動生成する研究が行われている [2–5]．開発者はテストケースを用いてプログラムの欠陥を検出した後，欠陥箇所の特定を行う必要がある．欠陥箇所はプログラム中に複数存在することもあるため，手動で欠陥箇所の特定を行うにはコストがかかる．そのため，欠陥箇所の特定を自動的に行う研究が行われている [6–8]．欠陥の修正を行う過程では，開発者がプログラムの動作を理解し，どのようにソースコードを変更すれば欠陥を取り除くことができるかを判断する必要がある．欠陥の修正にもコストがかかるため，欠陥の修正の自動化が求められている．

近年，欠陥の修正の自動化に関する研究が盛んに行われている [9–12]．欠陥の修正の自動化を行う手法は 2 つに大別される．1 つはヒューリスティックなアルゴリズムを用いて欠陥の修正を行う確率的な手法である．もう片方はプログラムが満たすべき性質を表す論理式を基にして欠陥の修正を行う手法である．論理式を基にする修正手法では，複雑な論理式を SMT ソルバ [13] で解く必要がある．そのため，現在の CPU の処理能力では，現実的な時間内に論理式を解くことは困難であるという課題点が存在する．

確率的な手法のうち，GenProg [9] が有望視されている．GenProg はソースコード中の行を利用して，遺伝的アルゴリズムによる自動プログラム修正を行う．それに対して Qi らは，自動プログラム修正における遺伝的アルゴリズムの必要性を疑問視し，RSRepair を提案した [10]．RSRepair は遺伝的アルゴリズムを用いず，ランダムにソースコードを変更する作業を繰り返すことでプログラムの修正を行う．以降，本研究では GenProg や RSRepair などの，ソースコード中の行を用いて欠陥の修正を行う手法を，再利用に基づく自動プログラム修正手法とよぶことにする．

再利用に基づく自動プログラム修正手法はソースコード中の行をランダムに選択して用いる．ランダムな選択では，修正対象のプログラムの規模が大きくなるにつれて欠陥の修正に寄与しない行を選択する回数が多くなり，修正が完了するのに時間がかかってしまう可能性がある．そのため，欠陥の修正に寄与する行が可能な限り早く選択されるような基準が必要とされている．横山らは，類似度順という基準を提案した [14]．類似度順とは，ソースコード中の各行を，その行の周辺コードと欠陥が存在する箇所の周辺コードの類似度が高い順で選択する基準である．横山らの研究ではオープンソースソフトウェアで過去に行われた欠陥の修正において挿入された行を用いて，類似度順の有用性を調査した．調査の結果，類似度順が行を選択する基準として有用であることを示した．しかし，欠陥の修正に寄与する行であるにも関わらず，その行の周辺コードと欠陥の存在する箇所の周辺コードが類似していない場合があることが確認されている．

本研究では更新順という基準を提案し，有用性の調査を行った．更新順とは，ソースコード中の各行を更新日時の新しい順で選択する基準である．調査は横山らの研究と同様に，オープンソースソフトウェアで過去に行われた欠陥の修正において挿入された行を用いて行った．調査の結果，対象の行のうちの 60%が更新順で挿入候補の 10%の行を選択するまでに選択できることがわかった．また，同じ対象に対して横山らの手法を適用し，比較を行った．その結果，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わかった．

以降，2 章では，本研究に関連した既存研究について述べる．3 章では，既存研究の課題点に対するアプローチを述べる．4 章では，調査の目的，手順，対象について述べる．5 章では，調査結果について述べ，考察を行う．6 章では，本研究における妥当性への脅威について述べる．最後にあとがきにおいて本研究をまとめる．

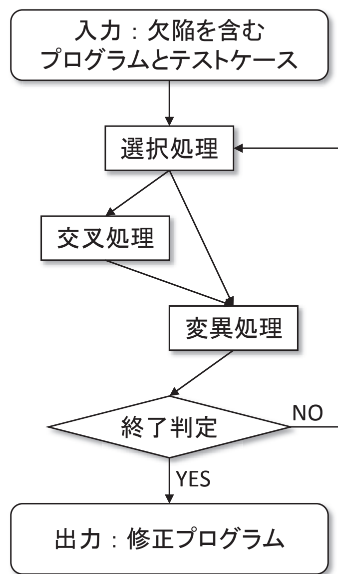


図 1: GenProg の動作の流れ

2 関連研究

本章では本研究に関連する研究について述べる．まずはじめに，再利用に基づく自動プログラム修正の研究として，GenProg [9] と RSRepair [10] を紹介する．その後，再利用に基づく自動プログラム修正が抱える課題点に関する既存研究 [14] について説明する．最後にその既存研究において提案された手法の課題点について述べる．

2.1 GenProg

GenProg [9] は遺伝的プログラミング [15] を用いて再利用に基づく自動プログラム修正を行う手法である．GenProg は入力として欠陥を含むプログラムとテストケースを受け取り，全てのテストケースを通過する修正プログラムを出力する．図 1 に GenProg の動作の流れを示す．

遺伝的プログラミングは，選択・変異・交叉の 3 つの処理を繰り返し行うことにより優秀な個体を生成する，生物の進化を模した手法である．選択・変異・交叉の 3 つの処理は，条件を満たす個体が生成されるまで繰り返し適用される．GenProg では，プログラムを個体とみなし，全てのテストケースを通過するプログラムが生成されるまで，3 つの処理を繰り返し適用する．

選択処理では，各個体のテストケース通過状況を基に適合値を求め，適合値が高い順に一定数の個体を取り出す．

変異処理では，各個体に対して変更を行う．変更を行う箇所は欠陥の原因として限局された行である．初期の GenProg [16] では，変更は削除・挿入・交換の 3 種類の操作である．削除操作では限局された行を削除する．挿入操作ではソースコード中からランダムに選択された行を限局された行のあ

