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.

ソースコードの変更予測手法による自動プログラム修正の高速化

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修正手法では，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するため，既存手法は修正に時間を要する．一方で，プログラムの構文情報と開発履歴を入力として，次の変更後にで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．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知ることが可能である．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することが可能である．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，実際の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，提案手法は 12 パターン中 9 パターンで既存手法よりも早く修正を行えることが分かった．既存手法よりも修正に時間がかかった 3 パターンについても，最大でも 10 秒程度と既存手法と同程度の時間で修正を終えられている事が分かった．平均修正時間で比較すると約 50%の削減であった．

キーワード　デバッグ，プログラム自動修正，コード再利用，機械学習，遺伝的プログラミング

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．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 [2]．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．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 [2]．

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，これは非効率である．一方で，プログラムの構文情報と開発履歴を入力として，次の変更後にで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知ることが可能である．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することが可能であると考える．

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 種類に分けられる．

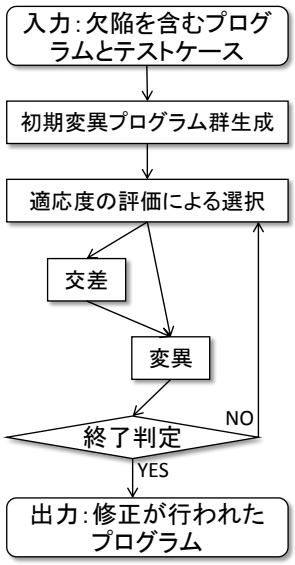


図 1: GenProg の動作の流れ

- 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 [12] [11]
- 修正パターンに基づく手法 [7] [13] [10]

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でき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，欠陥箇所に関わる全ての欠陥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 [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 [5]．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．

そこで SemFix を改良した DirectFix が提案された [12]．DirectFix は，SemFix よりも修正に時間を要するが，人が理解しやすい修正を生成することが可能であ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 と同程度の時間で複数箇所に対応した効率的に欠陥の修正を行うことができたと報告されている．

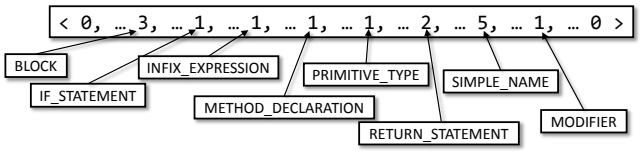
修正パターンに基づく手法である PAR は Null チェックやオブジェクトの初期化など 10 個の修正パターンを定義しており，それらに基づいて修正を行う [7]．Kim らは 6 つの OSS への適用によって GenProg よりも多くの欠陥を修正することに成功し，理解しやすい修正を生成できたと報告している．SPR も修正パターンに基づく手法の 1 つである [13]．SPR は条件式の変更や値の変更，制御フロー文の導入など，6 つの修正パターンを定義してプログラムの修正を行う．PAR と比べて抽象的な修正パターンが定義されており，より多くの欠陥を修正することが可能であることや，条件式や値の探索を枝刈りによって効率的に行っていることが特徴である．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 よりも多く行うことができたと報告している．

