kGenProg: 高拡張性・高処理効率性・高可搬性を備えた自動プログラム修正システム

柗本 真佑[1,a)]　肥後 芳樹[1,b)]　有馬 諒[1]　谷門 照斗[1]　内藤 圭吾[1]　松尾 裕幸[1]
松本 淳之介[1]　富田 裕也[1]　華山 魁生[1]　楠本 真二[1]

概要：ソフトウェア開発において，効率的なデバッグ作業の実現を目的とした自動プログラム修正に関する研究が数多く行われている．自動プログラム修正ではバグを含むソースコードとテストスイートを入力とし，自動的にバグ修正の施されたソースコードを出力する．本稿では，著者らが開発している自動プログラム修正のための研究用プラットフォーム kGenProg について紹介する．kGenProg はプログラム修正の過程に対して遺伝的アルゴリズムを採用しており，生物の進化や淘汰を模倣したプログラム修正を実現する．kGenProg は既存のプログラム修正システムと比較して，高拡張性・高処理効率性・高可搬性の 3 つの特徴を持つ．評価実験として，実際のバグを対象としたバグ修正実験を行い，既存のバグ修正ツールと比べて処理速度の向上を確認した．

1. はじめに

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大な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の報告もある [3], [6]．そのため，デバッグの支援はソフトウェア開発の効率化やソフトウェアの信頼性の向上に有益である．

デバッグ支援に関する研究がこれまでに数多く行われてきており，自動プログラム修正と呼ばれる技術が近年注目を集めている．自動プログラム修正とは，バグのあるプログラムから完全に自動でバグを取り除く技術である．GenProg [23] はこの自動プログラム修正分野にブレイクスルーをもたらしたツールである．GenProg の基本的なアイデアは，遺伝的アルゴリズムに基づき，バグのあるプログラムをバグのない状態に徐々に近づけていく，という点にある．GenProg は 8 つのオープンソースソフトウェア（OSS）に対して適用され，105 個中 55 個の欠陥の修正に成功した，との報告もある [17].

しかし GenProg には，実行時間が長いという課題がある．文献 [17] の実験では，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 分であった．GenProg では，そのアルゴリズムにおいて乱択を行う箇所が多く，アルゴリズムの改良の余地が多分にある．アルゴリズムを改良することで，より多くのバグの修正，あるいはより短時間でのバグ修正が期待される．

現在，著者らは遺伝的アルゴリズムを用いた自動プログラム修正のための研究用プラットフォーム **kGenProg** [*1] を開発している．kGenProg の主な特徴を以下に示す．

- 高拡張性
- 高処理効率性
- 高可搬性

kGenProg は，自動プログラム修正の研究を行う研究者と自動プログラム修正技術を利用したい開発者を利用者として想定している．一つ目の特徴である高拡張性は，主に研究者を対象とした特徴である．kGenProg は遺伝的アルゴリズムを利用した自動プログラム修正における様々な箇所のアルゴリズムを簡単に置き換えることができるように設計されている．既に述べたように，現在の遺伝的アルゴリズムを利用した自動プログラム修正技術はそのアルゴリズムにおいて改良の余地が多分にある．kGenProg を利用することで，新しいアルゴリズムを考案した研究者は，そのアイディアがうまく作用するのかを容易に実験可能である．

二つ目の特徴である高処理効率性は，研究者と開発者のどちらにも重要である．例えば，Java を対象とした自動プログラム修正ツールでは，以下の処理内容が必須である．

- 対象プログラムのソースコードの変更

[1] 大阪大学, Osaka University
[a)] shinsuke@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp

[*1] https://github.com/kusumotolab/kGenProg

- 変更したソースコードのコンパイル
- バイトコードへのカバレッジ計測命令の埋め込み
- テストの実行

上記全ての処理は通常はファイルシステムへの I/O を伴う処理である．遺伝的アルゴリズムを利用した自動プログラム修正では，これらの処理を非常に多くの回数行う．その場合に，ファイル I/O が処理速度のボトルネックとなりやすい．kGenProg では高処理効率性のため，上記の全ての処理を JavaVM のヒープ内で行う．

三つ目の特徴である高可搬性も，研究者と開発者のどちらにも重要である．公開されている既存の自動プログラム修正ツールの中には，特定の環境下でのみ利用可能なものもある．二つ目の特徴で述べたように，自動プログラム修正ツールは内部に様々な処理を持つため，多数のライブラリに依存している．そのため，対象プロジェクトの動作環境が，自動プログラム修正ツールを適用可能な環境と合致しない場合は，その利用が難しい．また，既存ツールは，ダウンロードした後に種々のパラメータの設定や特定のディレクトリ構成を構築する必要があり，すぐに利用開始することは難しい．一方，kGenProg は Java で開発されており，kGenProg の実行に必要なものは実行用 Jar ファイルのみである．kGenProg の種々のパラメータには著者らがこれまでの経験から有効と判断したデフォルト値が設定されているため，利用者は対象プロジェクトのルートディレクトリを kGenProg に与えるだけで実行できる．

本論文では，kGenProg の上記 3 つの特徴についてそれぞれ紹介する[*2]．さらに，特徴の一つである高処理効率性について評価実験を行う．評価実験では，複数の OSS で発生したバグをまとめた Defects4J データセット [11] を用い，jGenProg とのバグ修正能力，及びその修正時間の比較を行う．