とに挿入する．交換操作ではソースコード中からランダムに選択された行と限局された行の位置を交換する．このうち，交換操作は他の 2 つに比べて欠陥の修正に寄与する可能性が低いことが分かっている [17]．そのため，改良された GenProg [9] の変更は削除，挿入，置換の 3 種類の操作になっている．置換操作ではソースコード中からランダムに選択された行で欠陥の原因となる行を上書きする．すなわち，この置換操作による変更は削除操作を行ったのち挿入操作を行ったことと等しい．

交叉処理では，2 つの個体を混ぜあわせることにより，新たな個体を生成する．2 個体は個体の集合からランダムに選択される．

GenProg はオープンソースソフトウェアで過去に修正が行われた 105 個の欠陥のうち，55 個を自動的に修正できることに成功した [9]．

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全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わかった．

2.2 RSRepair

GenProg はその有用性をオープンソースソフトウェアの欠陥を修正することで示した．しかしながら，GenProg の有用性が遺伝的アルゴリズムによるものであるかどうかは示していない．Qi らは自動プログラム修正における遺伝的アルゴリズムの必要性を疑問視し，RSRepair を提案した [10]．RSRepair は遺伝的アルゴリズムを用いず，GenProg の変異処理と同様にソースコードを変更する作業を繰り返すことでプログラムの修正を行う．GenProg は個体の評価を行うため，テストケースの実行に多くの時間を要するという課題点がある．RSRepair では遺伝的アルゴリズムを用いないため個体の評価を行う必要が無い．そのため，テストケースを通過できなかった時点で個体を破棄し，残りのテストケースを実行する時間を削減できる．さらに，テストケースに優先順位をつける手法 [19] を用いてテストケースの実行に要する時間の短縮を図っている．

RSRepair の入力と出力は GenProg と同様であり，入力として欠陥を含むプログラムとテストケースを受け取り，全てのテストケースを通過する修正プログラムを出力する．RSRepair では，プログラム修正を行うとき，全てのテストケースに整数を付与する．個体がテストケースを通過しなかった場合，テストケースに付与した整数を 1 増やす．この整数は欠陥の検出しやすさを示し，値が大きいテストケースが優先して実行される．これは，多くの個体が通過しないテストケースほど，新たな個体も通過しない可能性が高いのではないかという考えに基づいている．元のソースコードから変異処理により個体を生成し，付与した整数が大きい順にテストケースを実行する．個体が全てのテストケースを通過すれば，その個体を修正プログラムとして出力し，修正を終了する．1 つでも通過しないテストケースがあれば，個体を破棄して次の個体の生成を行い，これらの処理を繰り返す．

RSRepair が変異を行うのは 1 箇所のみであるため，2 箇所以上の修正を必要とする欠陥を修正することができない．Qi らは，RSRepair の有用性を確認するために，GenProg [9] で対象とした欠陥

のうち 2 箇所以上の修正を必要とする欠陥を除いた 24 個の欠陥に対して RSRepair を用いてプログラム修正を行った．プログラム修正の結果，24 個中 23 個の欠陥において RSRepair の方が GenProg よりも生成された個体数が少ないことを示した．また，24 個中 23 個の欠陥において RSRepair の方が GenProg より実行したテストケースの数が少ないことを示した．

2.3 横山らの研究

本節では，再利用に基づく自動プログラム修正が抱える課題点に関する横山らの研究 [14] 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の定義を載せる．

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

再利用に基づく自動プログラム修正手法はソースコード中の行をランダムに選択して用いる．ランダムな選択では，プログラムの規模が大きくなるにつれ，挿入候補の数が増大し欠陥の修正に寄与しない行を選択する回数が増える．そのため，大規模なプログラムに対しては，プログラム修正に時間を要してしまうという課題点が存在する．この課題点を改善するために，横山らの研究では，ソースコードの各行に優先度付けを行い，優先度順に挿入する必要性があることを述べた．

横山らは，実際に行われた欠陥の修正の例を元に，ソースコードの行のうち，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．欠陥の存在する箇所と周辺領域が類似している行が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確認するために以下のような調査項目を設定し，調査を行った．

RQ: 挿入行を含む領域と欠陥の原因となる行の周辺領域の類似度は挿入行を含まない領域と欠陥の原因となる行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の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の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する．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”がコミットログに含まれているとき，そのコミットを修正コミットと呼ぶことにする．さらに，修正コミットにおいて追加された行を，挿入行とする．横山らの研究において，修正コミットにおいて追加された全ての行は欠陥の修正に用いられ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 [20] 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からもう片方の文字列への編集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，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，文字の代わりにトークンを用いている．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．そのため，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には，値の正規化を行う必要がある．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．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 $s(t_1, t_2)$ は以下のように定める．

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

ここで，$\mathrm{dist}(t_1, t_2)$ は t_1 と t_2 のレーベンシュタイン距離，$\mathrm{len}(t_1)$，$\mathrm{len}(t_2)$ はそれぞれ t_1，t_2 のトークン列の長さを表す．$\mathrm{dist}(t_1, t_2)$ の範囲は，$0 < \mathrm{dist}(t_1, t_2) < \max\{\mathrm{len}(t_1), \mathrm{len}(t_2)\}$ であるため，$s(t_1, t_2)$ の範囲は $0 < s(t_1, t_2) < 1$ 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 $s(t_1, t_2)$ は 1 に近づくことになる．

RQ に対する調査は表 1 の 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．調査において領域の行数は挿入行を中心とする前後 5 行とされた．類似度を算出した結果，Apache httpd， CBMC，JabRef においては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，挿入候補の行の周辺領域と欠陥箇所の周辺領域の類似度が高い順で選択する類似度順が，基準として有用であるかどう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．カバレッジとは，挿入行を選択するまで基準によって順番に選択した行の数を，ソースコードの総行数で割った値である．カバレッジを算出することで基準によって挿入候補の何%を選択すると挿入行が選択できるかがわかる．追加調査の対象は，RQ の調査で対象にしたプログラム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視認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，対象の行のうち 75%の行が挿入候補の 10%の行を選択するまでに選択されていることがわかった．

