ソースコードの変更予測手法を用いた自動プログラム修正の高速化

鷲見 創一[1]　肥後 芳樹[1]　楠本 真二[1]

概要：近年，既存プログラム文の再利用による自動プログラム修正手法が注目されている．再利用に基づく自動プログラム手法では，修正対象プログラムからプログラム文を取得して，欠陥であると特定された箇所へそのプログラム文を挿入する．修正対象プログラム中にはプログラム文が大量に存在しており，既存手法ではランダムにそれらの中からプログラム文を取得するため，修正に長い時間を要する．一方で，プログラムの構文情報と変更履歴を入力として，次の変更後にプログラムが持つ構文情報を出力するソースコードの変更予測手法が提案されている．そのようなプログラム文を修正に用いることにより，修正に要する時間を大きく削減できると筆者らは考えた．そこで本研究では，ソースコードの変更予測を用いて自動プログラム修正手法を高速化する手法を提案し，オープンソースソフトウェアの開発過程で発生した欠陥に対して提案手法を適用した結果を報告する．評価実験の結果，63 回の欠陥修正のうち 29 回において提案手法がより高速に修正を行った．また既存手法よりも 5 つ多くの欠陥を修正した．平均修正時間で比較すると約 18.3%の削減であった．

1. まえがき

デバッグはソフトウェアの信頼性の向上のために避けることのできない作業である．ソフトウェア開発においてデバッグは多くの労力を必要とする作業であり，開発工数の半数以上を占めると言われている [1]．そのためデバッグにかかる労力の削減は有益である．

これまでに提案されている自動プログラム修正手法の 1 つに，修正対象プログラムのプログラム文を用いて欠陥箇所を変更したプログラム (以降，変異プログラムと呼ぶ) の生成，評価を繰り返し行うことによりプログラムを修正する手法がある．この手法の 1 つとして，Weimer らが開発した GenProg がある [2]．GenProg は変異プログラムを複数生成し，遺伝的プログラミングに基づいて変異プログラムの評価，選択を繰り返すことにより欠陥の修正を行う．欠陥箇所の変更には修正対象プログラムに存在するプログラム文を用いる．また，プログラムの修正が完了したかどうかは，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判定する．

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 個の欠陥の修正に成功することによってその有効性を示した [2]．しかし，修正対象プログラム中に存在しないプログラム文が必要な欠陥は修正できないことや，修正に要する時間は修正成功の場合は平均 1 時間 36 分，修正失敗の場合は平均 11 時間 12 分と，修正に時間を要する事が課題である．

GenProg は変異プログラムを生成する際に修正対象プログラムからプログラム文をランダムに取得して，欠陥箇所の変更に用いる．修正対象プログラムにはプログラム文が大量に存在するため，効率的に修正を行えない場合が多い．

一方で，プログラムの構文情報と変更履歴を入力として，次の変更後にプログラムが持つ構文情報を出力するソースコードの変更予測手法が提案されている [9]．この手法によって得られる構文情報の予測結果と，予測対象の構文情報を比較することによって，次にどのような構文情報を持つプログラム文が追加される可能性が高いか予測できる．追加される可能性が高いプログラム文を変異プログラムの生成に用いることにより，修正に要する時間を大きく削減できると筆者らは考えた．

そこで本研究では，ソースコードの変更予測を用いて自動プログラム修正手法を高速化する手法を提案する．そして OSS の開発過程で発生した欠陥に対して提案手法を適用した結果を報告する．

2. 関連研究

既存の自動プログラム修正手法は以下の 3 種類に分けら

[1] 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
〒 565-0871 大阪府吹田市山田丘 1-5

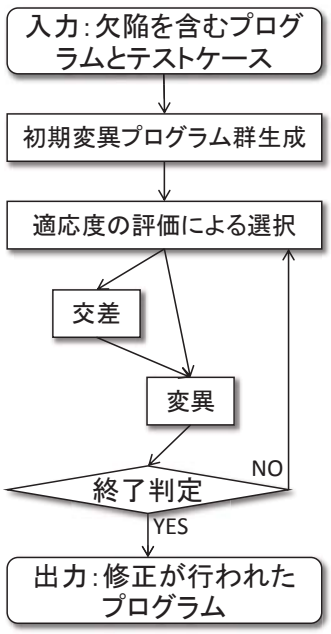


図 1: GenProg の動作の流れ

れる．

- 再利用に基づく手法 [2][3]
- プログラム意味論に基づく手法 [6][11]
- 修正パターンに基づく手法 [7][10][12]

これらの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．修正が完了したかどうかの判断には修正対象プログラムの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．

Weimer らは，再利用に基づく手法の 1 つである GenProg を提案した [2]．GenProg は修正対象プログラムのプログラム文を用いて欠陥の箇所を変更したプログラム (以降，変異プログラムと呼ぶ) の生成，評価，選択を遺伝的プログラミングに基づいて行う．プログラムの変更はプログラム文単位で行う．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．

挿入 欠陥箇所の前または後ろにプログラム文の挿入を行う処理

削除 欠陥箇所を削除する処理

置換 欠陥箇所の削除と挿入を同時に行う処理

GenProg は OSS の開発過程で発生した欠陥に対して適用され，105 個中 55 個の欠陥を修正することにより，その有用性を示した [2]．しかし GenProg は，変異プログラムを評価する際に全てのテストケースを実行するため，計算コストが高い．そこで，Qi らは実行するテストケースに優先順位付けを行い，失敗したテストが現れた時点でテストの実行を打ち切ることにより高速にプログラムの修正を行う RSRepair を提案した [3]．Qi らは RSRepair を 8 つの OSS に対して適用し，GenProg よりも多くの欠陥を短い時間で修正に成功したことを報告している．しかし，RSRepair は複数箇所の変更を必要とする欠陥を修正できない．