2. 自動プログラム修正技術

2.1 概要

自動プログラム修正技術は，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受け取り，全てのテストが成功するプログラムを出力する技術である．この技術においてキーとなる要素は，自動バグ限局[*3]，及び自動プログラム改変の 2 つである．自動バグ限局ではソースコードのどの箇所にバグが含まれていそうか，というバグ箇所の推定を自動で行う．その情報に基づき，実際にプログラムの特定の箇所に対して改変を行う．一般的に，自動プログラム修正ツールは，その内部に自動バグ限局の機能も内包する．以降，本節では自動バグ限局技術，及び自動プログラム修正技術について，既存技術を紹介する．

[*2] なお，本論文は著者らの国際会議 APSEC2018 でのポスター発表 [7] の内容を再整理しフルペーパーとしてまとめたものである．
[*3] Fault localization

2.2 自動バグ限局

自動バグ限局とは，バグのある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受け取り，バグ箇所の推測を行う技術である．自動プログラム修正技術を適用するための前処理としては，実行経路情報に基づくバグ限局[*4]が利用される．この手法は，以下の二種類の情報を用いてバグの原因箇所となっているプログラム文を推測する．

- 各テストの成否
- 各テストで実行されたプログラム文のリスト

実行経路情報に基づくバグ限局手法はこれまでに数多く提案されている．その一例を表 1 に示す．直感的には，失敗テストが通過するプログラム文ほど疑惑値（バグを含む可能性）が高くなり，成功テストが通過するプログラム文ほど疑惑値が低くなる．

Abreu らは，7 つの実行経路情報に基づくバグ限局手法を C 言語のデータセット [9] を利用して比較し，Ochiai [4] が優れた手法であると結論づけている [1]．Yang らは，自動プログラム修正では，自動バグ限局で得られた疑惑値に基づくルーレット選択を用いて修正箇所を決定する方が，疑惑値のランキングに基づいて修正箇所を決定するよりも，多くのバグを修正できたと報告している [26]．

2.3 自動プログラム修正

自動プログラム修正技術は，生成と検証[*5]に基づく手法 [12], [23], [24] と，プログラムの意味論[*6]に基づく手法 [15], [21], [25] に大別される．前者は元のプログラムをある戦略に基づいて改変し，その後にテストの成否を確認する手法である．一方，後者はプログラムとテストスイート

表 1 自動バグ限局手法の一例

手法名	疑惑値 $susp$ の計算式
Ample	$susp = \left\| \frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}} \right\|$
Jaccard	$susp = \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$susp = \frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})\times(a_{ef}+a_{ep})}}$
Zoltar	$susp = \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+10000\times a_{nf}\times\frac{a_{ep}}{a_{ef}}}$

a_{ep}: その行を実行した成功テストの数
a_{ef}: その行を実行した失敗テストの数
a_{np}: その行を実行しなかった成功テストの数
a_{nf}: その行を実行しなかった失敗テストの数
（$a_{ep}+a_{ef}+a_{np}+a_{nf}$ = テスト総数）

[*4] Spectrum-based fault localization
[*5] Generate and validate
[*6] Correct by construction

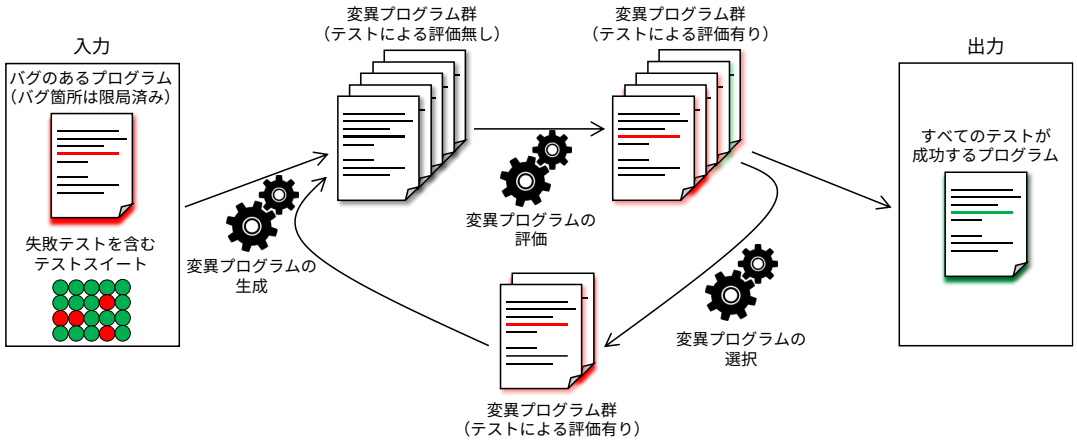

図 1　遺伝的アルゴリズムを用いた自動プログラム修正

からプログラムが満たすべき条件を特定し，その条件を満たすようにプログラムを合成する手法である．生成と検証に基づく手法を利用した場合には，全てのテストが成功するプログラムが生成されるまでに大量のプログラムが生成されることもあるが，意味論に基づく手法を利用した場合には，プログラムが合成できた場合にはそのプログラムは全てのテストが成功する．

生成と検証に基づく手法は，どのようにプログラムを生成するかによって，遺伝的アルゴリズム[*7]を利用した手法と [20], [23]，変更パターンを利用した手法 [13], [16] に分けることができる．遺伝的アルゴリズムを用いた手法は，単一のプログラム文のような比較的小さな単位の挿入や削除を繰り返し行うことでプログラムを徐々にバグの無い状態に近づけていく手法である．変更パターンを利用した手法は，何らかの方法で収集した過去の変更と同じ変更をバグのあるプログラムに加える手法である．

2.4　遺伝的アルゴリズムを利用した自動プログラム修正

著者らが開発している kGenProg は，遺伝的アルゴリズムを利用した手法であるため，本節では以降，この手法について説明を行う．図 1 にこの手法の概要を表す．図に示すように，遺伝的アルゴリズムを用いた手法は以下の 3 つの処理を含む．