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（日付）
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)

2.4 類似度順による選択の課題点

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．図 2 に例を挙げる．この例は，オープンソースソフトウェアの Apache httpd のプログラムに対して実際に行われた欠陥の修正である．

図 2 では，ある欠陥に対して，代入文（灰色の行）を他のメソッドに移動することで修正を行っている．つまり，移動先に対する行の挿入操作と移動元の行の削除操作により欠陥の修正が完了する．この代入文の移動元と移動先の周辺コードは類似度が高くなく，横山らの手法では効果的に欠陥の修正を完了させせることができない．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．

mod_log_config.c

```
212  static apr_hash_t *log_hash;
     …
1154 static void log_pre_config(apr_pool_t *p, apr_pool_t *plog, apr_pool_t *ptemp)
1155 {
1156    static APR_OPTIONAL_FN_TYPE(ap_register_log_handler) *log_pfn_register;
1157
1158    log_hash = apr_hash_make(p);
1159    log_pfn_register = APR_RETRIEVE_OPTIONAL_FN(ap_register_log_handler);
1160    if (log_pfn_register) {
1161        log_pfn_register(p, "h", log_remote_host, 0);
            …
1188    }
1189 }
1190
1191 static void register_hooks(apr_pool_t *p)
1192 {
1193    ap_hook_pre_config(log_pre_config,NULL,NULL,APR_HOOK_REALLY_FIRST);
1194    ap_hook_child_init(init_child,NULL,NULL,APR_HOOK_MIDDLE);
1195    ap_hook_open_logs(init_config_log,NULL,NULL,APR_HOOK_MIDDLE);
1196    ap_hook_log_transaction(multi_log_transaction,NULL,NULL,APR_HOOK_MIDDLE);
1197
1198    APR_REGISTER_OPTIONAL_FN(ap_register_log_handler);
1199 }
```

(a) 修正前（r88903）

mod_log_config.c

```
212  static apr_hash_t *log_hash;
     …
1154 static void log_pre_config(apr_pool_t *p, apr_pool_t *plog, apr_pool_t *ptemp)
1155 {
1156    static APR_OPTIONAL_FN_TYPE(ap_register_log_handler) *log_pfn_register;
1157
1158    log_pfn_register = APR_RETRIEVE_OPTIONAL_FN(ap_register_log_handler);
1159    if (log_pfn_register) {
1160        log_pfn_register(p, "h", log_remote_host, 0);
            …
1187    }
1188 }
1189
1190 static void register_hooks(apr_pool_t *p)
1191 {
1192    ap_hook_pre_config(log_pre_config,NULL,NULL,APR_HOOK_REALLY_FIRST);
1193    ap_hook_child_init(init_child,NULL,NULL,APR_HOOK_MIDDLE);
1194    ap_hook_open_logs(init_config_log,NULL,NULL,APR_HOOK_MIDDLE);
1195    ap_hook_log_transaction(multi_log_transaction,NULL,NULL,APR_HOOK_MIDDLE);
1196
1197    /* Init log_hash before we register the optional function. It is
        …
1202    */
1203    log_hash = apr_hash_make(p);
1204    APR_REGISTER_OPTIONAL_FN(ap_register_log_handler);
1205 }
```

(b) 修正後（r88904）

図 2: 類似度順で挿入する行を選択するのが適さない欠陥の修正の例

3 アプローチ

前章において，横山らの研究における基準である類似度順とは別の基準を設ける必要性を述べた．そこで本研究では，行を選択する基準として，更新順というものを提案する．更新順の定義は以下の通りである．行の更新日時は，バージョン管理システムを利用して取得する．

更新順 ソースコード中の各行を更新日時の新しい順で選択する基準

例えば，既存のソースコードに対して，ある 1 行が追加された場合を考える．追加された行は更新日時が他の行よりも新しいため，直後に更新順を用いて欠陥の修正を行うときは最初に選択される．

実際に更新順により順に挿入候補から行を選択することでうまくいく例として，前章の既存研究の課題点の例が挙げられる．図 2 では，代入文（灰色の行）を他のメソッドに移動することで欠陥の修正を行っている．この行は図の修正の少し前のコミットにおいて新たに追加された行であり，それ以降，図の修正が行われたコミットまでにソースコードは数十行しか更新されていない．ソースコード全体の行数は数万行であるため，更新順によって優先的に選択することができる．このように，最近追加されたある箇所が欠陥の原因であり，その箇所を移動することで欠陥の修正が実現できるような場合，更新順による選択は適している．

その他にも更新順による選択が有用である例として，2 箇所ある同様の欠陥に対してどちらか一方のみ欠陥の修正が行われた場合などが挙げられる．この場合，一方の修正直後であれば，更新順を用いるともう一方の修正に必要な行は優先的に選択されるため，有用である．

4　調査

本章では調査の内容を述べる．本研究では，更新順の有用性の調査を行う．調査は過去に行われた欠陥の修正において用いられた行を対象にして行う．各節において，調査の目的，手順，対象について述べる．

4.1　調査目的

前章において，更新順という基準を提案した．本研究では，再利用に基づく自動プログラム修正手法に更新順による行の選択を実装する前に，更新順が行を選択する基準として有用であるかどうかを調査する．更新順においてプログラムを修正する行が優先的に選択されれば，更新順が有用であるということを示せる．調査の結果，更新順が有用であることが示せれば，今後の研究で再利用に基づく自動プログラム修正手法に更新順を実装する価値が有ることになる．

4.2　調査手順

本研究ではオープンソースソフトウェアで過去に行われた欠陥の修正に対して調査を行う．過去に行われた欠陥の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を解析することにより収集する．調査対象としたソフトウェアについては 4.3 節で述べる．横山らの研究で説明した用語のうち，挿入候補，挿入行，修正コミットを本研究でも用いる．