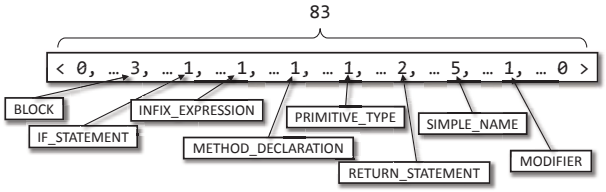
プログラム意味論に基づく手法である SemFix では，テストスイートを用いて欠陥箇所を特定し，欠陥箇所に関わると特定された箇所が満たすべき制約を導出し，制約を満たすプログラム文を生成する [4]．Nguyen らはこの手法を 5 つのソフトウェアに対して適用し，GenProg よりも多くの欠陥の修正に成功したことを報告している．再利用に基づく手法は既存ソースコードに存在しない記述による修正を行えないことに対し，プログラム意味論に基づく手法は既存ソースコードに存在しない記述を用いた修正が可能であることが特徴である．この手法はテストスイートから欠陥の箇所が満たすべき論理式を導出し，その論理式を SMT ソルバを用いて解く．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，生成されたコードが人が書いたコードとはかけ離れた非常に読みづらいコードになることも課題である．

そこで SemFix を改良した DirectFix が提案された [6]．DirectFix は，SemFix よりも修正に時間を要するが，読みやすさの点で SemFix よりも改善された修正を生成できると報告されている．Angelix は DirectFix を改良した手法である [11]．Angelix は欠陥箇所が満たすべき制約を導出する際に記号化する文を絞り込むことにより，DirectFix に比べて短時間で欠陥を修正できたと報告されている．

修正パターンに基づく手法である PAR は Null チェックやオブジェクトの初期化など 10 個の修正パターンを定義しており，それらに基づいて修正を行う [7]．Kim らは 6 つの OSS への適用によって GenProg よりも多くの欠陥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と報告している．SPR も修正パターンに基づく手法の 1 つである [12]．SPR は条件式の変更や値の変更，制御フロー文の導入など，6 つの修正パターンを定義してプログラムの修正を行う．PAR と比べて抽象的な修正パターンが定義されており，より多くの欠陥を修正可能なことや，条件式や値の探索を枝刈りによって効率的に行うことが特徴である．Long らは SPR を 8 つの OSS に対して適用し，開発者が行った修正と等価な修正を GenProg よりも多く行えたと報告している．Prophet はより高い確率で開発者が行った修正と等価な修正 (以下，正しい修正と呼ぶ) が可能となるよう，SPR を改良した手法である [10]．Prophet はまず OSS の変更履歴から欠陥修正コミットを抽出する．そして，修正パターンと修正箇所の周囲の構文情報から SPR によって生成された修正パッチが正しい修正かどうか判定する確率モデルを構築する．最後に構築したモデルを用いて生成されたパッチを並べ替え，もっとも正しい修正である可能性が高い順に出力する．Long らは OSS の開発過程で発生した 69 個の欠陥に対して Prophet を適用し，18 個

```
public int max(int x,int y){
    if(x>=y){
        return x;
    }else{
        return y;
    }
}
```

(a) ソースコード



(b) 状態ベクトル

図 2: 状態ベクトルの例

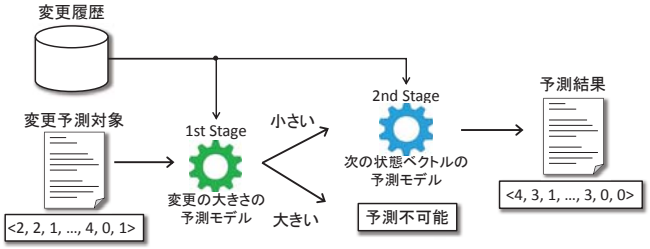


図 3: ソースコードの変更予測手法の概要

の欠陥に対して正しい修正を行う事ができたと報告している．しかし，定義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，複数箇所の変更が必要な欠陥は修正できないことが課題である．

そこで本研究は，再利用に基づく手法を対象とした自動プログラム修正の高速化手法を提案する．この手法は複数箇所の変更を必要とする欠陥を修正可能であり，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，より多くの欠陥を修正可能である．そのため修正に長い時間を要するという課題を解決できればより有用な手法となる．本論文では，まず提案手法について説明し，OSS の開発過程で発生した欠陥に対して提案手法を適用した結果を報告する．

3. ソースコードの変更予測手法

本章では，本論文で使用する用語と提案手法で用いるソースコードの変更予測手法 [9] について述べる．

3.1 状態ベクトル

状態ベクトルとは，ソースコード中に存在するプログラム要素の数のベクトルである．ここで，プログラム要素には if 文や return 文，変数宣言，識別子名などが含まれる．ソースコード中の各プログラム要素の出現回数をカウントしたものが状態ベクトルとなる．

図 2 に状態ベクトルの例を示す．図 2(a) のメソッドを状態ベクトルにしたものが図 2(b) である．図 2(a) のメソッドには，修飾子，基本型名，メソッド宣言，識別子名，return 文，if 文，２項演算子，ブロックの 8 種類のノードが出現している．状態ベクトルは 83 個の要素を持つため，それらの内の 8 つの要素は 1 以上の値を持ち，残りの 75 要素が 0 となる．