- 変異プログラムの生成
- 変異プログラムの評価
- 変異プログラムの選択

変異プログラムの生成は，存在している変異プログラムから新しい変異プログラムを生成する処理である．変異プログラムの生成は，変異と交叉の二種類の方法がある．変異はある単一の変異プログラムから新しい変異プログラムを生成することを表し，交叉は 2 つ以上の変異プログラムから新しい変異プログラムを生成することを表す．変異については，用いられる操作は以下の 3 つである．

挿入：バグの原因と思われるプログラム文の前もしくは後に別のプログラム文を挿入する．
削除：バグの原因と思われるプログラム文を削除する．
置換：挿入と削除の両方を行う．

挿入操作，及び置換操作では，利用するプログラム文を取得する必要がある．遺伝的アルゴリズムを利用した自動プログラム修正の初期のツールである GenProg では，対象プログラム内からランダムに挿入に利用するプログラム文を取得している [17]．また，Monperrus らによる GenProg の再実装である jGenProg では，利用するプログラム文はランダムに取得するという点においては GenProg と同じであるが，利用候補を同一ファイル内，同一パッケージ内，同一プロジェクト内から選択できるようにオプションが用意されている [20]．バグの原因箇所と類似しているコード片から優先的にプログラム文を利用するという手法も提案されている [10], [27]．プログラム文ではなく，より粒度の細かいプログラム式の単位で挿入操作や削除操作を行う手法 [24] や，ブロック単位での挿入を行う手法 [12] も提案されている．

3.　kGenProg

3.1　概要

kGenProg は Java 言語を対象とした，遺伝的アルゴリズムに基づく自動プログラム修正プラットフォームである．kGenProg は MIT ライセンスに基づく OSS として開発されており，自由に利用，改変が可能である．以降では，kGenProg の持つ 3 つの特徴，高拡張性，高処理効率性，高可搬性について，それぞれ説明する．

3.2　高拡張性

生成と検証に基づく自動プログラム修正は，「テストが成功するまでプログラムの改変を続ける」手法であり，その手法の細部に様々な選択肢が存在する．例えば，プログラムの改変手法としては，バグ限局されたプログラム文の

[*7] Genetic algorithm

```
public interface FaultLocalization {
  public List<Suspiciousness> exec(
    SourceCode source, // ソースコード
    TestResults result // テスト実行の結果
  );
}
```

図 2　自動バグ限局のインタフェース

削除のような単純な方法 [5] だけでなく，遺伝的アルゴリズムに基づいた交叉を適用する方法 [14], 過去の開発者による変更を参考にする方法 [13], [16]，条件分岐文の条件を変更する方法 [18] など多数存在する．他にも，自動バグ限局手法は表 1 にも示した通り複数存在しており，プログラム修正の効率に強く寄与することが知られている [26]．

kGenProg では，これら自動プログラム修正における個々のフェーズ（バグ限局やプログラム改変等）の処理に対し，ストラテジパターンを適用することで高い拡張性を実現している．各フェーズの処理はインタフェースとして切り離されており，具体的にどのような処理が適用されるかは，kGenProg プラットフォーム側からは隠蔽される．表 2 に現在 kGenProg で差し替え可能な戦略一覧を示す．自動バグ限局手法や，変異個体の生成方法，適用する交叉の戦略，変異個体の評価尺度（適応度の計算方法）などが自由に差し替え可能である．

戦略の具体的なソースコード例として，自動バグ限局の Java インタフェースを図 2 に示す．`FaultLocalization#exec` メソッドはソースコードの集合とテスト実行の結果をパラメタとして受け取り，各プログラム文に対応する疑惑値の集合を返す．ここで，バグ限局手法の一つである Ochiai を新たに実装することを考える．開発者は，図 2 のインタフェースを実装する新たなクラス Ochiai を作成し，表 1 の計算式を当てはめるだけで，新たなバグ限局手法を適用することが可能となる．

3.3　高処理効率性

遺伝的アルゴリズムに基づく自動プログラム修正は，一種の最適化問題であると捉えることもできる．通過するテスト数の最大化が目的関数であり，生成されるソースコード群が解の集合，ソースコードの変異方法が探索空間に該当する．ソースコードの変異方法は，どの場所（限局文全体や限局文内の演算子等）に，どの操作を（挿入や削除等），どのように（i++文の挿入や，単純な削除等）適用するかの組み合わせで決定される．よって探索空間は膨大である．加え，上記変異処理によって生成された全ての解に対して，目的関数を適用する必要がある．言い換えれば，全ての生成ソースコードに対するコンパイルとテスト実行が避けられない．効率的な自動プログラム修正の実現のためには，解空間の効率的な探索が必要であり，そのためにはコンパイルとテスト実行という多大な時間を要する処理の高速化が必須である．

表 2　kGenProg で差し替え可能な戦略一覧

戦略	戦略の具体例
自動バグ限局手法	Ample / Jaccard / Ochiai
交叉の方法	一点交叉 / 多点交叉 / 一様交叉 [14]
変異個体の生成方法	挿入 / 削除 / 置換 / 交叉
変異個体の評価尺度	テスト通過率 / 通過率+コードの良さ*1
変異個体の選択方法	上位のみ*2 / 上位+少数の下位個体*3

*1 行数等を評価に加えることで複雑なコードの生成を回避．
*2 エリート主義 [2]．良い個体のみを次世代に残す．
*3 遺伝子の多様性を確保する工夫．局所解の回避に繋がる．