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) を行うことが可能となるよう，SPR を改良した手法である [10]．Prophet はまず OSS の開発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することが可能である．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

```
public int max(){
  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

案手法について説明し，実際の OSS に対して提案手法を適用した結果を報告する．

3. 準　　備

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．ソースコードから抽象構文木を構築し，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 2a のメソッドを状態ベクトルにしたものが図 2b である．図 2a のメソッドには 8 種類のノードが出現している．そのため，状態ベクトルの 8 つの要素は 1 以上の値を持ち，残りの要素が 0 となる．

3.2 ソースコードの変更予測

本研究で用いる村上らのソースコードの変更予測手法について説明する．この手法はプログラムの開発履歴と変更予測対象の状態ベクトルを入力として受け取り，次に修正が加えられた後の状態ベクトルを出力する．予測結果と予測対象の状態ベクトルの差分を取る事によって，次の変更で追加，削除されるプログラム要素を知ることが可能である．

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．ソースコードの変更予測手法では，まずはじめに対象プログラムの開発履歴を入力とし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 つ目が次の変更で行われる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ような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．次に 1 つ目のモデルに状態ベクトルを与え，次の変更が大きいか小さいかを予測する．次の変更が小さいなら 2 つ目のモデルに状態ベクトルを与え，変更後の状態ベクトルを

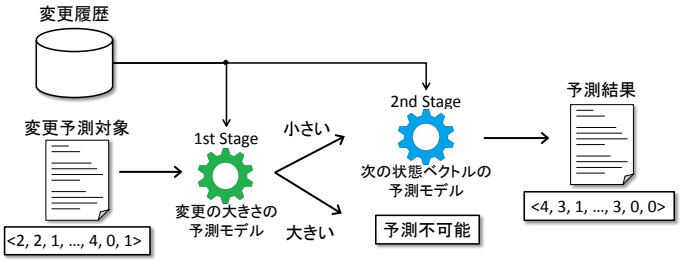


図 3: ソースコードの変更予測手法の概要

得る．次の変更が大きいなら，状態ベクトルの予測は行うことができない．変更の大きさは，変更前後の状態ベクトルのマンハッタン距離で表され，距離が定められた閾値以下なら変更が小さいとする．また予測アルゴリズムには，1 つ目に k 近傍法，2 つ目に重回帰分析を用いている．

村上らはこの手法を 2 つの OSS に対して適用し，変更の大きさの閾値が 3 の場合に約 70%の精度で次の状態ベクトルの全ての要素を正しく予測を行うことができたと報告している．

4. 提 案 手 法

本研究では，バージョン管理されているプログラムにおける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，修正箇所の情報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，村上らの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，自動プログラム修正の挿入候補中からそのプログラム要素を持つプログラム文を探し出して推薦する．

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．そのため，提案手法によって推薦されたプログラム文を用いることで自動プログラム修正を高速化することが可能であると考える．

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を状態ベクトルとプログラム文の集合とし，データベースに格納する．STEP2 では，修正対象プログラム，修正箇所の情報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推薦を行う．推薦されたプログラム文を自動プログラム修正に用いることで修正を高速化することが可能であると考える．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．

4.1 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，挿入候補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．そして抽象構文木を辿ることにより，プログラム文とその状態ベクト

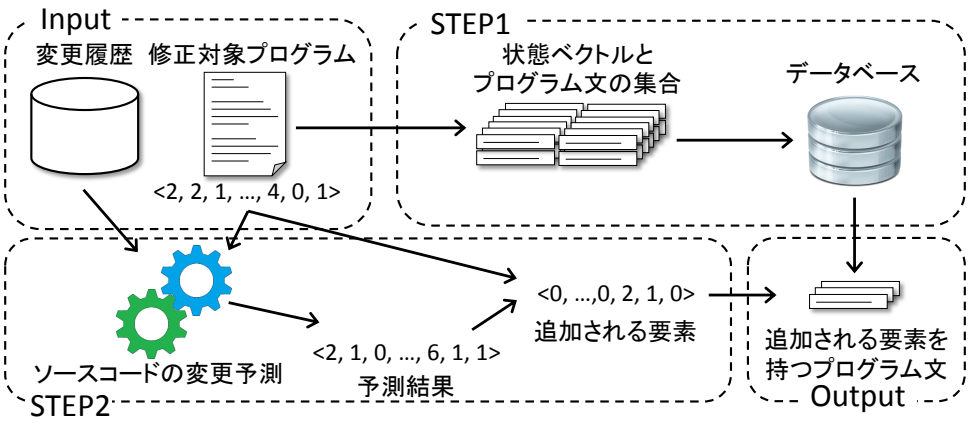


図 4: 提案手法の概要

ルをデータベースに格納する．

4.2 STEP2:プログラム文の推薦

STEP2 では，修正プログラムの状態ベクトル，修正箇所の情報，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．次に行う操作が置換なら，置換される文の状態ベクトルも追加されるプログラム要素の状態ベクトルに加える．最後に，STEP1 で構築したデータベースから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得ることが可能である．

4.3 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．

4.3.1 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，操作を行うことが可能であり，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 STEP 1 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 JDT の Statement クラスのサブクラスである抽象構文木の頂点を根とする全ての部分木を辿る事により取得した．

4.3.2 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することが可能であると考える．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．

5. 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 4 つの欠陥に対して GenProg を適用し，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比較を行う．実験は各欠陥に対して 3 回行う．

GenProg などの，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 [14]．そのため 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な修正 (以下，正しい修正とよぶ) であったかどうかについても調査する．正しい修正かどうかの判定は著者らが目視で行った．正しい修正に関する情報は Defects4J [8] より得た．また本実験では GenProg を Java に対して適用するため，Java のプログラムを修正可能な JGenProg [15] の挿入候補選択部分を変更して実験を行った．

5.1 実 験 対 象

実験対象には Defects4J [8] を用いた．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．Defects4J の詳細を表 1 に示す．Defects4J には以下の特徴を持つ欠陥が収集されている．

- 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