3.2 ソースコードの変更予測

本研究で用いるソースコードの変更予測手法について説明する．入力は，対象プログラムにおいて変更を予測したいソースファイルの状態ベクトルと，その対象プログラムの変更履歴である．出力は，修正が加えられた後の状態ベクトルである．変更前後の状態ベクトルの差分を取ることによって，次の変更で追加および削除されるであろうプログラム要素の状態ベクトルを取得可能である．

図 3 にソースコードの変更予測手法の処理の流れを示す．まずはじめに対象プログラムの変更履歴を用いて予測モデルを構築する．構築する予測モデルは 2 種類である．1 つ目が次の変更が大きいか小さいかを予測するモデル，2 つ目が次の変更でどのプログラム要素が追加または削除されるかを予測するモデルである．

2 つのモデルの構築が完了したら，変更を予測したいソースファイルの状態ベクトルを 1 つ目のモデルに与える．1 つ目のモデルが，次の変更は大きいと予測した場合，この変更予測手法は処理を中断する．1 つ目のモデルが，次の変更は小さいと予測した場合，2 つ目のモデルに状態ベクトルを与え，変更後のソースファイルの状態ベクトルを取得する．

1 つ目のモデルには，変更履歴から抽出されたソースファイルの状態ベクトルとそのソースファイルに加えられた変更の大小の情報が格納されている．予測アルゴリズムとして k 近傍法を利用した．また，変更の大きさは，2 つの状態ベクトル（変更前後のソースファイルの状態ベクトル）のマンハッタン距離を用いた．つまり，予測したいソースファイルの状態ベクトルが与えられた場合，その状態ベクトルと最も近い（マンハッタン距離が小さい）状態ベクトルを取得して，その状態ベクトルのラベルが大小のいずれかを確認する．小であれば，与えられたソースファイルに次に加えられる変更は小さいとの予測になるので，2 つ目のモデルを利用した変更の予測に進む．

2 つ目のモデルでは，変更履歴から抽出されたソースファイルの状態ベクトルとそれらに対する変更の関係を重回帰分析により表現している．つまり，状態ベクトルの各変数（各要素）の変更の予測が，重回帰式を用いて行われる．予測したいソースファイルの状態ベクトルを構成する各要素の値を用いて，全ての要素の重回帰式を計算することによ

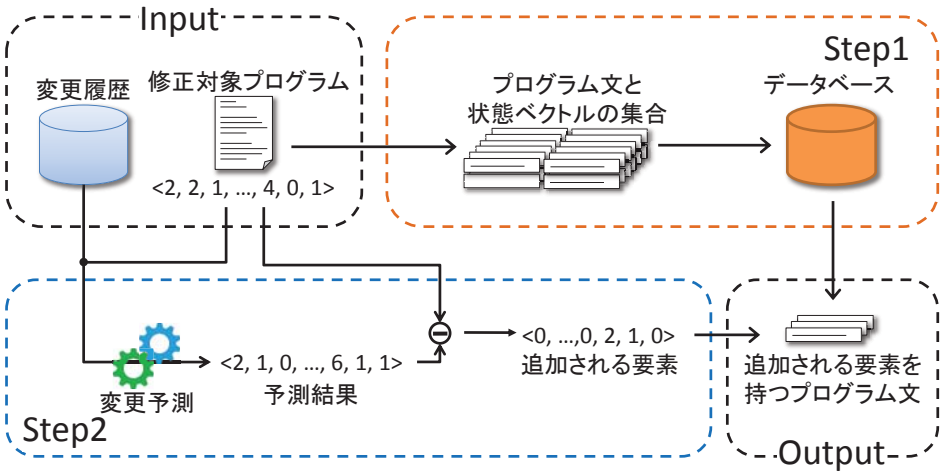


図 4: 提案手法の概要

り，変更後のソースファイルの状態ベクトルを取得する．

ソースコードの変更予測手法は 2 つの OSS に対して適用され，変更の大きさの閾値が 5 の場合に約 74–85%の精度[*1]で変更後の状態ベクトルの全ての要素の予測に成功したと報告されている [9][*2]．

4. 提案手法

本研究では，自動プログラム修正を高速化する手法を提案する．提案手法の概要を図 4 に示す．提案手法は修正対象プログラムと修正対象プロジェクトの変更履歴を入力として，追加されるプログラム要素を持つプログラム文を出力する．まず，ソースコードの変更予測手法を用いて，修正対象プログラムが次の変更でどのようなプログラム要素が追加されるかを予測する．そして，追加されるプログラム要素を持つプログラム文を修正対象プログラム中から取得する．提案手法は，変異プログラムの作成に挿入もしくは置換の操作を行う場合に利用される．削除の場合には提案手法を利用されない．

既存の自動プログラム修正手法は挿入候補からプログラム文をランダムに選択して修正に用いる．提案手法によって取得されたプログラム文を用いることで，全てのテストを通過するまでに生成される変異プログラムの数が減少する．そのためテストケースを実行する回数が減り，自動プログラム修正を高速化できると筆者らは考えた．

提案手法は以下の 2 つの STEP に分かれている．

- STEP1: データベースの構築
- STEP2: プログラム文の推薦

STEP1 では修正対象プログラムからプログラム文とその状態ベクトルを抽出し，データベースに格納する．STEP2 では，修正対象プログラムと修正対象プロジェクトの変更履歴を入力として，追加されるプログラム要素を持つプログラム文の特定を行う．以降では各 STEP で行う処理について述べた後，実装について述べる．

STEP1: データベースの構築

STEP1 では，修正対象プログラムから抽象構文木を構築し，プログラム文と状態ベクトルを対応付けてデータベースに格納する．まず，修正対象プログラムから抽象構文木を構築する．そして抽象構文木のノードの内，プログラム文のノードを根とする全ての部分木を辿り，全てのプログラム文とその状態ベクトルを得る．最後に得られた状態ベクトルをキー，プログラム文をバリューとしてデータベースに格納する．