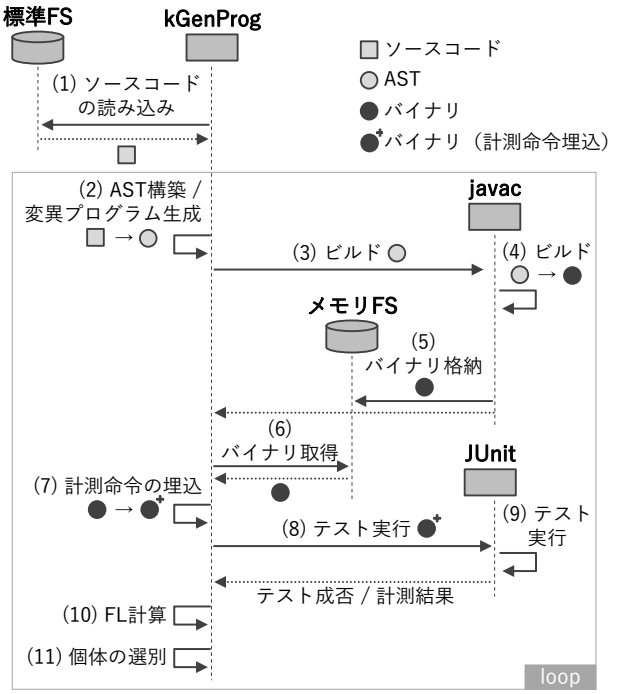

図 3　kGenProg の処理フロー

効率的な空間探索を実現するために，kGenProg では処理のボトルネックとなりやすいファイル I/O を伴う処理を，JavaVM のヒープ内で行うという戦略を取っている．より具体的には，変異とコンパイルの対象となる抽象構文木（AST），及びテスト実行の対象となるバイナリデータを Java のオブジェクトとして表現し，それらをメモリ上の論理的なファイルシステム（FS）上に設置する．このメモリ FS の利用により，最初期に行われるソースコードファイルの読み込み，及び終了時に行われる結果の書き出しを除き，全処理の Java ヒープ上での実行を実現する．

kGenProg の処理の流れを図 3 に示す．まず (1) kGenProg は標準ファイルシステムからソースコードを読み込み，そのデータを Java のオブジェクトとして保持する．次に，(2) このソースコードオブジェクトに対し，AST を構築し変異プログラムを生成する．(3) javac を用いてビルドを行う．この時，javac の出力となるバイナリデータは，標

準 FS ではなくメモリ上の論理 FS に書き出すように設定が加えられている．(4) javac がビルドを行い，(5) バイナリデータをメモリ FS に書き出す．kGenProg が (6) メモリ FS からバイナリを取得し，(7) 各テストでどの文が実行されたかを計測するための命令を当該バイナリに埋め込む．(8) (9) JUnit を用いてテストを実行し，各種テスト実行の結果を取得する．最後に (10) 自動バグ限局を行い，(11) 個体の選別を行う．(2) から (11) までの手順を繰り返し，遺伝的アルゴリズムに基づく自動プログラム修正を行う．

さらに，高処理効率を実現するために再計算回避のための工夫も取り入れている．生成と検証に基づくプログラム修正手法では，偶然同じ個体（つまり同じ AST 構造を持つソースコード）を生成する可能性がある．このような個体の生成は探索済み解空間の再探索に繋がるため，枝刈りによる再探索回避が必要である．kGenProg では AST のハッシュ値を記憶しておくことで，同一個体の再探索を回避している．また，ある個体の中で，一部のソースコードが以前に生成 / ビルド済みであったというケースも発生する．例えば，2 つのソースコード `X` と `Y` を持つプログラムを考える．ここで，ある変異個体の一つが，`X'` は新規に生まれたユニークなコードではあるが，`Y'` は以前にコンパイル済みであったとする．この場合，`X'` の変異によりプログラムの振る舞いが変わる可能性があるため，個体の評価（コンパイルとテスト実行）が必要である．図 3 で示したメモリ FS は，コンパイル結果のバイナリを蓄えるキャッシュとして働くよう設計されている．kGenProg ではこのキャッシュを用いることにより，`Y'` に対する再コンパイルを回避している．

3.4 高可搬性

自動プログラム修正技術の普及や発展，さらに提案ツールの実用性の確保のためには，研究者のみならず一般的な Java 開発者が手軽に利用できることが重要である．そのために，kGenProg では可搬性を確保するためのいくつかの工夫が施されている．

第一に，自動プログラム修正における各種パラメタを設定ファイルとして切り出すことができる．自動プログラム修正では，プロダクトソースコードへのパス（`src/main/java` 等）や，テストソースコードへのパス（`src/test/java` 等）といった対象プロジェクト固有の構成情報に加え，遺伝的アルゴリズムの生成個体数，最大世代数，乱数の値といった様々なパラメタを調整する必要がある．kGenProg では，これらの情報を設定ファイルに記述し，対象プロジェクトのルートパス直下に設置するだけで利用することが可能である．

kGenProg のインストールと利用の流れを図 4(a) に示す．図に示す通り，(1) カレントパスの移動，(2) 設定ファイルの作成，(3) jar バイナリのダウンロード，(4) 実行の 4 手順のみで利用が可能である．このような対象プロジェクト直下に設定ファイルを設置するという起動構成は，ビルドツール（Gradle や Maven 等）や，CI ツール（TravisCI や CircleCI 等）でも広く用いられている．この方法を採用することで，kGenProg を様々なビルドツールや CI ツールに組み合わせることが容易となる．例えば，kGenProg と CI 環境を組み合わせた *"Nightly repair"* のような利用方法について考える．まず，開発者は kGenProg の設定ファイル（`kGenProg.conf`）を自身のプロジェクトのリポジトリに登録する．さらに，CI 上でのテスト失敗時に図 4(a) の (3) ダウンロードと (4) 実行のコマンドを起動するように，CI 環境の設定を加える．これにより，CI 環境上で master ブランチのテストが失敗した際に，同 CI 環境の上で自動的にバグの修正を試みるといった利用が可能である．

```
# (1) カレントパスの移動
$ cd YOUR_PROJECT

# (2) 設定ファイルの作成
$ vi kGenProg.toml
src  = ["src/main/java"]
test = ["src/test/java"]

# (3) jarバイナリのダウンロード
$ curl -LO https://github.com/.../releases/
    download/v1.0.0/kGenProg.jar

# (4) 実行
$ java -jar kGenProg.jar
```

