テストケース選択による自動プログラム修正の効率化

松田 直也[1,†1]　丸山 勝久[1,a)]

概要：近年，バグを含むプログラムとその動作に関する正解を与えてバグのない正しいプログラムを出力する，自動プログラム修正の研究が活発に行われている．しかしながら，現時点では，自動プログラム修正を実際の開発において利用することは難しい．その原因の一つとして，プログラムの修正にかかる時間が長いことがあげられる．特に，テストケースをオラクルとして用いる探索ベースの自動プログラム修正では，大量の修正パッチ候補が内部で生成され，それらに対するテスト時間の長さが問題となっている．本論文では，実行するテストクラスの一部を削除することで，修正時間の短縮を達成する自動プログラム修正システムを提案する．バグを含む 6 個のプログラムを用いた合計 3,000 回の自動修正の試行を通して，修正に成功するプログラムの数を減らさずに，修正時間を短縮できる場面があることを確認した．

1. はじめに

ソフトウェア開発において，デバッグは困難な作業である．このため，自動プログラム修正（Automated Program Repair，以下，APR）が注目されている．APR とは，バグを含むプログラムとその動作に関する正解（オラクル）を与えて，修正したプログラム（バグを取り除くパッチ）を出力する技術である [1]．バグ修正を自動化することで，デバッグにかかるコストを大きく削減できる．

テストケースをオラクルとして用いる探索ベースの APR を Test Based Repair（TBR）と呼ぶ．TBR のプロセスは主に，バグが含まれている可能性の高い箇所の特定，バグを修正するパッチの生成，生成したパッチの検証の 3 つの工程で構成されている．パッチを適用したプログラムが与えられたすべてのテストケースに合格した（テストが成功した）場合，それが修正に成功したプログラムとなる [2]．

TBR ではその内部工程で，一般的に大量のパッチ候補を生成する．これらのパッチ候補により修正されたそれぞれのプログラムに対して，実際にテストを実行する．よって，数多くのテストケースが用意されている場合，TBR による自動修正には長い時間がかかる．APR システムを実際の開発現場で採用するためには，修正の効率化が必須である [3], [4], [5]．ここでの修正の効率化とは，修正までに費やす時間の短縮を指す．

修正の効率化を達成するためにまず思い付くのが，パッチ候補の生成数を抑制することである．そこで，制約を利用する技法 [6]，経験則に基づく技法 [7], [8]，テンプレートを利用する技法 [9]，学習に基づく技法 [10] を採用した APR システムが提案されている．これらの技法を導入することで，修正までに費やす時間の短縮が達成される反面，修正できるプログラムは一般的に限定される．さらに，パッチ候補の生成数の抑制したとしても，テストを実行する時間を除外できるわけではない．このため，パッチ候補の生成数を抑制することと直交して，テストの実行時間を短くすることが重要である．

本論文では，実行するテストケース（本論文ではテストクラス）の一部を削除するテストケース選択手法を組み込んだ APR システムを提案する．TBR のパッチ検証工程において，実行するテストクラスの数を減らすことで，より短い時間でプログラムが修正できるはずである．これにより，開発者の許容する実行時間内に修正を完了できる機会が増加する．その一方で，テストクラスを削除することでバグ箇所の特定精度が悪くなり，修正の失敗をより引き起こすことが予想される．修正の成功数が極端に小さくなれば，開発者は自動修正の恩恵を受けられない．

そこで，我々は，Defect4J [11] のプロジェクトから 6 個の修正対象プログラムを選び，合計 3,000 回の自動修正の試行において，テストケース選択手法を利用した際の修正に費やす時間と修正に成功するプログラムの数を測定した．一部の修正対象プログラムに限定されるものの，テストクラスを削除しない場合に比べて，修正時間を短縮しつつ，修正に成功するプログラムを同程度あるいはより多く出力することが確認できた．

本論文の構成は次の通りである．2 章では，テストケー

[1] 立命館大学
[†1] 現在，株式会社 はてな
[a)] maru@cs.ritsumei.ac.jp

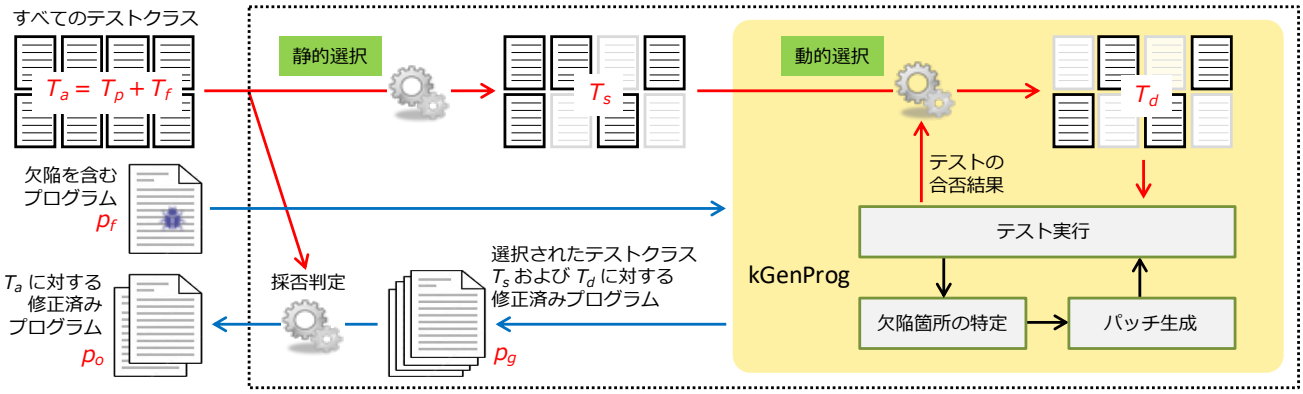

図 1　テストケース選択を組み込んだ APR システムの概要

ス選択手法を組み込んだ APR システムを提案する．3 章では，提案システムを用いた評価実験の結果を示す．4 章では，実験結果に対する考察を述べる．5 章では，関連研究としてテストケースを並べ替える APR システムを紹介する．最後に 6 章で，本論文のまとめと今後の課題を述べる．