STEP2: プログラム文の推薦

STEP2 では，修正対象プログラムの状態ベクトルと修正対象プロジェクトの変更履歴を入力として，プログラム修正のために追加されるプログラム要素を持つプログラム文の推薦を行う．

まず，ソースコードの変更予測手法を用いてプログラムの状態ベクトルと変更履歴から，状態ベクトルが次の変更でどのような状態ベクトルになるかを予測する．その後，修正対象プログラムの次の状態ベクトルと元の状態ベクトルとの差分をとることにより，次の変更で追加されるプログラム要素の状態ベクトルを得る．最後に，追加されるプログラム要素の状態ベクトルをキーとして STEP1 で構築したデータベースに対して問い合わせを行うことにより，次の変更で追加されるプログラム文の集合を取得可能である．

5. 実装

本節では提案手法の詳細な実装方法について述べる．実

[*1] 双方のモデルが正しく予測できた精度．1 つ目のモデルの精度と 2 つ目のモデルの精度の積である．

[*2] ソースコードの変更予測手法を提案している文献 [9] では，変更の大きさはユークリッド距離を用いて表現されているが，本研究ではユークリッド距離ではなくマンハッタン距離を用いて変更の大きさを評価している．

験ツールは，GenProg の Java 実装である jGenProg[16] の挿入候補選択部分を変更して実装した．

5.1 状態ベクトルの取得方法

状態ベクトルはソースコードから抽象構文木を構築して根から葉まで辿り，各プログラム要素の出現回数をカウントしたものである．抽象構文木の構築には JDT(Java Development Tools) を用いた．JDT は抽象構文木の構築や操作が可能であり，各プログラム要素をあらわす 83 種類のノードが定義されている．そのため状態ベクトルの次元は 83 となる．また提案手法の STEP1 では，修正対象プログラムに存在する全てのプログラム文を取得する必要がある．これは抽象構文木のノードのうち，JDT の Statement クラスのサブクラスであるノードを根とする全ての部分木を辿る事により取得した．

5.2 欠陥修正コミットの特定

提案手法の STEP2 では，ソースコードの変更予測手法を用いて次の変更でプログラムの状態ベクトルがどうなるかを予測する．提案手法では，ソースコードの変更予測手法を欠陥の修正に用いるため，変更履歴から欠陥修正に関する変更のみを抽出して変更予測に用いる．これにより，次の欠陥の修正でどのような状態ベクトルになるかを予測可能であると筆者らは考える．欠陥の修正の特定はコミットコメントが “bugfix”, “fix” などの欠陥の修正を表すキーワードを含んでいるかどうかによって判断した．

6. 評価実験

本実験の目的は，ソースコードの変更予測手法を用いて，自動プログラム修正を高速化できるかどうかを確かめることである．そのため本実験では，実際の OSS の開発過程で発生した 106 個の欠陥に対して GenProg と提案手法を適用する[*3]．修正に用いるプログラム文をランダムに選択した場合と提案手法によって推薦されたプログラム文を用いた場合で修正時間の比較を行う．また，GenProg は修正に用いるプログラム文だけでなく，修正箇所も欠陥限局された箇所の中からランダムに選択するため，修正時間のばらつきが大きい．同じ実験対象であっても実行のたびに修正時間やパッチが生成されるかどうかが異なるため，実験は各実験対象について 3 回行った．

実験は 2.40GHz Intel Xeon CPU(2 プロセッサ，計 16 コア)，メモリサイズ 128GB の計算機に Docker コンテナを 7 個同時に立ち上げて行った．各コンテナには CPU コアを 2 つ，メモリを 18GB 割り当てた．タイムアウトは 3 時間とした．また，実験に用いるデータは全て SSD 上に配置した．

GenProg など，全てのテストケースを通過したかどうかで修正が成功したかを判断する自動プログラム修正手法は，開発者の意図とは異なる修正パッチを出力する事があると知られている [13]．そのため GenProg，提案手法によって行われた修正が正しい修正であったかどうかについても調査する．実験対象が多いため，正しい修正であったかどうかの調査は 1 回目の実行で出力されたパッチに対してのみ行う．修正が正しいかどうかは GenProg や提案手法が開発者が行った修正と同じ修正かどうかにより判定する．正しい修正に関する情報は Defects4J[8] より修正前のリビジョンと修正後のリビジョンを取得し，差分を取ることにより得た．各修正パッチが正しいかどうかの判定は目視で行った．

6.1 実験対象

実験には Defects4J[8] を用いる．Defects4J とは，Java で記述された 5 つの OSS(jFreechart, Closure compiler, Apache Commons-Lang, Apache Commons-Math, Joda-time) の開発過程で発生した 357 個の欠陥を収集したものである．本実験では Defects4J で収集された欠陥の内，Apache Commons-Math の 106 個の欠陥を実験対象とする．Defects4J の詳細を表 1 に示す．Defects4J に収集されている欠陥は次の特徴を持つ．

- 課題管理システムで課題と関連付けられており，コミットメッセージで修正されたと述べられている．
- 修正が 1 つのコミットで完結している．
- Java のソースコードの変更により修正されている (設定ファイルやテストファイルに関する欠陥は含まない)．
- 修正前には失敗テストが存在し，修正後には全てのテストを通過する．

6.2 実験条件の統一

GenProg は修正箇所や挿入候補の選択などにランダム値を用いているため，同じ欠陥の修正であっても実行時に乱数生成器に与えるシード値によって修正時間が大きく異なる．また同じシード値を乱数生成器に与えたとしても提案手法は挿入候補の選択にランダム値を用いないため，変異プログラムを同じ数だけ生成したとしても，GenProg と提案手法で乱数を取得する回数は異なる．そのため各変異