(a) 基本的な利用の流れ

```
# (4) 実行（CUIからのパラメタの上書き）
$ java -jar kGenProg.jar \
  --config kGenProg.toml \ # 基本設定
  --max-generation 100   \ # 最大世代数
  --headcount 10         \ # 各世代の個体数
  --random-seed 1        \ # 乱数シード
  --scope PACKAGE          # 再利用候補範囲
```

(b) CUI 上でのパラメタの上書き

図 4 kGenProg の利用の流れ

また可搬性確保のための工夫として，設定ファイルの CUI からの上書きも可能である．パラメタ上書き時の実行方法を図 4(b) に示す．先述の通り，自動プログラム修正は最適化問題の一つであり，対象プロジェクトに応じた適切なパラメタの調整作業が欠かせない．このパラメタ調整のために，パス情報などのプロジェクト固有かつ変更されにくい情報を設定ファイルに記述し，最大世代数などのプログラム修正の効率に関わる変更されやすい情報を CUI 上から指定することができる．なお，各パラメタにはデフォ

ルト値が設定されており，明示的にパラメタを指定しなくても利用は可能である．

さらに，継続的デリバリ [8] のプラクティスを取り入れることにより高い可搬性と利用性を確保している．図 4(a) の手順 (3) でのダウンロード対象となる jar バイナリは，全依存ライブラリが梱包されており，その実行は極めて容易である．加え，このバイナリの生成と配布に対しては，CI の援用によるリリース管理を行っている[*8]．安定版は著者ら開発チームでの合意でリリースしており，2〜3 週間ごとの更新を目標としている．またナイトリービルド版に関しても，毎晩 GitHub リポジトリの master ブランチを対象に生成されている．これらの工夫により，kGenProg の利用者は任意のタイミングで最新版を手軽に利用することが可能である．

4. 評価

4.1 実験概要

kGenProg の評価実験として，jGenProg [20] とのバグ修正時間の比較を行う．本実験の目的は，kGenProg の特徴の一つである高処理効率性が，実際のプログラム修正に対してどの程度効果があるかを確かめる点にある．比較対象とした jGenProg は自動プログラム修正分野のブレイクスルーとなった GenProg の Java 実装版であり，kGenProg と同様，Java 言語が対象，かつ遺伝的アルゴリズムに基づいた自動プログラム修正ツールである．

4.2 実験題材

実験題材として Defects4J のデータセット [11] を用いる．Defects4J データセットには，複数の OSS プロジェクトで実際に発生したバグの情報が含まれている．バグ情報には，いつ，どのソースコードのどの箇所にバグが発生したか，さらにはそのバグが，どのように，誰によって修正されたかといった様々な情報が含まれる．加え，そのバグによってどのテストが失敗したかという情報も含まれており，自動プログラム修正に対する入力（バグを含むソースコードとテスト）が用意されている．よって，自動プログラム修正分野で広く用いられており，ベンチマークとしても活用されている [19]．

本実験では Defects4J に含まれる Apache Commons Math プロジェクト（以降，Math）を実験題材として用いる．Math には 106 個のバグ情報が含まれており，いずれも実際の Math 開発の中で発生したバグである．Math を題材として選定した理由は，jGenProg の提案論文 [20] や jGenProg のベンチマーク論文 [19] において，Math に含まれるバグが最も多く修正されていたためである．

[*8] https://github.com/kusumotolab/kGenProg/releases

4.3 実験設定

実験設定の一覧を表 3 に示す．変異プログラムの生成や選択において，kGenProg，jGenProg 共に乱択に基づいた操作が含まれる．よって単一のバグ修正試行ごとに，0 から 9 までの 10 個の乱数シードを設定し修正を試みる．単一試行あたりの制限時間は 30 分とし，それを超えた場合はバグ修正失敗と見なす．変異プログラムの生成方法としては，kGenProg，jGenProg 共に，乱択によるプログラム文の挿入，バグ限局箇所の削除，それら 2 つを同時に行う置換，及び 2 つの個体からの交叉，の 4 つの操作をランダムに適用する．この挿入と置換操作においては，どの範囲からプログラム文を再利用するかというスコープの設定が可能である．kGenProg におけるスコープとしては，改変対象のファイルを起点として，そのファイルのみ，そのファイルの属するパッケージ，そのファイルの属するプロジェクトの 3 つを指定できる．スコープが広いほど探索空間が広く，つまり解を発見できる可能性は上がるが多くの時間を要するようになり，スコープが狭いとその逆となる．実験では kGenProg，jGenProg 共に空間の広さが中程度である同一パッケージをスコープとする．世代数の上限は無限とし，時間切れ，あるいは正解（つまりバグが修正された）個体を発見するまで修正を繰り返す．

kGenProg と jGenProg はその本質的なアイデアは共通であるものの，その実装や設計の細部には様々な違いが存在する．その内，最も実験に影響すると考えられる要因が回帰テストの有無である．kGenProg は回帰テストなし，jGenProg は回帰テストありである．この違いはツールの戦略の違いに起因しており，回帰テストなしでは開発者に多くのバグ修正のヒントを与えることが可能であり，対して回帰テストなしではバグ修正の誤検出を低減することができる．この違いを回避するために，jGenProg の実装を改変し，回帰テストを適用しない設定で実験を行った．