調査の手順について述べる．まずはじめに，ソースコードリポジトリの全コミットに対してコミットログを調べ，修正コミットを特定する．特定した全ての修正コミットに対して，以下の 4 つの処理を行う．

ステップ 1　修正前後のリビジョンのソースコード取得

ステップ 2　挿入行の抽出

ステップ 3　更新順位表の作成

ステップ 4　挿入行の更新順の順位取得

上記の各ステップを図 3 に示す．それぞれのステップについて説明する．

ステップ 1：修正前後のリビジョンのソースコードの取得

修正コミットで行われた修正の前後のリビジョンからソースコードを取得する．修正前のリビジョンでは，各行の更新日時も取得する．

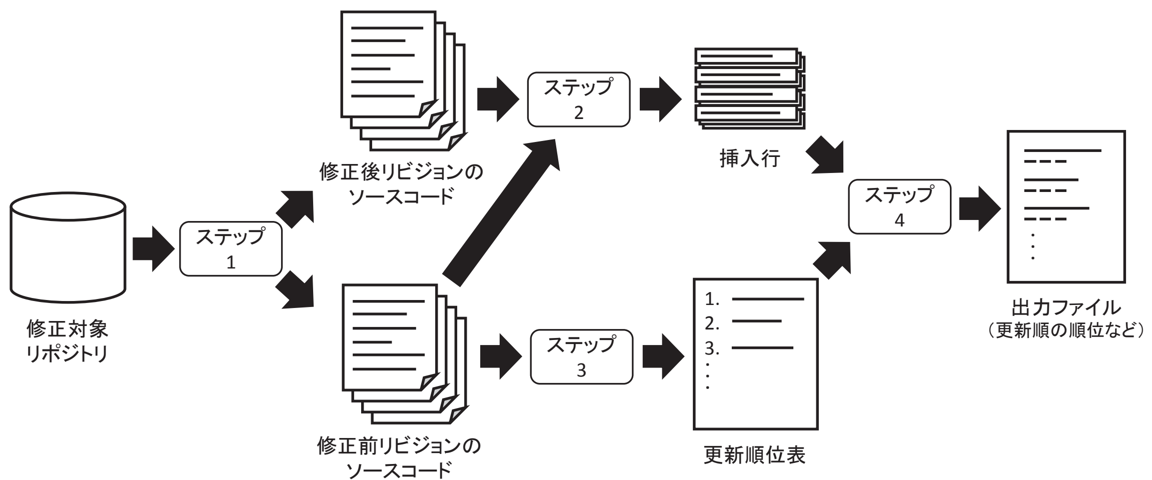


図 3: 修正コミットごとの処理

ステップ **2**：挿入行の抽出

修正前後のリビジョンにおけるソースコードの差分を取り，挿入行を抽出する．差分は行単位の追加・削除で表される．そのうち，追加された行を挿入行とする．

ステップ **3**：更新順位表の作成

修正前のリビジョンにおけるソースコード中の全ての行を更新日時の新しい順で並び替えて表を作成する．このとき，同じ更新日時の行の順番はランダムに決める．作成した表を更新順位表とよぶことにする．

ステップ **4**：挿入行の更新順の順位取得

順位表を基にそれぞれの挿入行の更新順の順位を取得する．この順位は，更新順により挿入候補を選択するとき，挿入行を選択するまでに選択する行の数を表す．得られた順位は評価に用いるためソースコードの総行数とともにファイルに出力する．

以上で 1 つの修正コミットに対する処理を終了する．

全ての修正コミットにおける挿入行の更新順の順位を取得した後，評価を行う．評価のために，カバレッジを定義する．カバレッジは以下の式で算出される．

$$\text{カバレッジ} = \frac{\text{更新順の順位 (=更新順で挿入行を選択するまでに選択する行の数)}}{\text{ソースコードの総行数}}$$

カバレッジは 0 より大きく 1 以下であるような値を取る．値が 0 に近いほど更新順で優先的に挿入

行が選択されることを示し，その挿入行に対しては更新順を用いた挿入候補の選択が適しているといえる．全ての挿入行に対してカバレッジを算出し，評価を行う．

さらに，更新順と横山らの手法 [14] における基準である類似度順の比較を行う．同じ調査対象に対して横山らの手法を適用し，類似度順で同様にして各挿入行のカバレッジを算出する．その後，更新順により算出したカバレッジと比較を行う．

4.3 調査対象

調査対象のソフトウェアを表 2 に示す．調査対象はバージョン管理システム Subversion で管理された 4 つのオープンソースソフトウェアであり，横山らの研究 [14] における調査対象と同じである．本研究ではリビジョンの範囲を初期リビジョンから 2015/12/31 までのリビジョンと定め，全ての挿入行を調査する．なお，JabRef に関しては現在 Git のみのバージョン管理となっているため，Subversion でバージョン管理されていたリビジョンまでを範囲とする．Apache httpd の初期リビジョンが r1 でないのは，同じソースコードリポジトリで複数のソフトウェアが管理されており，途中から Apache httpd が追加されたためである．

調査対象のプログラムは C 系の言語と Java からそれぞれ 2 つずつ採用している．これにより，結果が言語に依存するかどうかを見ることができる．

表 2: 本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（日付）
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)

5　結果と考察

本章では，調査の結果を図にまとめた後，考察を行う．5.1 節において，挿入行から算出した更新順のカバレッジを分布にして図にまとめた後，更新順の有用性の評価を行う．5.2 節において，挿入行の更新順と類似度順のカバレッジを散布図にまとめた後，更新順と類似度順の比較を行う．

5.1　更新順の有用性の評価

はじめに，各ソフトウェアにおける挿入行の数を表 3 にまとめる．挿入行の数は 4.2 節の調査手順で述べた挿入行の抽出において抽出した行の数である．カバレッジを算出した挿入行の数は挿入行のうち修正前のリビジョンのソースコード中に挿入行が存在するものの数である．