表 1: Defects4J の詳細

プログラム	欠陥数	総行数 [LOC]	テスト数
JFreeChart	26	96,000	2,205
Closure Compiler	133	90,000	7,927
Commons Math	106	85,000	3,602
Joda-Time	27	28,000	4,130
Commons Lang	65	22,000	2,245

[*3] 2 章で紹介した GenProg の高速化手法である RSRepair は，複数行の変更に対応していないため，本実験では比較対象とはしていない．

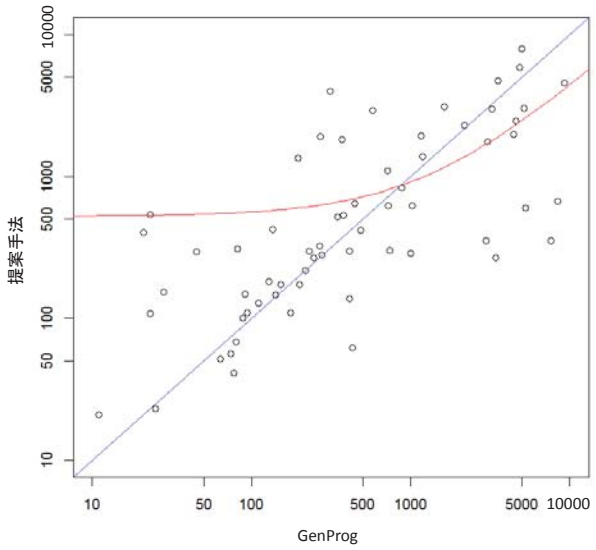


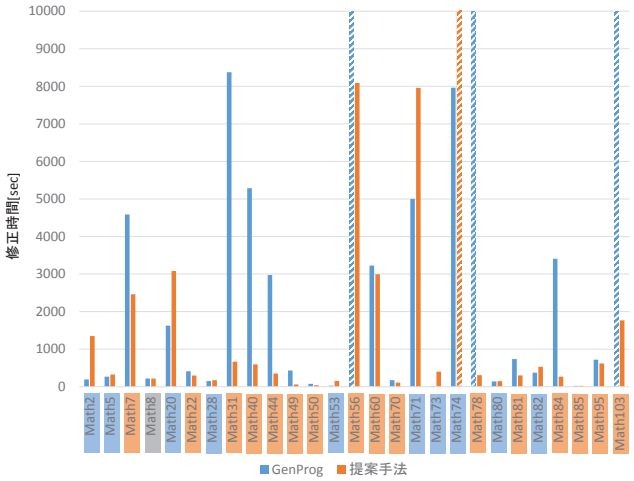
図 5: 欠陥修正に要した時間の散布図

プログラムの修正箇所や適用する操作は異なってしまう．GenProg と提案手法をより厳密に比較するため，本実験では乱数生成器を修正箇所用とその他用に分けることにより，同じシード値を乱数生成器に与えて変異プログラムを同じ数だけ生成した場合に，同じ箇所に同じ操作が加わるよう実装した．

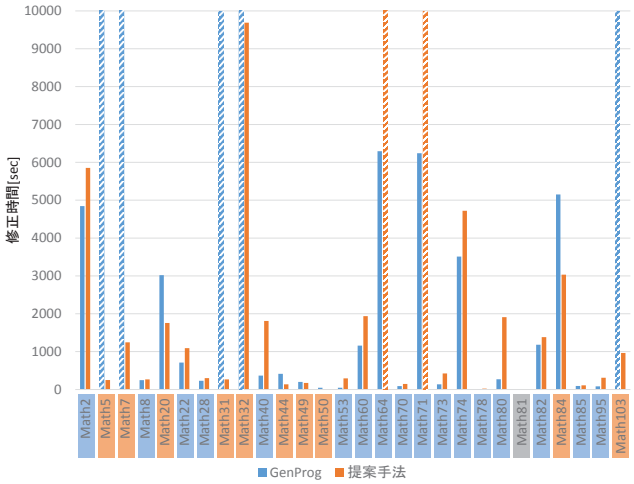
6.3 実験結果

実験結果を図 5 および図 6 に示す．図 5 の散布図に示されている実験結果は，106 個の欠陥に対する 3 回の実験（合計 318 回の欠陥修正の試み）のうち，提案手法を用いなかった場合と用いた場合でどちらも修正に成功した場合の実験結果を示している．グラフは対数スケールで表している．縦軸は提案手法の修正時間，横軸は GenProg の修正時間を示している．修正時間の単位は秒であり，全てのテストケースを通過する変異プログラムが生成されるまでにかかった時間を示している．グラフ中の丸は 1 つの欠陥に対する修正時間を示している．青色の対角線上に丸があれば提案手法と GenProg の修正時間が等しいことを示し，青線よりも右側に丸がある場合は GenProg よりも提案手法のほうが早く修正を終えたことを示す．青線よりも左側に丸がある場合は，提案手法よりも GenProg のほうが早く修正を終えたことを示す．赤線は実験結果の回帰直線である．