なお，実験環境としては Amazon Web Services が提供する EC2 サービスを用いており，2 個の論理 CPU と NVMe ストレージを持つ `c5d.large` インスタンスを用いた．

表 3 実験設定

項目	設定値
実験題材	Apache Commons Math
修正対象バグ数	106 個
乱数シード	0〜9（= 10 試行）
制限時間	1 試行あたり 30 分
変異生成の操作	挿入 / 削除 / 置換 / 交叉
再利用操作のスコープ	同一パッケージのみ
最大世代数	無制限
終了条件	正解個体発見か時間切れ
実験環境	AWS EC2 (2CPUs, 4GB mem)

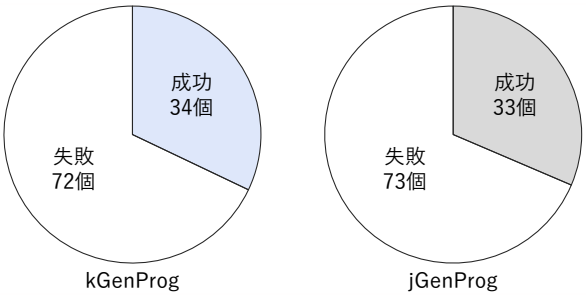

図 5 修正成功バグの割合

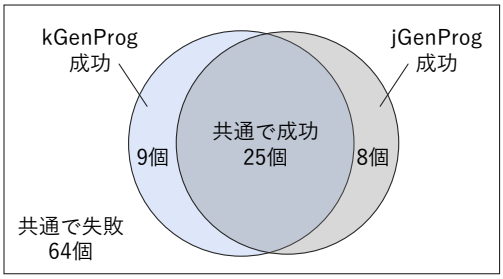

図 6 修正成功バグ集合のベン図

4.4 実験結果

4.4.1 修正成功バグ数の比較

全バグ 106 個に対する修正成功バグ数の比較を図 5 に示す．ここでは 10 個の乱数シードに対する 10 回の試行の内，一度でも修正に成功したバグを修正成功と定義している．図より，修正成功バグの数は kGenProg では 34 個（約 32%），jGenProg では 33 個（約 31%）であり，その差はほとんど見受けられない．

さらに kGenProg と jGenProg の修正成功バグの集合の関係を図 6 に示す．両方のツール共通で修正できたバグは 25 個であり，kGenProg のみで修正できたバグは 9 個，jGenProg では 8 個であった．両ツールは自動バグ修正の中でも遺伝的アルゴリズムを用いるという点で共通しており，修正可能なバグの傾向は強く似通っている．提案ツールの性能向上効果により，効率的な空間探索が可能となり，結果として修正可能なバグ数が増えることを期待していたが，その傾向は確認できなかった．

4.4.2 バグ修正時間の比較

次に処理効率を比較するために，バグ修正時間の箱ひげ図を図 7 に示す．縦軸が全試行に対するバグ修正時間を表しており，低いほど短い時間で修正に成功したことを意味する．図では，各ツールにおける修正成功バグのみを抜粋しており，この抜粋対象は図 5 の色つき部分（kGenProg は 34 個，jGenProg は 33 個）に該当する．この抜粋の理由は，実験試行のほとんどがタイムアウト（30 分）であり，その値が多数を占めることにより修正時間の比較が困難になるためである．この抜粋により，本図は「バグ修正に成功した場合にどの程度の時間で完了出来るか」を表していると解釈出来る．

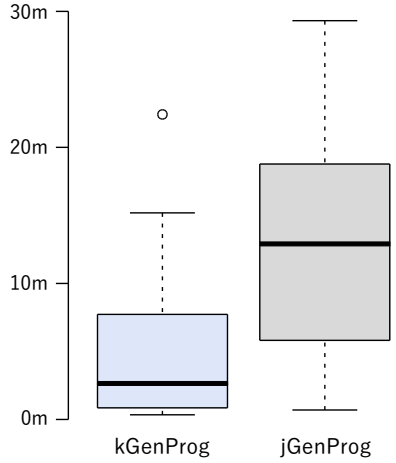

図 7 修正時間の比較

図 7 より，kGenProg が短い時間で修正に成功していることが確認できる．中央値は kGenProg では 203 秒（3 分強），jGenProg で 639 秒（10 分強）と 3 倍近い差があった．修正可能なバグの種類そのものに差は生まれなかったが，処理能力自体の向上は確認できたといえる．

さらに詳細な調査として，上記箱ひげ図の元データとなった，全バグそれぞれに対する修正時間の比較を図 8 に示す．縦軸は箱ひげ図と同様，修正成功時の平均修正時間を，横軸は Defects4J で規定された Math プロジェクトのバグ ID を示す．バグ ID は 1 を開始として連番で割り振られており，106 まで存在する．図 7 の箱ひげ図と同様に，2 つのツールで修正出来たバグ ID のみを抜粋している．縦軸上限の破線は各試行の制限時間 30 分を表しており，高さがこれに一致するケースは制限時間切れ，つまり修正失敗であることを意味する．さらに，上向きの緑矢印は 2 倍以上の速度向上を，下向きの赤矢印は 2 倍以上の速度低下を意味する．