表 3 の各挿入行で算出されたカバレッジを分布にしてまとめたものが図 4 である．図が複雑になるのを防ぐためにカバレッジが 0.5 以上の範囲は 1 つにまとめている．

図 4 からそれぞれの調査対象のソフトウェアにおいて挿入候補の 10 分の 1（カバレッジが 0.1 以下）に約 60%の挿入行が集中しているということがわかる．JabRef においては 70%近くの挿入行が集中している．また，挿入候補の半分を見ることで約 90%の挿入行を選択することができる．つまり，更新順で選択した場合，挿入候補の半分以上を選択する必要がある挿入行は 10 行中 1 行しかないということである．

調査対象のソフトウェアは C 系の言語で記述されたソフトウェアと Java で記述されたソフトウェアからそれぞれ 2 つずつ選択した．Apache httpd，CBMC は C 系の言語で記述されたソフトウェア，JabRef，jEdit は Java で記述されたソフトウェアである．挿入行のカバレッジが値の小さい範囲に集中する割合でソフトウェアの大小を表すと，JabRef，CBMC，Apache httpd，jEdit の順番になる．一方の言語において割合が極端に集中するといった偏りがないため，本研究の調査対象においては，更新順が言語に大きくは依存していないと判断できる．

表 3: 各ソフトウェアにおける挿入行の数

ソフトウェア	挿入行の数	カバレッジを算出した挿入行の数
Apache httpd	26,679	7,807
CBMC	4,890	1,642
JabRef	12,260	4,116
jEdit	21,161	7,672

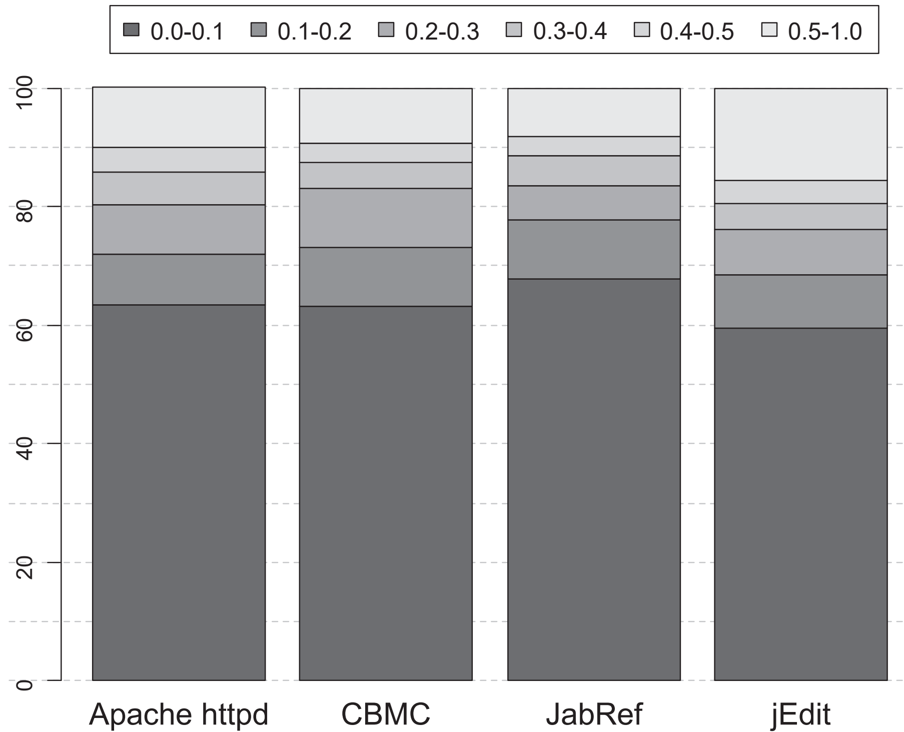


図 4: 更新順のカバレッジの分布

5.2 更新順と類似度順の比較

図 5 から図 12 より，更新順と類似度順の関係を 4 つの対象ソフトウェアそれぞれについて 3 つの図を用いて調査の結果を記載する．

上部に存在する図は更新順と類似度順のそれぞれのカバレッジを挿入行に対してプロットした散布図である．タイトルの右に記述された n は更新順と類似度順のカバレッジを比較した挿入行の数である．縦軸方向に更新順（Freshness）のカバレッジ，横軸方向に類似度順（Similarity）のカバレッジを取る．プロットされた点が下にあるほどその挿入行が更新順において優先的に選択されることを示す．プロットされた点が左にあるほどその挿入行が類似度順において優先的に選択されることを示す．

各頁において下部に存在する 2 つの図は更新順と類似度順を組み合わせることでどれくらいの挿入行が選択できるかを表した図であり，散布図を基に作成している．左はそれぞれの基準のカバレッジが 0.1 以下である挿入行の集合によるベン図である．右はそれぞれの基準のカバレッジが 0.2 以下である挿入行の集合によるベン図である．*Freshness0.1*，*Freshness0.2* はそれぞれ更新順のカバレッジが 0.1 以下，0.2 以下である挿入行の集合を表す．*Similarity0.1*，*Similarity0.2* はそれぞれ類似度順ののカバレッジが 0.1 以下，0.2 以下である挿入行の集合を表す．