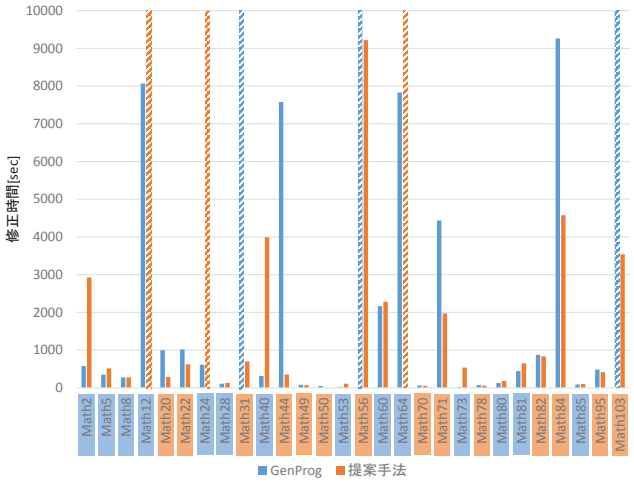
図 5 から，GenProg において挿入候補を選択する際に提案手法が提案するプログラム文を用いた場合に，修正時間に差がある 63 の欠陥修正のうち，29 回の欠陥修正において提案手法を用いなかった場合よりも高速に修正を行った．このことから修正を早く終えた欠陥修正の数で比較すると提案手法と GenProg はほぼ同数である．しかし，提案手法を用いない場合，用いた場合両方で修正に成功した



(a) seed=0



(b) seed=1



(c) seed=2

図 6: 欠陥修正に要した時間

欠陥に対する平均修正時間を比較すると，提案手法を使わない場合は 1,379.7 秒，提案手法を用いた場合は 1,127.8 秒であり，提案手法によって平均修正時間を 18.3%削減した．図 5 の回帰直線に注目すると，修正時間が 700 秒以内では GenProg のほうが早く修正を終えるが，700 秒を超えてか

表 2: 実験結果の詳細 (seed=0)

修正対象	GenProg		提案手法			差 [sec]
	修正時間 [sec]	正しいパッチ	修正時間 [sec]	推薦された文	正しいパッチ	
Math-2	196	No	1,350	No	No	1,154
Math-5	268	Yes	325	Yes	Yes	57
Math-7	4,588	No	2,463	Yes	No	-2,125
Math-8	218	No	218	No	No	0
Math-20	1,628	No	3,082	–	No	1,454
Math-22	412	Yes	298	Yes	Yes	-114
Math-28	152	No	174	Yes	No	22
Math-31	8,374	No	666	Yes	No	-7,708
Math-40	5,286	No	596	No	No	-4,690
Math-44	2,976	No	351	Yes	No	-2,625
Math-49	432	No	62	Yes	No	-370
Math-50	77	No	41	Yes	No	-36
Math-53	28	No	153	No	No	125
Math-56	–	No	8,091	Yes	No	–
Math-60	3,225	No	2,996	Yes	No	-229
Math-70	175	No	110	Yes	No	-65
Math-71	5,003	No	7,956	–	No	2,953
Math-73	21	No	403	No	No	382
Math-74	7,962	No	–	–	No	–
Math-78	–	No	307	Yes	No	–
Math-80	141	No	146	-	No	5
Math-81	735	No	302	-	No	-433
Math-82	376	No	534	No	No	158
Math-84	3,407	No	267	No	No	-3,140
Math-85	25	No	23	Yes	No	-2
Math-95	719	No	619	Yes	No	-100
Math-103	–	No	1,767	No	No	–
平均	1,672.3	–	1,005.9		–	-666.4

らは提案手法のほうが大幅に早く修正を終える傾向があることが分かる．このことから提案手法は特に GenProg が修正に時間がかかる対象の高速化に有用であると言える．

図 6 は，GenProg と提案手法の少なくともどちらか一方が修正に成功した欠陥の修正時間を示している．図 6(a)，図 6(b)，図 6(c) は順に 1 回目の実行結果，2 回目の実行結果，3 回目の実行結果である．縦軸は修正時間，横軸は実験対象を示している．青色は GenProg の実験結果，赤色は提案手法の実験結果を示している．各実験対象についている色はどちらの手法が早く修正を終えたかを示している．網状のマスクがかかった実験結果はタイムアウトした実験結果を示している．図 6 において，一方のみが修正できた実験結果に着目すると，GenProg は 6 個，提案手法は 11 個であり，提案手法は GenProg に比べて 5 個多くの欠陥の修正に成功した．

表 2 は図 6(a) の詳細な実験結果を示す．提案手法は，推薦されたプログラム文を用いて修正が完了しなかった場合に GenProg と同様，プログラム文をランダムに取得して修正に用いる．各表には修正時間の数値データに加え，5 列目に提案手法で行われた修正が提案手法によって推薦されたプログラム文かどうかを示している．Yes なら提案手法が推薦したプログラム文によって全てのテストケースを通過するプログラムが生成され，修正に成功したことを示す．No なら，GenProg と同様にランダムに選択されたプログラム文により修正に成功したことを示す．

全ての実験結果のうち，推薦されたプログラム文を用いて修正に成功した実験結果の修正時間に注目する．推薦されたプログラム文によって修正に成功した実験結果のうち提案手法のほうが早く修正を終えたものは 28 個，GenProg のほうが早く修正を終えたものは 14 個であり，66.7%の欠陥に対して提案手法手法が早く修正を終えた．また，修正時間が大きく変化した場合について考察するため，GenProg と提案手法の修正時間に 100 秒以上の差がある 23 個の実験結果に着目する．それらの実験結果について，提案手法のほうが早く修正を終えたものは 18 個，GenProg のほうが早く修正を終えたものは 5 個であり，提案手法が推薦したプログラム文によって修正を終えた場合に，提案手法は 78.3%の欠陥に対して修正時間を大きく短縮するということが分かった．