全体として，kGenProg が jGenProg より修正時間が倍以上短いケース（例えば，左からバグ ID が 2，4，5，7 等）が多数確認できる一方で，jGenProg の方が短いケース（左から 12，30，42，44 等）も存在する．その割合としては kGenProg が短いケースが 32 個，jGenProg が短いケースが 10 個であった．また，修正時間に 2 倍以上の差があったケースに着目すると，kGenProg が短いケースが 24 個，jGenProg が短いケースが 3 個であった．生成と検証に基づくバグ修正手法では，その実験結果が乱択に強く影響する．しかしながら，10 個の乱数シード（すなわち 10 回の実験試行）で，大幅な速度改善ケースの増加が見られたことから，kGenProg の高い処理能力が確認出来たといえる．

具体的な修正例として，バグ ID80 に対する修正内容を図 9 に示す．図の修正方法は一行の削除のみであり，どちらのツールでも同様の修正が行われていた．このような修正が容易なバグに対しても kGenProg はおよそ半分の時間で修正を完了しており（kGenProg: 76 秒，jGenProg: 480

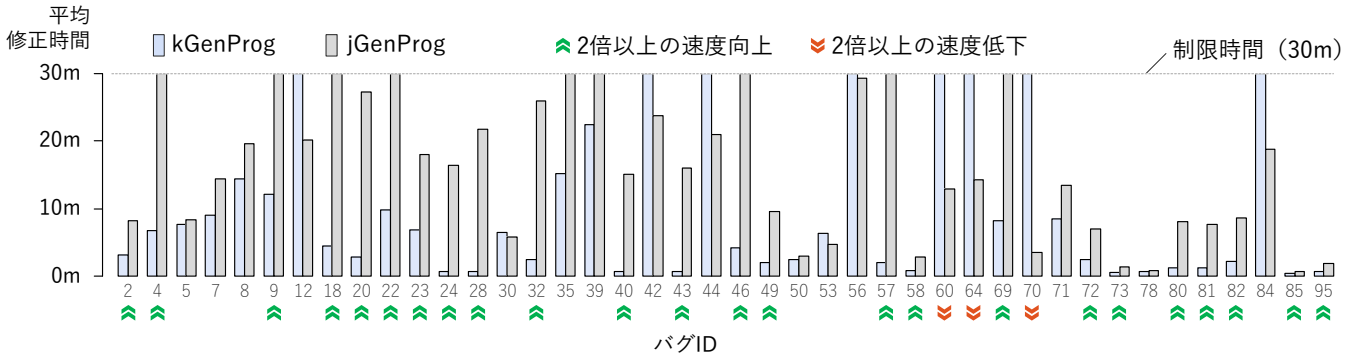

図 8　個々のバグに対する修正時間の比較

```
--- org.apache...EigenDecompositionImpl
+++ org.apache...EigenDecompositionImpl
@@ -1477,7 +1477,6 @@
int np;
if (dMin == dN) {
  gam = dN;
- a2 = 0.0;
  if (work[nn - 5]  >  work[nn - 7]) {
      return;
  }
```

図 9　バグ修正の一例（Math バグ ID: 80）

秒），処理効率の高さが確認できる．

5. 妥当性の脅威

4 節で行った実験内容について，いくつかの妥当性の脅威が存在する．まず内的妥当性として，比較対象とした kGenProg と jGenProg の比較の条件の公平性が考えられる．4.3 節でも述べた通り，実験では 2 つのツールの本質的な振る舞いの差が極力少なくなるよう，対象のソースコードを書き換えるといった工夫を施している．しかし，全ての振る舞いを同等にはできておらず，その差が実験結果に影響している可能性がある．特にバグ限局手法に関しては，kGenProg では Ochiai，jGenProg では Zoltar が適用されている．これらの手法を選択した理由は，文献 [26] では Ochiai が最も性能がよかったと指摘している点，及び，jGenProg の標準バグ限局手法が Zoltar であり，Ochiai を適用する方法が見つからなかった点に起因する．Ochiai，Zoltar 共にテストの実行経路情報に基づくバグ限局手法ではあるものの，この差が実験結果に影響している可能性がある．

外的妥当性への脅威としては，Math 以外のプロジェクトでの実験や jGenProg 以外のツールとの比較が必須である．Defects4J には Math プロジェクト以外にも JFreeChart，Closure Compiler 等の 6 種類のプロジェクトのバグ情報が含まれている．これらを題材とすることで，実験結果の外的妥当性を確保できる．

6. おわりに

本論文では，高拡張性・高処理効率性・高可搬性を備えた自動プログラム修正ツール kGenProg を提案した．高処理効率性については，既存ツールである jGenProg との比較を行った．kGenProg と jGenProg は共に遺伝的アルゴリズムを用いた自動プログラム修正ツールであり，理論的には修正可能なバグに違いはない．そのため，比較対象として jGenProg を用いることより，kGenProg の高処理効率性を評価できる．OSS で発生した 106 のバグを収集し，30 分の制限時間で，両ツールを用いて修正を試みたところ，修正可能なバグの傾向に大きな差はなかったものの，そのバグ修正に要した時間に対しては大幅な削減が確認できた．kGenProg は既に GitHub で公開されており，誰でも無料で利用することができる．

今後の課題として，本稿で評価が出来ていない 2 つ特徴，高拡張性と高可搬性の評価が必須である．これらは定量的な評価は困難であるが，被験者実験による定性評価を実施するべきである．また，より詳細な実験結果の分析も必要である．本稿での実験では，修正可能バグの種類，及びその修正時間という 2 つの観点で比較を行い，修正時間の向上という結果を得た．一方で，kGenProg のどの要因が性能向上に寄与したかというより詳細な分析は実施できておらず，重要な課題となっている．さらに近年のバグ修正研究では，開発者によるバグ修正内容との比較が広く行われるようになっている．自動バグ修正技術では，テストには通過するものの真にバグを修正出来ていない，というオーバーフィットの問題が避けられない．この修正内容の質という観点からの kGenProg の評価は一つの重要な課題である．また，kGenProg の拡張という観点では，差し替え可能な戦略の充足化や，生成と検証に基づくバグ修正の高速化手法 [22] の取り込み，各戦略の組み合わせによるバグ修正能力の評価，などの点に取り組んでいく予定である．

謝辞　本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

参考文献

[1] Abreu, R., Zoeteweij, P., Golsteijn, R. and van Gemund, A. J. C.: A Practical Evaluation of Spectrum-based Fault Localization, *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792 (2009).