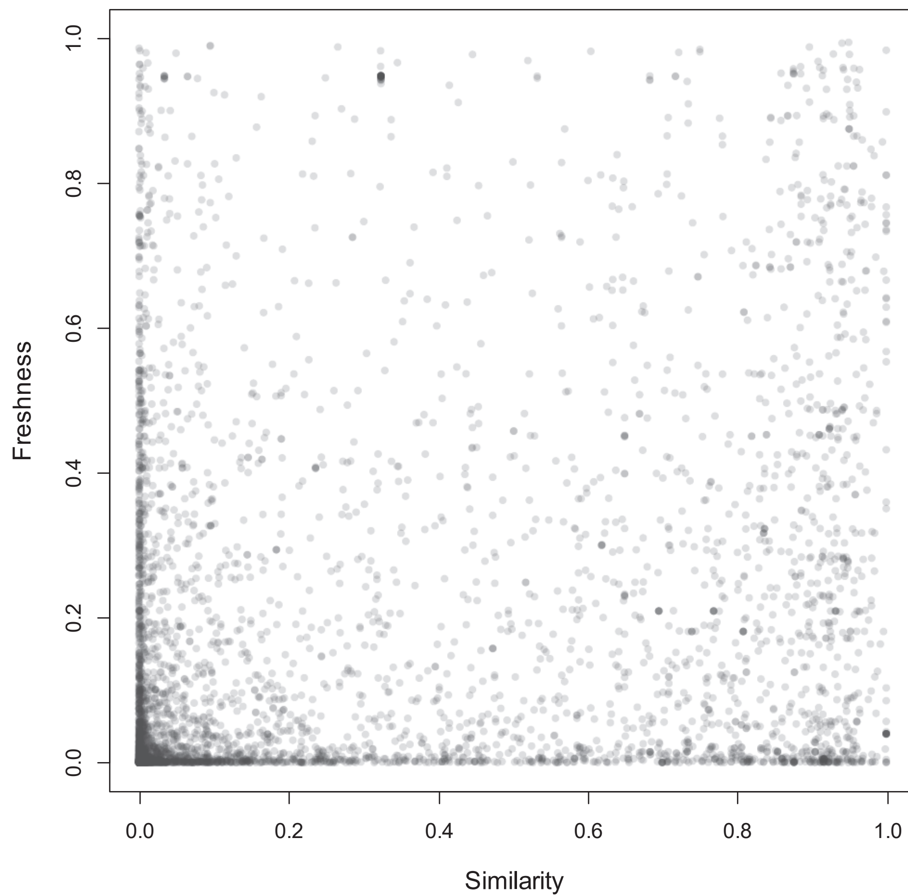


図 5: 更新順と類似度順の関係（Apache httpd）

(a) カバレッジ 0.1 以下

(b) カバレッジ 0.2 以下

図 6: 各基準でカバレッジの小さい挿入行の集合によるベン図（Apache httpd）

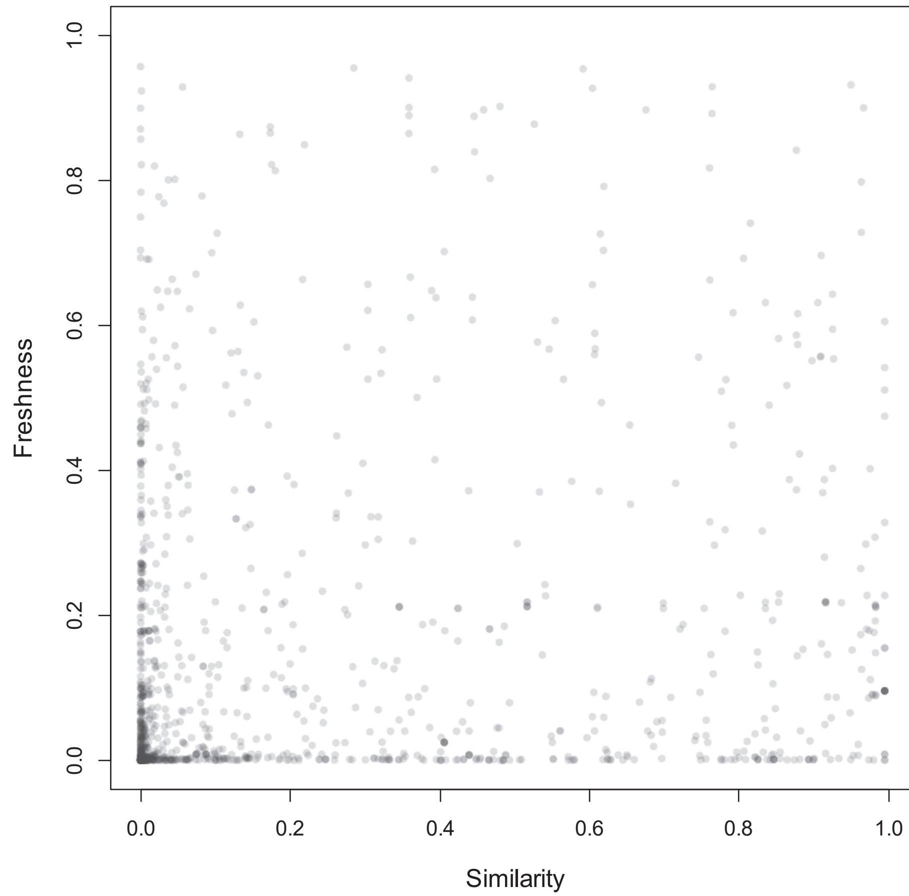


図 7: 更新順と類似度順の関係（CBMC）

(a) カバレッジ 0.1 以下

(b) カバレッジ 0.2 以下

図 8: 各基準でカバレッジの小さい挿入行の集合によるベン図（CBMC）

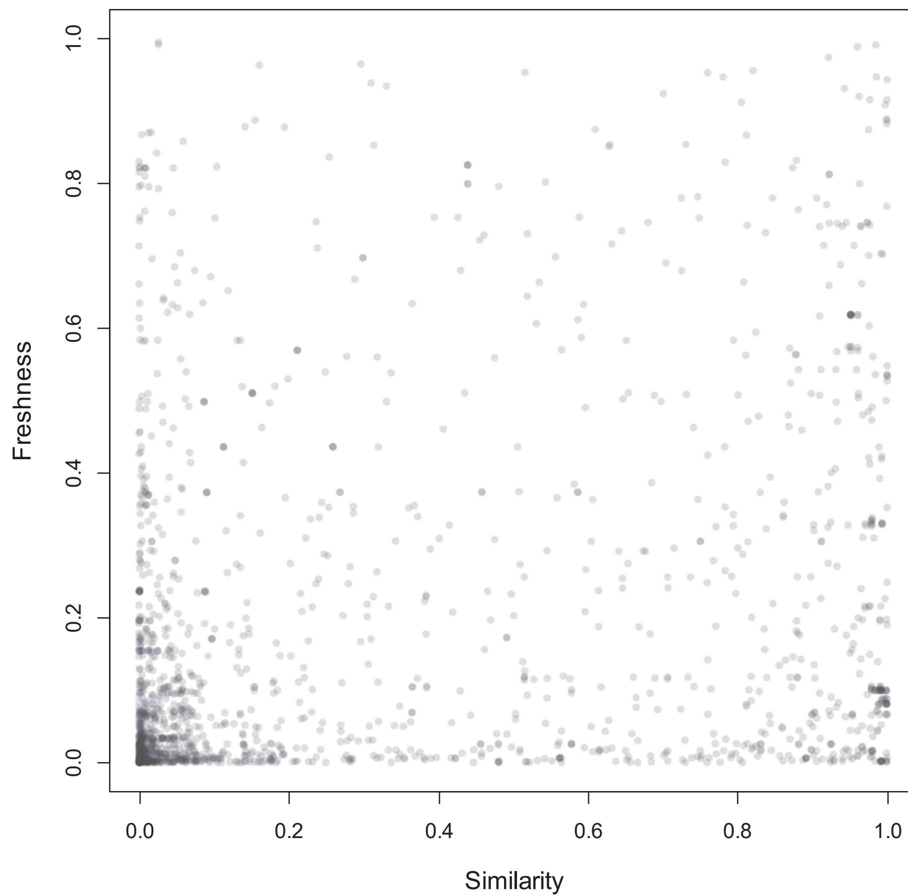
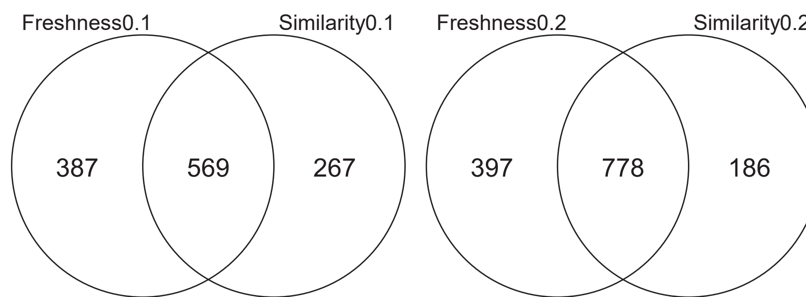


図 9: 更新順と類似度順の関係（JabRef）



(a) カバレッジ 0.1 以下

(b) カバレッジ 0.2 以下

図 10: 各基準でカバレッジの小さい挿入行の集合によるベン図（JabRef）