GenProg を含めた，テストの通過状態を用いた現在の自動プログラム修正手法が行う修正は，全てのテストを通過するパッチを出力するが開発者が行う修正と意味的に等し

```
--- a/src/main/java/org/apache/commons/math3/complex/Complex.java
+++ b/src/main/java/org/apache/commons/math3/complex/Complex.java
@@ -302,7 +302,7 @@ public class Complex implements FieldElement<Complex>, Serializable  {
         }

         if (real == 0.0 && imaginary == 0.0) {
-            return INF;
+            return NaN;
         }

         if (isInfinite) {
```

図 7: 開発者と等価な修正パッチの例 (Math-5)

```
--- a/src/main/java/org/apache/commons/math3/distribution/FDistribution.java
+++ b/src/main/java/org/apache/commons/math3/distribution/FDistribution.java
@@ -272,7 +272,7 @@ public class FDistribution extends AbstractRealDistribution {

     /** {@inheritDoc} */
     public boolean isSupportLowerBoundInclusive() {
-        return false;
+        return true;
     }
--- a/src/main/java/org/apache/commons/math3/distribution/UniformRealDistribution.java
+++ b/src/main/java/org/apache/commons/math3/distribution/UniformRealDistribution.java
@@ -181,7 +181,7 @@ public class UniformRealDistribution extends AbstractRealDistribution {

     /** {@inheritDoc} */
     public boolean isSupportUpperBoundInclusive() {
-        return true;
+        return false;
     }
--- a/src/main/java/org/apache/commons/math3/special/Beta.java
+++ b/src/main/java/org/apache/commons/math3/special/Beta.java
@@ -170,9 +170,9 @@  public class Beta {
     public static double logBeta(double a, double b) {
+        double m;
         return logBeta(a, b, DEFAULT_EPSILON, Integer.MAX_VALUE);
     }
```

図 8: 開発者と等価な修正パッチの例 (Math-22)

```
diff --git a/src/main/java/org/apache/commons/math/complex/Complex.java b/src/main/java/org/apache/commons/
math/complex/Complex.java
index ab58c78..e0a8e97 100644
--- a/src/main/java/org/apache/commons/math/complex/Complex.java
+++ b/src/main/java/org/apache/commons/math/complex/Complex.java
@@ -150,9 +150,6 @@ public class Complex implements FieldElement<Complex>, Serializable  {
     public Complex add(Complex rhs)
         throws NullArgumentException {
         MathUtils.checkNotNull(rhs);
-        if (isNaN || rhs.isNaN) {
-            return NaN;
-        }
         return createComplex(real + rhs.getReal(),
             imaginary + rhs.getImaginary());
     }
```

図 9: 全てのテストケースは通過するが正しくないパッチの例 (Math-53)

いとは限らない [14]．本研究では，GenProg，提案手法によって行われた修正が開発者が行った修正と意味的に等価かどうかについて調査した．調査は目視で行った．また，3 回行った実験のうち 1 回の実験結果について調査を行った．表 2 の 3 行目，6 行目の正しいパッチという項目は，GenProg，提案手法によって行われた修正が開発者が行った修正と等価かどうかを示している．調査の結果，Math-5 と Math-22 に対して提案手法，GenProg 共に正しいパッチを生成できた事が分かった．

図 7 に提案手法と GenProg が Math-5 に対して生成したパッチを示す．Math-5 では図 7 の if 文内で返す値が INF であった為に未通過テストが存在していた．提案手法，GenProg により if 文内から返す値を INF に変更するパッチが生成され，修正に成功した．また，パッチは開発者が作成したパッチと同じであった．

図 8 に提案手法と GenProg が Math-22 に対して作成したパッチを示す．Math-22 では図 8 の isSupportLowerBoundInclusive メソッドと isSupportUpperBoundInclusive メソッド内で返す boolean 値が間違っていた．そのためこの欠陥の修正には複数箇所の変更を正しく行う必要がある．提案手法，GenProg は各メソッドから返す値を正しいプログラム文に変更するパッチが生成し修正に成功した．但し提案手法，GenProg が生成したパッチは，プログラムの入出力としては開発者のものとは同じものの，Beta クラス内の logBeta メソッド内に不要な double 型の変数 m を追加していた．

Math-5，Math-22 以外の欠陥に対して生成されたパッチは全て開発者が生成したパッチとは異なるパッチであった．全てのテストを通過するが必要な機能が削除されてしまっているパッチの例を 2 つ示す．図 9 に Math-53 に対して開発者が作成したパッチを示す．Math-53 で Complex クラスの add メソッドにおいて，引数 rhs が NaN である

場合の処理が不足しており，テストに失敗していた．開発者はこの欠陥に対して Complex クラスの add メソッド内にメソッドの引数 rhs が NaN かどうかをチェックする処理を記述した．対して，提案手法が作成したパッチでは，MathUtils.checkNotNull メソッドの呼び出しを削除し，開発者と同様の処理を追加していた．その為，テストは通過するが，引数が Null のときに行うべき処理が削除されている．

GenProg が上記したパッチを生成することは Long らが報告している内容と一致する [14]．一方で，Le らは自動プログラム修正手法によって生成されたパッチを並び替え，開発者によって生成されたパッチである可能性が高いパッチを生成する手法を提案している [15]．本研究で提案した手法を用いてより多くのパッチを生成し，HDRepair[15] と組み合わせる事により，より多くの実験対象に対して，より早く開発者が作成するパッチと等価なパッチを生成できるようになると考える．

提案手法は，変更予測手法を用いて自動プログラム修正に用いるプログラム文を絞り込むが，全ての修正において変更予測手法が正しく予測を行えているわけではない [9]．提案手法を利用したプログラム文の再利用だけではなく，自動プログラム修正の過程でランダムなプログラム文の選択も混ぜることによって，より短い時間で修正が完了できる欠陥もあるかもしれない．