[2] Ahn, C. W. and Ramakrishna, R. S.: Elitism-based compact genetic algorithms, *IEEE Transactions on Evolutionary Computation*, Vol. 7, No. 4, pp. 367–385 (2003).

[3] Britton, T., Jeng, L., Carver, G., Cheak, P. and Katzenellenbogen, T.: Reversible Debugging Software - Quantify the time and cost saved using reversible debuggers (2013).

[4] da Silva, Meyer, A., Augusto, Franco Garcia, A., Pereira, de Souza, A. and de Souza, Jr., C. L.: Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L), *Genetics and Molecular Biology*, Vol. 27, No. 1, pp. 83–91 (2004).

[5] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering*, pp. 1–1 (2018).

[6] Hailpern, B. and Santhanam, P.: Software debugging, testing, and verification, *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12 (2002).

[7] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-performance, High-extensibility and High-portability APR System, *In Proc. Asia-Pacific Software Engineering Conference*, pp. 697–698 (2018).

[8] Humble, J. and Farley, D.: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, Addison-Wesley Professional (2010).

[9] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T.: Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria, *In Proc. International Conference on Software Engineering*, pp. 191–200 (1994).

[10] Jiang, J., Xiong, Y., Zhang, H., Gao, Q. and Chen, X.: Shaping Program Repair Space with Existing Patches and Similar Code, *In Proc. International Symposium on Software Testing and Analysis*, pp. 298–309 (2018).

[11] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *In Proc. International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).

[12] Ke, Y., Stolee, K. T., Goues, C. L. and Brun, Y.: Repairing Programs with Semantic Code Search, *In Proc. International Conference on Automated Software Engineering*, pp. 295–306 (2015).

[13] Kim, D., Nam, J., Song, J. and Kim, S.: Automatic Patch Generation Learned from Human-written Patches, *In Proc. International Conference on Software Engineering*, pp. 802–811 (2013).

[14] Kou, R., Higo, Y. and Kusumoto, S.: A Capable Crossover Technique on Automatic Program Repair, *In Proc. International Workshop on Empirical Software Engineering in Practice*, pp. 45–50 (2016).

[15] Le, X. B. D., Chu, D. H., Lo, D., Le Goues, C. and Visser, W.: S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples, *In Proc. Joint Meeting on Foundations of Software Engineering*, pp. 593–604 (2017).

[16] Le, X. B. D., Lo, D. and Le Goues, C.: History Driven Program Repair, *In Proc. International Conference on Software Analysis, Evolution, and Reengineering*, pp. 213–224 (2016).

[17] Le Goues, C., Dewey Vogt, M., Forrest, S. and Weimer, W.: A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each, *In Proc. International Conference on Software Engineering*, pp. 3–13 (2012).

[18] Long, F. and Rinard, M.: Staged Program Repair with Condition Synthesis, *In Proc. Joint Meeting on Foundations of Software Engineering*, pp. 166–178 (2015).

[19] Martinez, M., Durieux, T., Sommerard, R., Xuan, J. and Monperrus, M.: Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset, *Springer Empirical Software Engineering*, Vol. 22, No. 4, pp. 1936–1964 (2016).

[20] Martinez, M. and Monperrus, M.: ASTOR: A Program Repair Library for Java, *In Proc. International Symposium on Software Testing and Analysis*, pp. 441–444 (2016).

[21] Mechtaev, S., Nguyen, M. D., Noller, Y., Grunske, L. and Roychoudhury, A.: Semantic Program Repair Using a Reference Implementation, *In Proc. International Conference on Software Engineering*, pp. 129–139 (2018).

[22] Mehne, B., Yoshida, H., Prasad, M. R., Sen, K., Gopinath, D. and Khurshid, S.: Accelerating Search-Based Program Repair, *In Proc. International Conference on Software Testing, Verification and Validation*, pp. 227–238 (2018).

[23] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically Finding Patches Using Genetic Programming, *In Proc. International Conference on Software Engineering*, pp. 364–374 (2009).

[24] Wen, M., Chen, J., Wu, R., Hao, D. and Cheung, S. C.: Context-aware Patch Generation for Better Automated Program Repair, *In Proc. International Conference on Software Engineering*, pp. 1–11 (2018).

[25] Xuan, J., Martinez, M., DeMarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D. and Monperrus, M.: Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55 (2017).

[26] Yang, D., Qi, Y. and Mao, X.: An Empirical Study on the Usage of Fault Localization in Automated Program Repair, *In Proc. International Conference on Software Maintenance and Evolution*, pp. 504–508 (2017).

[27] Yokoyama, H., Higo, Y., Hotta, K., Ohta, T., Okano, K. and Kusumoto, S.: Toward Improving Ability to Repair Bugs Automatically: A Patch Candidate Location Mechanism Using Code Similarity, *In Proc. Annual ACM Symposium on Applied Computing*, pp. 1364–1370 (2016).