2. テストケース選択を備えた APR システム

本章では，4 つのテストケース選択手法を組み込んだ APR システムを提案する．

2.1 システムの概要

テストケース選択手法を組み込んだ APR システムの概要を図 1 に示す．点線で囲まれた部分が提案システムである．黒色の矢印は TBR のプロセスの順序，赤色の矢印はテストクラスとそれに関する情報の流れ，青色の矢印は入出力であるプログラムの流れを表す．

APR システムとして，kGenProg [12]（1.6.1 版）[*1]を採用し，その一部を改造した．kGenProg は，遺伝的プログラミングに基づく GenProg [13] を Java プログラム向けに実装した APR システムである [14]．遺伝的プログラミングに基づく APR システムはソースコードに対する多様な変換を実現でき，複数の箇所に存在するバグを修正できる可能性が高い [8]．

提案システムに組み込まれたテストケース選択手法は，静的選択と動的選択の 2 つに分けられる．静的選択では，もとのテストケースに対する合否の結果だけを利用して，kGenProg に入力するテストクラスを削除する．これに対して，動的選択では，kGenProg におけるテスト実行のたびにその合否の結果を収集し，次世代に引き継ぐ変異個体を生成する際に，テストクラスを入れ替える．通常，一つのテストクラスには複数のテストメソッドが定義されている．しかし，kGenProg で指定可能なテストケースの最小単位はクラスである．そこで，提案システムでは，テストケースの選択（削除や入れ替え）をクラス単位で行う．

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

いま，もともと存在するすべてのテストクラスを含む集合 T_a のうち，バグを含むプログラムに対して成功するテストクラスの集合を T_p，失敗するテストクラスの集合を T_f とする．テストケース選択の前提として，T_f に含まれるテストクラスは必ず kGenProg のテスト実行の対象とする．このような前提を設定することで，kGenProg の出力する修正済みプログラム p_g は必ず T_f に含まれるテストクラスを用いたテストに合格することが保証される．

静的選択では，kGenProg に与えるテストクラスを T_p から削除する．この方法では，T_f に含まれるテストクラスは，必ず静的選択によって残る T_s に含まれる．動的選択では，kGenProg に与えられたテストクラスの集合 T_s から T_f を取り除いた集合（$T_s - T_f$）に含まれるテストクラスが削除の対象となる．これにより，T_f に含まれるテストクラスは，必ずテストクラスの削除において残る T_d に含まれることになる．

ここで，T_d を用いて修正したプログラム p_g は，T_p の中で削除したテストクラスを用いたテストに失敗する可能性がある．このような p_g を開発者は受け入れない．そこで，提案システムでは，最後に T_a に含まれるテストクラスを用いた採否判定を行うことで，開発者が受け入れないプログラムを出力から排除する．最終的に，採否判定に合格した修正済みプログラム p_o だけを出力する．

2.2 テストケース選択の方針

プログラムに変更を加えた際，その変更によって既存の振る舞いが破壊されていないかどうかを検査することを回帰テストと呼ぶ．テストケースの規模が大きくなるにつれ回帰テストの実行には膨大な時間がかかるため，回帰テストを頻繁に行うことは難しい．この問題を解決するため，回帰テストにおけるテストの回数を削減するテストケース選択手法（RTS: Regression Test Selection）が提案されている [15], [16], [17]．

RTS では，もとのテストケースを実行することで検出できる欠陥を，選択されたテストケースだけを実行すること

で検出できることを条件としている [18]．この条件を満たすためには，変更前後のソースコードにおいて変更されたコード断片および変更箇所と依存関係を持つ（影響を与える，あるいは影響を受ける）コード断片を特定することが必要である．特定したコード断片に無関係なテストケースは削減できる．言い換えると，特定したコード断片に関係のあるテストケースだけを，もとのテストケースから選択すればよい．

RTS を自動プログラム修正に取り入れた APR システムはすでに提案されている [19]．たとえば，Mehne らのシステム（SPR [20] の拡張）では，テストケースの実行経路が，生成したパッチ候補によって変更される行を含む場合に，そのテストケースを選択する [4]．Yuan らのシステム（ARJA）では，パッチ生成前に特定した変異操作を適用する文（変更される文）に対して，その文に関係しない成功テストケースを削除する [8]．これらのシステムにおいて選択されるテストケースは RTS の条件を満たすため，生成されたパッチを適用したプログラムはもとのテストケースで（理論的には）必ず成功する．

ここで，遺伝的アルゴリズムに基づく APR では，前の世代の変異個体に対して段階的にコード変換が適用される．そのため，変異個体に対して，変異操作を適用する行は世代ごとに変化する可能性がある．このことは，APR の適用前において，変異操作が適用される行が一意に特定できないことを指す．RTS を適用するためには，変異操作が適用される行が特定されている必要があり，テストケースの静的選択において，RTS の採用は原理的に不可能である．

これに対して，テストケースの動的選択においては，変異操作が適用された行を特定できるため，RTS を採用することが可能である．それにもかかわらず，提案システムでは RTS を採用しないこととした．RTS では，TBR の内部工程で生成された個々のパッチ候補において変異操作が適用された行に対して，個別にテストケースを選択する必要がある．これは，APR システムに RTS を採用する際の大きなオーバーヘッドとなると考えた．また，メソッドごとにテストケースが用意されている場合，メソッド内部に独立する実行経路が複数存在しないと，テストケース削減の効果は見込めない．特に，if-文や while-文の条件式を含む行が変異した場合，削除されるテストケースは限られる．テストケースの削除が可能な場面が限定されると，修正までに費やす時間の大幅な短縮は見込めないと考えた．

提案する APR システムでは RTS を採用しないことを踏まえて，実装の容易さの観点から，回帰テストにおけるカバレッジベースの優先順位付け [17] に着目した．その際，TBR の内部工程で生成されたパッチ候補は，ミューテーション解析におけるミュータントと見なすことができる．そこで，カバレッジベースの優先順位付けにおいて，FEP (Fault-Exposing-Potential) に基づく技法 [21] を応用して，テストクラスの削減を実現する．FEP では，複数のミュータントに対する失敗テストケースとそれを実行した際に通過するコード断片から，テストケースが失敗する確率を計算する [22]．