実験対象を表 2 に示す．実験対象の Math-5,70,73,95 は Defects4J の BugID を示している．提案手法で用いるソースコードの変更予測手法は次の変更が小さい場合にのみ変更の予測が可能であり，提案手法が有効に働くと考えられる．これらの欠陥は小さい変更で修正可能であり，提案手法が有効に働く例になると考えたため実験対象とした．

5.2 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おり，同じ欠陥の修正であっても実行時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，各変異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．本実験では GenProg と提案手法をより厳密に比較するため，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，変異プログラムのインデックスの時に同じ箇所に同じ操作が加わるようにした．

5.3 実験結果

実験結果を表 3 に示す．表 3 において，左の列から順に BugID は実験対象，シードは乱数生成器に与えるシード値を示している．GenProg，提案手法はそれぞれ挿入候補のプログラム文をランダムに選んだ場合と提案手法によって推薦されたプログラム文を使用した場合を示している．修正時間は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，等価は開発者が行った修正と等価であったかどうかを示している．実験結果の下線はもう一方の実験条件よりも短時間で修正を行えたことを示している．

実験結果から，GenProg の挿入候補を選択する際に提案手法によって提案されたプログラム文を用いた場合に 12 パターン中 9 パターンで提案手法を用いなかった場合よりも高速に修正を行うことができた．また，GenProg が提案手法よりも早く修正を行ったパターンが 3 つあったが，提案手法との実行時間の差は全て 10 秒以内であり，提案手法を用いた場合でもほぼ同じ時間で修正を行うことできた．平均修正時間は提案手法を使わない場合は 209.67 秒，提案手法を用いた場合は 103.33 秒であり，提案手法によって平均修正時間を 50.71%削減する事ができた．また，GenProg と提案手法によって行われた修正が正しい修正であったどうかについても調査を行った．調査の結果，提案手法を使わない場合は 12 パターン中 11 パターン，使った場合は 10 パターンで正しい修正を行うことができた．

6. 妥当性の脅威

本研究では提案手法を Java で記述された OSS で発生した 4 つ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．今回の実験では小さな変更で修正可能な欠陥を実験対象とした．そのため，提案手法が有効に働きにくいと考えられる，修正に大きな変更を必要とする欠陥に対しても GenProg と同程度の時間で修正を行えるかどうか調査を行う必要がある．

また，本研究では，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．

7. 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

表 1: Defects4J の詳細

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

表 2: 実験対象

BugID	修正前リビジョン（日付）		修正後リビジョン（日付）	
Math-5	f5bcba81	(2013/05/22)	e54a1c92	(2013/05/22)
Math-49	0b78d9ac	(2011/08/13)	50d4715	(2011/08/13)
Math-70	f184aeb7	(2010/04/25)	f41fcd85	(2010/05/03)
Math-73	1687372e	(2010/02/21)	29c3b75e	(2010/02/23)
Math-95	e640d161	(2008/09/16)	fbf87122	(2008/09/16)

表 3: 実験結果

BugID	シード	GenProg		提案手法	
		修正時間 [sec]	等価	修正時間 [sec]	等価
Math-5	0	257	Yes	195	Yes
	1	300	Yes	191	Yes
	2	175	Yes	185	Yes
Math-70	0	20	Yes	29	Yes
	1	39	Yes	34	Yes
	2	10	Yes	11	Yes
Math-73	0	412	Yes	118	Yes
	1	209	Yes	52	Yes
	2	180	Yes	23	Yes
Math-95	0	512	Yes	276	Yes
	1	137	Yes	8	No
	2	275	No	118	No
平均		209.67	–	103.33	–

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 12 パターン中 9 パターンで既存手法よりも早く修正を行うことができた．既存手法よりも修正に時間がかかった 3 パターンについては，最大でも 10 秒程度と既存手法と同程度の時間で修正を終えていた．平均修正時間で比較すると約 50%の削減をすることができた．

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．

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：JP25220003) 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] 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

[3] 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

[4] 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

[5] 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

[6] 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.

[7] 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

[8] R. Just , D. Jalali and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," ISSTA'14 pp.437–440, 2014.

[9] H. Murakami, K. Hotta, Y. Higo and S. Kusumoto, "Predicting Next Changes at the Fine-Grained Level," APSEC'14 pp.126–133, 2014.

[10] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," POPL '16 pp.298–312, 2014.

[11] S. Mechtaev, J. Yi and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," ICSE '16 pp.691–701, 2016.

[12] S. Mechtaev, J. Yi and A. Roychoudhury, "DirectFix: Looking for Simple Program Repairs," ICSE '15 pp.448–458, 2015.

[13] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," ESEC/FSE '15 pp.166–178, 2015.

[14] Z. Qi, F. Long, S. Achour and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems," ISSTA '15 pp.24–36, 2015.

[15] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," ISSTA '16, 2016.