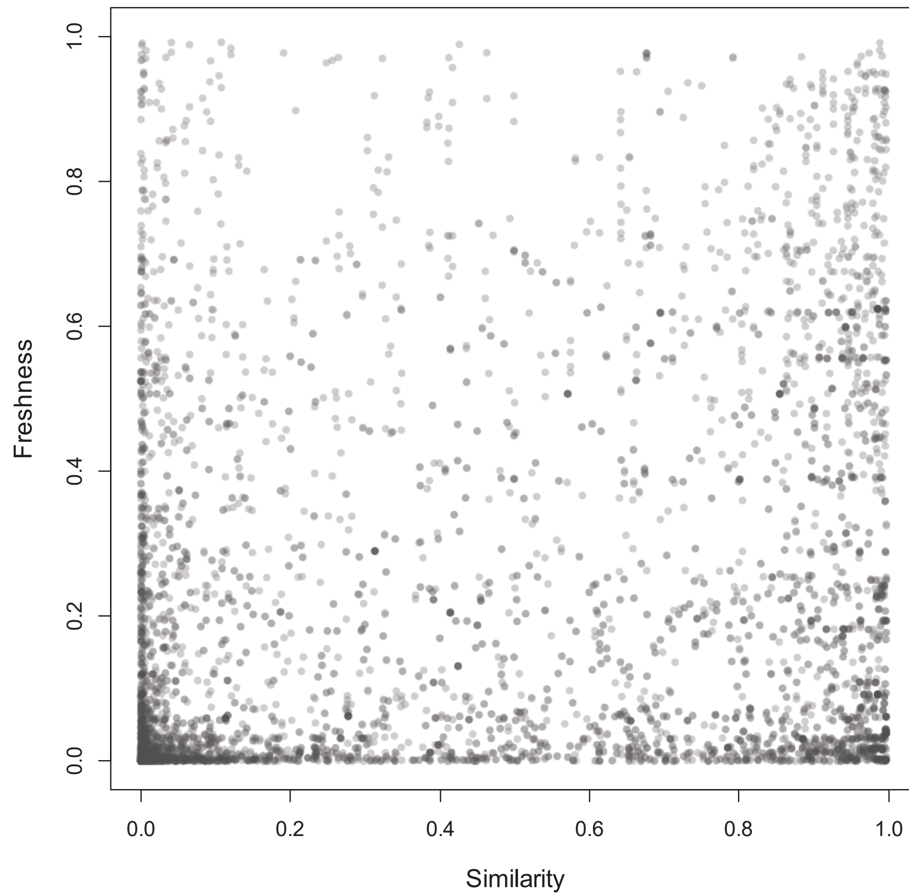


図 11: 更新順と類似度順の関係（jEdit）

(a) カバレッジ 0.1 以下

(b) カバレッジ 0.2 以下

図 12: 各基準でカバレッジの小さい挿入行の集合によるベン図（jEdit）

更新順と類似度順の関係性について考察する．各ソフトウェアの散布図において軸付近に多くの点がプロットされていることが分かる．Similarity 軸付近の点は更新順で優先的に選択される挿入行を表す．Freshness 軸付近の点は類似度順で優先的に選択される挿入行を表す．軸付近の点である挿入行の数はそれぞれのソフトウェアのベン図から読み取ることができる．カバレッジが 0.1 以下の挿入行の集合によるベン図に注目する．どのソフトウェアにおいても，更新順でカバレッジが 0.1 以下であり，類似度順でカバレッジが 0.1 より大きい挿入行が多数存在する．カバレッジが 0.2 以下の挿入行の集合のベン図でも同様のことがいえる．よって，更新順では優先的に選択されるが，類似度順では優先的に選択されない挿入行が多数存在することがわかった．