7. 妥当性の脅威

実験対象

本研究では提案手法を Java で記述された OSS の開発過程で発生した 106 個の欠陥に対して適用した．他の実験対象や Java 以外の言語で書かれたプログラムを修正対象とした場合に異なった結果が得られる可能性がある．

欠陥修正コミット

提案手法で欠陥修正コミットを特定する際に欠陥の修正に関連した “bug” や “fix”, “bugfix” などのキーワードを含むコミット全てを欠陥修正コミットとした．そのためキーワードを含むが実際には欠陥修正コミットではないコミットを欠陥修正コミットとして扱っている可能性がある．

jGenProg における抽象構文木

jGenProg では修正対象プログラムの抽象構文木を Eclipse JDT 等の広く使われているライブラリではなく独自の形式で扱っている．jGenProg における抽象構文木では，プログラム要素を約 40 種類しか定義していない．提案手法は EclipseJDT を用いているため予測の精度に影響は考えにくいが，予測結果を用いたプログラム文の絞り込みの際により多くプログラム文が推薦されてしまっている可能性がある．JDT を用いて jGenProg を再実装した場合に異なった結果が得られる可能性がある．

機械学習ライブラリ

本研究では，Weka のライブラリを用いて学習モデルを構築している．Weka は学習モデルを構築する際に様々なオプションを設定する事ができる．本研究では，デフォルト設定のまま学習モデルを構築した．異なったオプションや他の機械学習ライブラリを用いた場合に本研究で得られた結果と異なる結果が得られる可能性がある．

8. あとがき

本研究では，ソースコードの再利用に基づく自動プログラム修正手法を高速化するため，ソースコードの変更予測手法を用いた自動プログラム修正の高速化手法を提案した．既存のソースコードの再利用に基づく自動プログラム修正手法は欠陥箇所を変更するプログラム文を修正対象のソースコードからランダムに選択する．プログラム文は修正対象中に大量に存在するためこれは非効率である．そこで提案手法では，ソースコードの変更予測を用いて次に追加されるプログラム要素を特定し，そのプログラム要素を持つプログラム文を修正に用いる．実際の OSS 開発で発生した欠陥に対して提案手法と既存手法を適用した結果から，提案手法は 63 回の欠陥修正のうち，29 回において既存手法よりも高速に修正を行った．また，既存手法よりも 5 つ多くの欠陥を修正した．提案手法を用いない場合，用いた場合両方で修正に成功した欠陥に対する平均修正時間の比較から，提案手法は平均修正時間を約 18.3%削減できた．

今後は，提案手法をより多くの欠陥や異なった言語のプログラムに適用し，その有効性を検証する予定である．また，欠陥修正を行う前に欠陥修正に必要な時間を予測する手法についても研究を行いたい．

謝辞 本研究は，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．

参考文献

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak and Tomer Katzenellenbogen, "Reversible Debugging Software - Quantify the time and cost saved using reversible debuggers," 2013. University of Cambridge, http://docplayer.net/11413249-Reversible-debugging-software-quantify-the-time-and-cost-saved-using-reversible-debuggers.html, Accessed 2017-01-04.

[2] 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," In Proceedings of the 34th International Conference on Software Engineering, pp3–13, Zurich, Switzerland, Jun. 2012.

[3] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," In Proceedings of the 36th International Con-

ference on Software Engineering, pp.254–265, Hyderabad, India, May 2014.

[4] H.D.T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," In Proceedings of the 35th International Conference on Software Engineering, pp.802–811, San Francisco, CA, USA, May 2013.

[5] L. De. Moura and N. Børner, "Z3: An efficient smt solver," In Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, pp337–340, Budapest, Hungary, Mar. 2008.

[6] S. Mechtaev. J. Yi and A. Roychoudhury. "DirectFix: Looking for Simple Program Repairs," In Proceedings of the 37th International Conference on Software Engineering, pp.448–458, Florence, Italy, May 2015.

[7] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," In Proceedings of the 35th International Conference on Software Engineering, pp.772–781, San Francisco, CA, USA, May 2013.

[8] R. Just , D. Jalali and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," In Proceedings of the 36th International Symposium on Software Testing and Analysis, pp.437–440, San Jose, CA, USA, Jul. 2014.

[9] H. Murakami, K. Hotta, Y. Higo and S. Kusumoto, "Predicting Next Changes at the Fine-Grained Level," In Proceedings of the 21st Asia-Pacific Software Engineering Conference, pp.126–133, Jeju, Korea, Dec. 2014.

[10] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.298–312, St. Petersburg, FL, USA, Jan. 2016.

[11] S. Mechtaev, J. Yi and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," In Proceedings of the 38th International Conference on Software Engineering, pp.691–701, Austin, Texas, May 2016.

[12] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp.166–178, Bergamo, Italy, Aug. 2015.

[13] Z. Qi, F. Long, S. Achour and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems," In Proceedings of the 18th International Symposium on Software Testing and Analysis, pp.24–36, Baltimore, MD, USA, Jul. 2015.

[14] F. Long and M. Rinard, "An analysis of the search spaces for generate and avalidate patch generation systems," In Proceedings of the 38th International Conference on Software Engineering, pp.702–713, Austin, Texas, May 2016.

[15] X. D. Le, D. Lo and C. Le Goues, "History Driven Program Repair," In proceedings of the IEEE 23rd international Conference on Software Analysis, Evolution, and Reengineering, Osaka, Japan, March 2016.

[16] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," In Proceedings of the 25th International Symposium on Software Testing and Analysis, SaarbrÃijcken, Germany, Jul. 2016.