2.3 テストケース選択手法

以下，提案システムに組み込んだ，4 つのテストケース選択手法について述べる．

(1) 失敗に基づく静的選択 (FONLY)

T_f のみを残して，T_p をすべて削除する．テストクラスの削減数がもっとも多い選択である．

(2) 関連に基づく静的選択 (REL)

テストクラス t を実行した際に通過したプログラムのソースコードの行の集合を $L(t)$ と表す．T_f に含まれるすべてのテストクラスを実行した際に通過したプログラムのソースコードの行の集合は $L_f = \{\, l \mid l \in L_f(t_f) \wedge t_f \in T_f \,\}$ となる．この手法では，T_p に含まれるテストクラス $t_p \in T_p$ が L_f に含まれる行を 1 行以上通過する ($L(t_p) \cap L_f \neq \emptyset$) 場合に，$t_p$ を残す．言い換えると，L_f に含まれる行を 1 行も通過しない ($L(t_p) \cap L_f = \emptyset$ となる) t_p を削除する．

L_f に含まれる行はバグを含んでいる可能性が高く，それらの行（に存在するコード断片）は kGenProg において書き換え対象となる機会が多い．書き換え対象の行を通過しないテストクラスを削除しても，バグ修正への影響が少ないという予測に基づく選択手法である．

(3) カバレージに基づく静的選択 (COV)

回帰テストにおけるテストケース優先順位付け手法の一つである [23]．REL と同様に，テストクラス t を実行した際に通過したプログラムのソースコードの行の集合を $L(t)$ と表す．あるテストクラス $t_{f1} \in T_f$ に対する $L(t_{f1})$ とその他のテストクラス $t_{f2} \neq t_{f1}, t_{f2} \in T_f$ に対する $L(t_{f2})$ で重複する行の数が多いテストクラスを貪欲に選択する．重複する行の数が同じテストクラスが複数存在する場合には，それらの中からランダムに選択する．どのテストクラスを選んでも，通過する行のカバレージが増加しないとき，選択が終了する．最後まで選択されなかったテストケース $t_p \in T_p$ が削除対象となる．

重複する行が多いテストクラスを優先的に選択することで，より少ないテストクラスで REL と同じカバレージを達成できる．

(4) Patch Killing Count に基づく動的選択 (PKC)

Patch Killing Count（以下，PK 値）とは，TBR の内部工程で生成したパッチ候補に対して，各テストクラスが失敗した回数である [24]．PK 値が大きいテストクラスほど，別のパッチ候補に対してもテストの失敗を引き起こす可能性が高いと考え，次の世代のテストの実行対象とする．

いま，n 個のテストクラスを含む選択対象の集合を $T = \{ T_1, T_2, \ldots, T_n \}$ とおく．T_i の PK 値を $PK(T_i)$ とし，T_i が選択される重み $w(T_i)$ を次のように定義する．

$$w(T_i) = PK(T_i) \ / \ \sum_{j=1}^{n} PK(T_j)$$

たとえば，$PK(T_1) = 2$, $PK(T_2) = 10$, $PK(T_3) = 8$ となる 3 つのテストクラスがあるとする．このとき，$w(T_1) = 0.1$, $w(T_2) = 0.5$, $w(T_3) = 0.4$ となる．

選択するテストクラスの数 m（小数点以下は切り捨て）は，テストクラスの総数 n に対する割合 $r(\%)$ $(0 \leq r \leq 100)$ で指定する．たとえば，$r = 80\%$ のとき $m = 2$ となる．提案システムでは，$w(T_i)$ の値に応じて（$w(T_2) = 0.5$ の方が $w(T_1) = 0.1$）より 5 倍選ばれやすくした上で），ランダムに m 個のテストケースを選択する．選択されなかった $(n - m)$ 個のテストケースを削除する．

ここで，PK 値に応じてテストクラスを降順に並び替え，上位から m 個のテストクラスを選択するという戦略も考えられる．しかし，そのような戦略では，PK 値の低いテストクラスは実行されず，PK 値は 0 のままである．よって，最初の PK 値で今後選択されるテストクラスが決まってしまい，常に同じテストクラスが選択され続けることになる．そこで，提案システムでは，PK 値に応じた重みを用いてテストクラスを選択することで，テストクラスが固定化されることを回避している．

3. 評価実験

本章では，2 章で述べた提案システムを用いて，自動プログラム修正を行った実験の内容を述べ，その結果を示す．

自動プログラムの効率性に関する評価項目を以下に示す．

Q1: テストクラスを削除することで，修正にかかる時間は短縮されるか．

Q2: テストクラスを削除することで，修正に成功するプログラムの数は変化するか．

一般的に，実行するテストクラスを削除すればするほど，kGenProg による修正時間の短縮が見込める．Q1 では，どのテストケース選択手法がどの程度実行時間を短縮するのかを明らかにする．また，利用するテストクラスの増減は，kGenProg が出力するプログラムの数に影響を与えることが分かっている [25]．Q2 では，どのテストケース選択手法を採用するかによって，修正に成功するプログラムの数が増加あるいは減少するのかを明らかにする．

3.1 実験における設定

本実験におけるテストケース選択手法の設定の一覧を**表 1** に示す．表中の「-」は，静的選択や動的選択を適用していないことを指す．PKC に付与された数値は，選択するテストクラスの数の割合 r の値を示す．ORIG は，テストケース選択を一切行わないことを指す．REL-PKC は，REL による静的選択と PKC よる動的選択の両方を適用することを指す．組み合わせとしては，FONLY と PKC，COV と PKC も考えられる．しかしながら，今回の実験では FONLY および COV で選択されるテストクラスの数がそもそも少なく，動的選択を併用することでさらにテストクラスを削除する意義はないと考えた．

表 1 テストケース選択手法の設定

設定名	静的選択	動的選択
ORIG	-	-
FONLY	FONLY	-
COV	COV	-
REL	REL	-
PKC20	-	PKC (20%)
PKC50	-	PKC (50%)
PKC80	-	PKC (80%)
REL-PKC20	REL	PKC (20%)
REL-PKC50	REL	PKC (50%)
REL-PKC80	REL	PKC (80%)