更新順と類似度順を組み合わせて行を選択する基準について考える．各ベン図内の数値を足しあわせると表 4 のようになる．比較した挿入行の数は，更新順と類似度順のカバレッジを比較した挿入行の数である．カバレッジが 0.1 以下の和，0.2 以下の和は，更新順と類似度順のどちらか一方の基準でカバレッジが 0.1 以下，0.2 以下であるような挿入行の数である．更新順と類似度順のどちらか一方の基準でカバレッジが 0.1 以下であるような挿入行は全体の約 70%，0.2 以下であるような挿入行は全体の約 80%存在する．ベン図より，片方の基準でのみカバレッジが小さい挿入行が多数存在するので，互いの基準が相手の基準の欠点を補っていることが分かる．よって，更新順と類似度順を組み合わせて行を選択する基準を再利用に基づく自動プログラム修正手法に実装することで，欠陥の修正にかかる時間を短縮できる可能性がある．このとき，更新順と類似度順を組み合わせる基準による手法は，自動プログラム修正の最初にそれぞれの基準によって挿入候補を順位付けする処理を行う必要があり，それぞれの基準を単独で使用するより時間を要してしまうというデメリットがある．基準を組み合わせて再利用に基づく自動プログラム修正手法に実装をするときは，それぞれの順位付けの処理を並列プログラミングを用いて行うなど，デメリットを考慮した実装が必要になる．

表 4: 更新順と類似度順の組み合わせによる挿入行の割合

プログラム	比較した挿入行の数	カバレッジ 0.1 以下の和	カバレッジ 0.2 以下の和
Apache httpd	4,729	3,528 (74.6%)	3,878 (81.8%)
CBMC	1,209	967 (80.0%)	1,055 (87.2%)
JabRef	1,675	1,223 (73.0%)	1,361 (81.3%)
jEdit	5,996	4,049 (67.5%)	4,582 (76.4%)

6　妥当性への脅威

この章では本研究の調査における妥当性への脅威について述べる．

本研究では，欠陥の修正が行われたコミットを表す修正コミットを，コミットログに特定の修正に関するキーワードを含むコミット，と定めている．そのため，キーワードを含まないが欠陥の修正を行ったコミットを対象にすることができていない．また，キーワードを含むが欠陥の修正を行っていないコミットを対象にしてしまっている可能性が考えられる．

本研究では，欠陥の修正に用いられた行を表す挿入行を修正コミットにおいて更新された全ての行としている．しかし，欠陥の修正の際には修正とは無関係の行を同時にコミットすることが考えられる．そのような場合，挿入行に修正とは無関係な行が含まれる可能性がある．

本研究では，実験対象として 4 つのソフトウェアのソースコードリポジトリを調査した．実験対象のソースコードリポジトリを変更すると調査の結果が変わる可能性がある．プログラムは C 系の言語から 2 つ，Java から 2 つ選択して対象としている．それ以外の言語のプログラムでは調査を行っていない．そのような言語を対象とした自動プログラム修正手法に更新順を実装する際には再度調査を行うべきである．

7　あとがき

本研究では，再利用に基づく自動プログラム修正における挿入候補の選択手法として，更新順という基準を用いることを提案した．更新順の有用性を調べるために，オープンソースソフトウェアで過去に行われた欠陥の修正において挿入された行を用いて調査を行った．調査の結果，対象の行のうち約 6 割の行が更新順で挿入候補の 10%の行を選択するまでに選択されていることがわかった．さらに，既存研究の基準である類似度順による手法を同様の対象に適用することで，類似度順と更新順の比較を行った．その結果，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わかった．

今後の課題は，挿入候補から行を選択する基準として，更新順単独または更新順と類似度順を組み合わせた基準を再利用に基づく自動プログラム修正手法に実装して，自動プログラム修正に要する時間を短縮できるかどうか調べることである．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，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て，研究に対する考え方や方向性など，丁寧なご指導かつ的確なご助言を頂きました肥後芳樹准教授に心より感謝申し上げます.

本研究を行うにあたり，議論の中で貴重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.

本研究の全過程を通して，日常の中で声をかけて頂き，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 年の横山晴樹氏に深く感謝申し上げます.

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. *University of Cambridge-Judge Business School, Tech. Rep*, 2013.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, pp. 215–222, 1976.

[3] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07*, pp. 75–84, 2007.

[4] Carlos Pacheco. *Directed Random Testing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, 2009.

[5] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07*, pp. 416–426, 2007.

[6] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pp. 467–477, 2002.

[7] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *ICSE '09*, pp. 45–55, 2009.

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

[9] 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 *ICSE '12*, pp. 3–13, 2012.

[10] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *ICSE '14*, pp. 254–265, 2014.

[11] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *ICSE '13*, pp. 772–781, 2013.

[12] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE '13*, pp. 802–811, 2013.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS'08/ETAPS'08*, pp. 337–340, 2008.

[14] 横山晴樹, 大田崇史, 堀田圭佑, 肥後芳樹, 岡野浩三, 楠本真二. 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. 電子情報通信学会技術研究報告, 第 115 巻, pp. 047–052, 2015.

[15] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Vol. 1. MIT press, 1992.

[16] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pp. 364–374, 2009.

[17] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954, 2009.

[18] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *FSE '14*, pp. 306–317, 2014.

[19] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, Vol. 27, pp. 929–948, 2001.

[20] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, Vol. 21, pp. 168–173, 1974.