表 2 kGenProg のオプションの設定

オプション	値
変異操作によって 1 つの世代に生成する個体の数	40
交叉操作によって 1 つの世代に生成する個体の数	40
選択操作によって 1 世代に残される個体の最大数	20
最大世代数	120
実行を打ち切る時間（分）	720
各個体のビルド・テストを打ち切る時間（分）	3

表 3 バグを含む修正対象のプログラム

プログラム名	プロジェクト名	id	LOC	TC
cli-37	commons-cli	37	6,212	26
codec-16	commons-codec	16	16,338	48
compress-42	commons-compress	42	42,104	97
lang-7	commons-lang	7	60,242	108
math-49	commons-math	49	126,430	284
jacksonCore-21	jackson-core	21	39,396	86

本実験において，テストケースの静的選択と修正済みプログラムに対する採否判定のためのモジュールを Python で作成した．また，動的選択のため，kGenProg のソースコードにおいて，テストクラスを選択するモジュールに改造を行った．実験には，4GHz Intel Core i7 と 32GB メモリを搭載した iMac (macOS 10.14) を用いた．

設定した kGenProg のオプションの値を**表 2** に示す．バグを含むそれぞれのプログラムに対して，kGenProg によるプログラム修正を 50 回ずつ試行した．試行における乱数発生のためのシードとして，0 ～ 100,000 の中からランダムに生成した整数を指定した．プログラムが 1 つ出力される，世代数が 120 を超える，あるいは実行時間が 720 分（12 時間）を超える場合に，kGenProg の実行が終了する．実行時エラーの発生により，これら以外の条件で実行が終了した場合，試行をやり直した．

表 4　選択されたテストクラスの数

修正対象プログラム	ORIG	FONLY	COV	REL	PKC20	PKC50	PKC80	REL-PKC20	REL-PKC50	REL-PKC80
cli-37	26	1	2	11	5	13	20	2	5	8
codec-16	48	1	2	18	9	24	38	3	9	14
compress-42	97	1	2	42	19	48	77	8	21	33
lang-7	108	1	3	20	21	54	86	4	10	16
math-49	284	1	10.02	257	56	142	227	51	128	205
jacksonCore-21	86	1	4.34	86	17	43	68	17	43	68

3.2　修正対象プログラム

本実験で利用するバグを含むプログラムは，APR のベンチマークとして利用されることの多い Defects4J（バージョン 2.0.0）[*2]から収集した．その際，修正に成功しないプログラムの修正時間を計測しても意味がないと考え，表 2 の設定で kGenProg を 5 回実行した．ここでは，5 回の実行のうち一度でも修正に成功した 6 個のプロジェクトから，識別番号が最も小さいバグを含む版を 1 つずつ選択した[*3]．

本実験で修正対象として利用したバグを含む 6 個のプログラムを**表 3** に示す．id は Defects4J におけるバグの識別番号を指す．LOC は，各プロジェクトに含まれるソースファイルの行数（テストファイルの行数を除く）を指す．TC は，本実験において，テストケース選択の対象になるテストクラスの数を示す．

3.3　結果

表 3 に示す修正対象のプログラムに対して，表 1 に示すテストケース選択手法の設定を用いて自動修正を実施した．実験におけるプログラムやデータを `http://www.fse.cs.ritsumei.ac.jp/TSAPR/` に公開する．

それぞれのテストケース選択手法の設定において，選択されたテストクラスの数を**表 4** に示す．ORIG において選択されるテストクラスの数は，表 3 の TC と等しい．COV の場合は選択されるテストクラスの数が変動するため，平均値を示す．PKC20，PKC50，PKC80 は，それぞれ ORIG の 20%，50%，80%となる．また，REL-PKC20，REL-PKC50，REL-PKC80 は，それぞれ REL の 20%，50%，80%となる．

以下，修正に費やした時間に関する結果を 3.3.1 節，修正に成功したプログラムの数に関する結果を 3.3.2 節に示す．

3.3.1　修正に費やした時間

各テストケース選択手法の設定において，kGenProg の 50 回実行に費やした修正時間 t_{fix} の箱ひげ図を**図 2** に示す．修正時間の単位は分 (m) である．修正時間には，PKC による動的選択の時間（すべての選択で 1 秒未満）を含む．

まず，修正対象プログラムごとに，テストクラス削除の効果を述べる．cli-37，lang-7，compress-21 では，どのテストケース選択手法を用いた場合でも修正時間が短縮されている．codec-16 の修正時間はきわめて短い（5 秒以内の）ため，静的選択よる修正時間の短縮が見られるものの，テストクラス削減の意義は高くない．これら 4 つの修正対象プログラムに比べて，math-49 に対する修正時間の短縮の程度はテストケース選択手法により異なる．jacksonCore-21 に対しては，テストケース選択手法を採用することでタイムアウトによる打ち切りが頻繁に発生しており，COV を除いて時間短縮はほぼ見られない．

次に，テストケース選択手法ごとに，テストクラス削除の効果を述べる．FONLY については，jacksonCore-21 を除いて，修正時間の短縮が見られる．COV については，どの修正対象プログラムに対してもテストクラスの削減率が 90%を超えており（92.3%〜95.0%），修正時間が短縮されている．REL については，cli-37，codec-16，compress-21，lang-7 に対して修正時間が短縮されているものの，math-49 と jacksonCore-21 に対する修正時間の短縮はほとんど見られない．PKC については，テストクラスの削減率（20%，50%，80%）に関係なく，cli-37，compress-21，lang-7 で修正時間が短縮されている．また，math-49 に対しては，PKC20 と PKC50 の場合にのみ修正時間の短縮が見られる．その一方で，codec-16 と jacksonCore-21 に対しては，修正時間の短縮は見られない．REL-PKC については，REL と同様に，math-49 と jacksonCore-21 以外で修正時間が短縮されている．特に，cli-37 と lang-7 に対しては，REL と PKC を組み合わせることで修正時間の短縮が達成されている．math-49 に対しては REL-PKC20 と REL-PKC50 の場合に修正時間の短縮が見られるものの，math-49 に対しては修正時間の短縮は見られない．

ここで，提案システムでは，図 2 に示す kGenProg による修正時間 t_{fix} に加えて，テストクラスを削除する時間 t_{del} と kGenProg が出力するプログラム（図 1 の p_g）の採否を判定する時間 t_{post} が必要である．さらに，t_{del} は修正対象プログラムに対して失敗するテストクラス（T_f に含まれるテストクラス）を見つける時間 t_{pre} とテストクラスを選択する時間 t_{sel} に分けられる．以上より，提案システムにおけるオーバーヘッドは，$t_{pre} + t_{sel} + t_{post}$ となる．

[*2] `https://github.com/rjust/defects4j`

[*3] 実際には，9 個のプロジェクトで修正済みプログラムが出力された．その中で，commons-collections では，実行時間が 720 分を超える打ち切りが何度も発生した．50 回の試行では実験時間が膨大になることが予想されるため，本実験の対象から除外した．closure-compiler と jackson-databind では，kGenProg により出力された修正済みプログラムが JUnit 4.12 による再テストに失敗したため，本実験の対象から除外した．

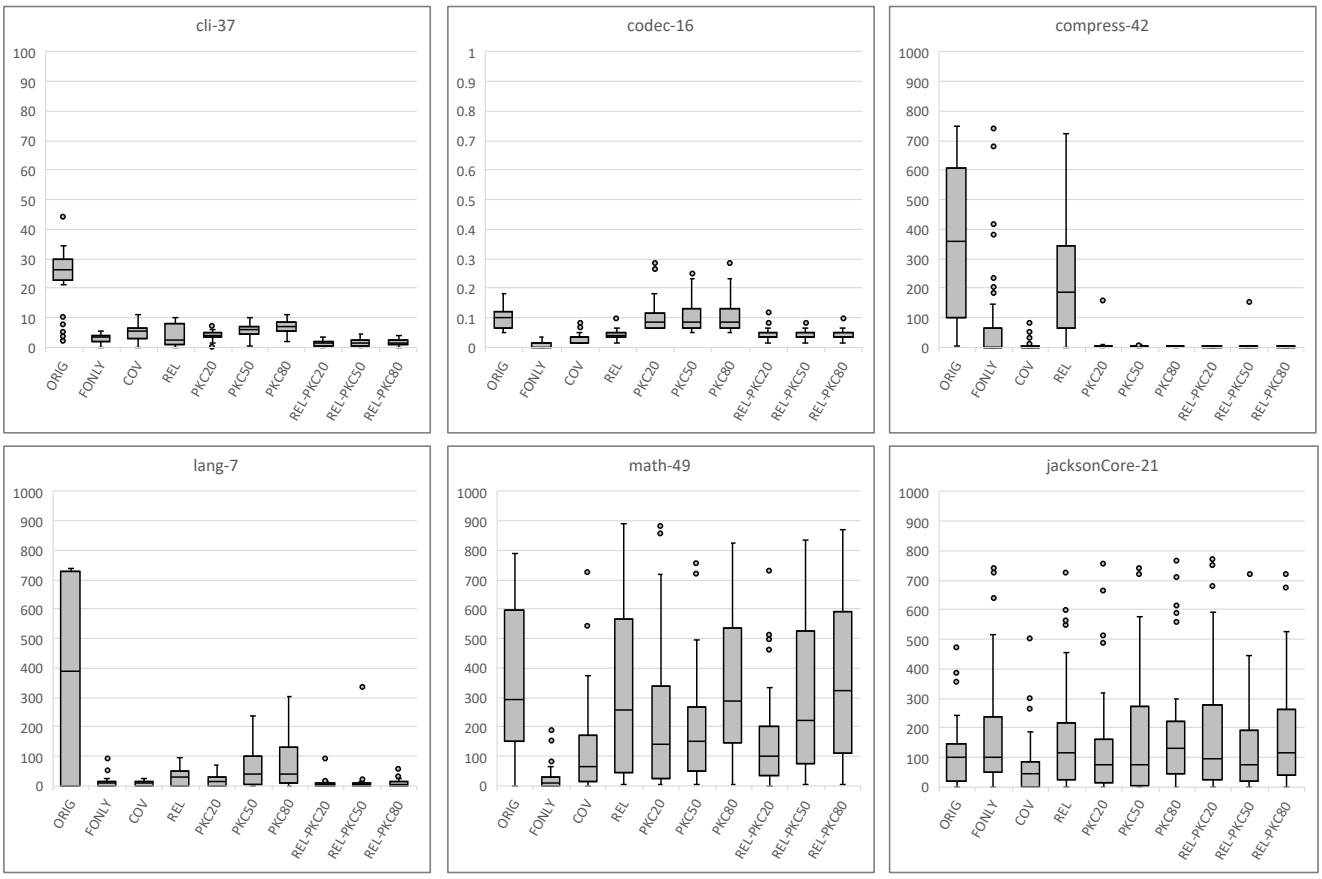

図 2　修正に費やした時間 t_{fix}

それぞれの修正対象プログラムに対して，50 回の試行における t_{pre}，t_{sel}，および t_{post} の平均実行時間を**表 5** に示す[*4]．実行時間の単位は分 (m) である．ORIG ではテストクラスの削除はないので，t_{sel} は 0 である．FONLY では失敗するテストクラスを選択するだけであり，t_{sel} はほぼ 0 であった．PKC20，PKC50，PKC80 において，動的選択の実行時間は t_{fix} に含まれており，t_{sel} は 0 である．REL-PKC20，REL-PKC50，REL-PKC80 における t_{sel} は，REL の t_{sel} と同じである．

表 5 を見ると，codec-16 を除いて，t_{pre} と t_{post} は t_{fix} に比べて圧倒的に短い．また，codec-16 を除いて，t_{sel} も t_{fix} に比べて相対的に短い．たとえば，math-49 に対して，COV による t_{sel} は約 13 分とやや長いが，ORIG の t_{fix} の平均である約 348 分に比べれば 4%程度である．以上より，codec-16 を除いて，オーバーヘッドは無視できると考えてよい．codec-16 については，t_{sel} が t_{fix} を超えている．もとの修正時間が短い場合には，オーバーヘッドが修正時間に対する短縮の効果を打ち消す場合が起こり得る．

[*4] t_{pre} と t_{post} は，3.8GHz Intel Core i7 と 64GB メモリを搭載した iMac (macOS 10.15) で計測した．

表 5　テストケース選択に関わるオーバーヘッド時間

修正対象プログラム	t_{pre}	t_{sel}(COV)	t_{sel}(REL)	t_{post}
cli-37	0.03	0.45	0.30	0.08
codec-16	0.08	1.14	0.72	0.19
compress-42	0.25	2.17	2.48	0.30
lang-7	0.25	2.61	2.62	0.47
math-49	1.01	12.31	7.34	1.90
jacksonCore-21	0.11	2.29	1.79	0.30

Q1 に対する回答

6 個中 4 個の修正対象プログラムに対して，修正に費やす時間が短縮された．その一方で，テストケース選択手法によっては，修正にかかる時間を短縮しない，あるいは増加させることがある．

3.3.2　修正に成功したプログラムの数

本実験において，1 回の試行では 1 個の修正済みプログラムを出力するか，1 個も出力しないかのどちらかである．よって，出力されるプログラムの数は修正に成功した回数と同じになる．50 回の試行において，出力されるプログラムの数を**図 3** に示す．紺色は，提案システムが出力するプログラム（図 1 の p_o）の数を指す．灰色は，kGenProg が出力したプログラム（図 1 の p_g）のうち，採否判定に合格しなかったプログラムの数を指す．たとえば，cli-37 の ORIG を見ると，テストケース選択手法を利用しない 50 回

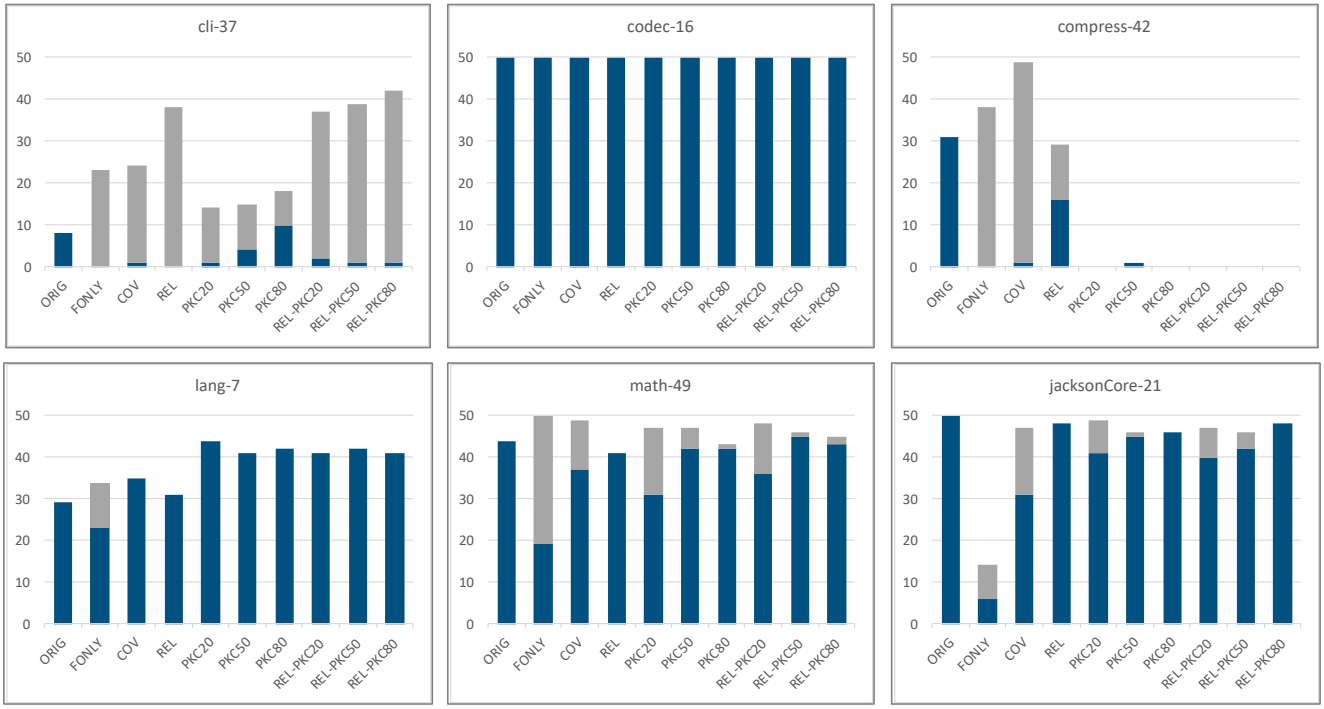

図 3 修正に成功したプログラムの数

の試行において kGenProg が出力したプログラムの数が 8 個である．これに対して，テストケース選択手法 COV を組み込んだ提案システムでは，kGenProg が出力したプログラムの数が 24 個と増加し，採否判定に合格するプログラムは 1 個となっている．採否判定に合格しなかった 23 個のプログラムは出力されない．今回の実験では，kGenProg が出力したプログラムがすべてのテストケースを満たす場合のみを修正対象プログラムとしたため，ORIG において灰色の数は必ず 0 である．

まず，修正対象プログラムごとに，成功プログラムの数に対する増減を述べる．cli-37 では，PKC80 で ORIG のときの出力数を超えているものの，その他のテストケース選択手法で成功プログラムの数が減少している．codec-16 では，テストケースの選択手法にかかわらず，50 回の試行のすべてにおいて成功プログラムを出力している．compress-42 では，どのテストケース選択手法でも成功プログラムの数が減少している．特に，REL を除くテストケース選択手法では，成功プログラムの出力数がほぼ 0 である．lang-7 においては，FONLY を除くテストケース選択手法で，ORIG における成功プログラムの出力数を上回っている．math-49 と jacksonCore-21 では，FONLY，COV，PKC20，REL-PKC20 で成功プログラムの数が減少しているものの，REL，PKC80，REL-PKC80 において ORIG と同程度の数の成功プログラムを出力している．

次に，テストケース選択手法ごとに，成功プログラムの数に対する増減を述べる．FONLY は，codec-16 を除くすべてのプロジェクトにおいて，成功プログラムの数がもっとも少ない．COV は，FONLY に比べると成功プログラムの数は多くなるものの，lang-7 を除いてその数は少ない．REL は，cli-37 と compress-42 を除いて，成功プログラムの出力数は ORIG とほぼ同じか超えている．compress-42 でも，ORIG を除いて成功プログラムの数はもっとも多い．PKC と REL-PKC は，cli-37 と compress-42 を除いて，成功プログラムの数は比較的多い．

Q2 に対する回答

テストクラスを削除することで，成功プログラムの数が減少する傾向にある．ただし，成功プログラムの数がそれほど減少しない修正対象プログラムや，逆に増加する修正対象プログラムが存在する．

4. 議論

本章では，それぞれのテストケース選択手法が有効であるかどうかの知見，実験結果に関する考察，今回の調査における妥当性の脅威について述べる．

4.1 テストケース選択手法の有効性

3.3 節に示した結果より，テストケース選択手法の有効性を議論する．

FONLY テストクラスの削減率はきわめて高く，修正時間の短縮が達成されている．しかしながら，成功プログラムを出力するという観点からは，積極的に採用する理由は見当たらない．

REL テストケースの削減率が高い 4 つの修正対象プロ

グラムで修正時間の短縮が達成できている．その反面，残り 2 つの修正対象プログラムでは時間短縮の効果はほぼない．多くの修正対象プログラムに対して成功プログラムを出力できているものの，他のテストケース選択手法に比べて実行時間の短縮という点で特段有利な結果を示しておらず，積極的に採用する理由はない．

COV FONLY と同様に，テストクラスの削減率は高く，修正時間の短縮が達成されている．また，すべての修正対象プログラムに対して成功プログラムを出力できている．よって，TBR において有効なテストケース選択手法の一つであるといえる．

PKC 4 つの修正対象プログラムで，修正時間の短縮が達成されている．また，REL と組み合わせることで，成功プログラムの数を減少させずに，修正時間を短縮する場面も観測されている．さらに，5 つの修正対象プログラムにおいて，テストクラスを削減しないときと同程度あるいはそれを超える数の成功プログラムを出力している．これらより，TBR において有効なテストケース選択手法の一つであるといえる．ただし，今回の実験では，テストクラス削減率が高いにもかかわらず，修正時間が逆に長くなってしまう場面が観測された．このことは，PKC の採用において，適切な削減率を決定することが難しいことを表している．

4.2 修正時間に関する考察

テストクラスが削減されると，それに応じてテストの実行回数が削減され，修正時間が短縮される．その一方で，修正された行に関係する成功テストクラスが削除されてしまうと，本来そのテストクラスでは失敗するはずであった修正済みプログラムが出力され，修正が早期に終了する．

また，成功テストクラスの削除が疑惑値に影響を与え，欠陥箇所が適切に絞り込めなくなる可能性がある．絞り込めなくなった場合，欠陥に関係ない箇所で変異操作が数多く適用されることになり，修正にかかる時間が長くなる．

具体的に，どのプロジェクトにおいて，どのような要因で修正時間が短縮されたのかに関する調査は今後の課題である．

4.3 修正の成功数に関する考察

修正時間が短縮されることで，タイムアウトによる修正の打ち切りが回避され，出力されるプログラムの数の増加が期待できる．その一方で，成功テストクラスの削除により，欠陥箇所が適切に絞り込めなくなることが起こる．これにより，最大世代数に到達するかタイムアウトによる修正の打ち切りが発生し，修正済みプログラムの出力数が減少する可能性がある．

また，修正された行に関係する成功テストクラスが削除されてしまうと，本来そのテストクラスで失敗するはずの（採否条件に合格しない）修正済みプログラム（図 3 の灰色部分）が出力される機会が増加する．本実験における 1 回の試行では，1 個の修正済みプログラムが kGenProg から出力されると修正は終了する．このため，本来出力されるはずの修正済みプログラム（図 3 の紺色部分）の出力が抑制される．結果的に，提案システムによる修正に成功したプログラムの数は減少する．

具体的に，どのプロジェクトにおいて，どのような要因で修正に成功したプログラムの数が減少したのかに関する調査は今後の課題である．

4.4 妥当性の脅威

今回の実験においては，kGenProg のオプションの値は著者らが経験に基づき決定した．当然ながら，オプションの値は kGenProg による修正結果に影響を与える．最大世代数を無制限と設定すれば，修正に成功する場面が増加するかもしれない．また，実行時間が 720 分 (12 時間) を超えた場合に実行を打ち切っている．50 回の試行でも，このような実行の打ち切りが観測された．このような状況で，実行を打ち切る時間を延長した場合，修正が成功する可能性が高まるだけでなく，修正時間の平均値や中央値が変化する可能性が高い．さらに，プログラムが 1 つ出力された時点で実行を終了している．修正済みプログラムが必ず受け入れられるという前提に立てば，プログラムが 1 つ出力されるまでの修正時間を計測すれば十分である．しかしながら，このような前提は一般的に成立しない．このような状況では，プログラムが 1 つ出力された時点で修正を停止するという設定を見直す必要があるかもしれない．

実験では，APR ツールとして kGenProg を利用した．TBR に基づく APR システムには，遺伝的プログラムに基づく jGenProg [26] や ARJA [8]，修正テンプレートを利用する TBar [9]，過去のプログラムから学習した確率モデルに基づく探索を利用する Prophet [10] などがある．これらのシステムにおいて，本論文で提案したテストケース選択手法を組み込んだ場合に，同様の効果が得られるかどうかは不明である．また，欠陥限局手法には，kGenProg のデフォルトである Ochiai [27] を利用した．欠陥限局手法によって，APR システムの修正能力が変わることが知られている [28], [29]．異なる欠陥限局手法を利用することで，実験結果が変わる可能性が高い．

プロジェクトに存在するテストケースが開発者が望む振る舞いを網羅しているわけではない．よって，提案システムが出力する成功プログラムの一部は，修正に成功しているものの，開発者に受け入れられない可能性がある．評価実験の結果の信頼性を高めるためには，成功プログラムを開発者に受け入れられるものと受け入れられないものに分

類する必要がある．分類の実施や分類の基準によって，評価実験の結果に基づく主張が変わる可能性がある．

本評価実験では，Defects4J に含まれる 6 個のプログラムを修正対象に選んだ．客観的な基準に従って選んだものの，6 個中 5 個は Apache Commons プロジェクトであり，修正対象プログラムの実装や用意されているテストケースがプロジェクトの特性に偏っている可能性がある．実装の詳細が修正結果に影響を与えることはすでに報告されている [5]．また，修正対象プログラムの数は少なく，今回の評価実験の結果を単純に一般化することはできない．このため，修正時間の短縮が別のプロジェクトにおいて達成される保証はない．

5. 関連研究

TBR のパッチ検証の工程において，複数のテストケースに対して一つでも失敗した時点で，そのパッチは修正を成功させないことが判明する．よって，その時点で後続のテストケースの実行を打ち切ることができる．このような観点から，失敗するテストケースをより早期に（優先的に）実行することで，テスト実行時間の削減が達成できる [21]．テストケースの実行順序の並べ替えを組み込んだ APR システムとして，TrpAutoRepair[3], RSRepair [30], MPTPS (Modification Point-aware Test Prioritization and Sampling) [24] が提案されている．これらのシステムは，実際に実行したテストケースが失敗するごとにその実行の順序を他のテストケースより優先する，FRTP (Fault Recorded Testing Prioritization) 戦略を採用している．

TrpAutoRepair は GenProg を利用してパッチの変異を実現している．その際，テストケースが失敗した時点で残りのテストケースの実行を単純に打ち切ると，各パッチに対してテストケースが成功および失敗する回数を正しく計測できない．そこで，TrpAutoRepair では，GenProg の利用において，テストケースの合否の回数に基づき次世代に引き継ぐパッチを選択するフィットネス値の計算を放棄し，第一世代のパッチだけを検査の対象としている．RSRepair はランダム探索を利用してパッチの変異を実現しているため，テストケースの合否回数の計測は不要である．

MPTPS では，実行に関して順序付けされたテストケースをいくつかの集合に（それぞれの集合のテストケースの数は均等になるように）分割することで，遺伝的プログラミングを利用した jGenProg において FRTP 戦略を実現している [24]．特定の集合でテストケースの実行が失敗した場合でも，その集合内の残りのテストケースの実行は打ち切らず，フィットネス値を擬似的に計算する．その一方で，残りのテストケースの集合に対する実行を打ち切ることで，テスト時間が短縮できる．

これらのシステムがテストケースの実行順序を並び替える戦略を採用しているのに対して，提案システムはテストクラスを削除する戦略を採用している．いま，TBR のパッチ検証の工程において，テスト実行を逐次的に実行するのであれば，失敗するテストケースをより早期に実行することでテスト時間の削減が見込める．しかしながら，テスト実行が並列に実行できる場合，並べ替えるという戦略ではテスト時間の削減は見込めない．実行するテストケースそのものを削除することで，テスト実行の順序によらずその回数だけを単純に削減できる．このような戦略を採用することで，提案システムは，逐次的なテスト実行において実行時間を短縮できるのと同時に，同一世代内のパッチ検証においてテストを並列実行できる場合にも並列数を抑えることができる．また，遺伝的プログラミングを利用した kGenProg において，MPTPS のようにフィットネス値を擬似的に計算するオーバーヘッドをなくすことができる．

6. おわりに

本論文では，TBR における修正の効率化を実現するために，4 つのテストケース選択手法を組み込んだ APR システムを提案した．Defects4J のプロジェクトから選んだ 6 個の修正対象プログラムに対する評価実験を通して，カバレージに基づく静的選択（COV）と，Patch Killing Count に基づく動的選択（PKC）が自動プログラム修正の効率化において効果を持つ場面があることが分かった．

今後の課題として，提案システムの再実装がある．提案システムの使用性を向上させるために，Python で作成したテストケースの静的選択モジュールと修正済みプログラムに対する採否判定モジュールを kGenProg に統合する予定である．また，kGenProg 以外の APR システムへの組み込みも検討している．その上で，修正対象のプログラムの数を増やした評価実験を実施する予定である．

現在のテストケース選択手法では，削除の単位をクラスとしている．これに対して，テスト実行の成功と失敗はメソッドレベルで観測できるため，テストメソッドを削除の単位とすることが考えられる．実際，Lou らの調査では，クラスレベルでの削除よりメソッドレベルでの削除の方が効果が高いことが報告されている [19]．粒度の細かなテストケース選択手法が TBR における修正の効率化に貢献するかどうかの調査を予定している．

最後に，RTS を導入したテストケース選択手法との比較が必須である．RTS を実装した APR システムを構築し，評価実験を行う予定である．

謝辞 本研究の一部は，科研費 (20H04166) の助成を受けたものである．助言を頂いた肥後芳樹氏と柗本真佑氏に感謝する．

参考文献

[1] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A survey, *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67 (2017).
[2] Weimer, W., Nguyen, T., Goues, C. L. and Forrest, S.: Automatically Finding Patches Using Genetic Programming, *Proc. ICSE '09*, pp. 364–374 (2009).
[3] Yuhua Qi, X. M. and Lei, Y.: Efficient Automated Program Repair through Fault-Recorded Testing Prioritization, *Proc. ICSM '13*, pp. 180–189 (2013).
[4] Mehne, B., Yoshida, H., Prasad, M. R., Sen, K., Gopinath, D. and Khurshid, S.: Accelerating Search-Based Program Repair, *Proc. ICST '18*, pp. 227–238 (2018).
[5] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y. L.: On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs, *Proc. ICSE '20*, pp. 615–627 (2020).
[6] Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G. and Zhang, L.: Precise Condition Synthesis for Program Repair, *Proc. ICSE '17*, pp. 416–426 (2017).
[7] Jiang, J., Xiong, Y., Zhang, H., Gao, Q. and Chen, X.: Shaping Program Repair Space with Existing Patches and Similar Code, *Proc. ISSTA '18*, pp. 298–309 (2018).
[8] Yuan, Y. and Banzhaf, W.: ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming, *IEEE Trans. Software Engineering*, Vol. 46, No. 10, pp. 1040–1067 (2020).
[9] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T. F.: TBar: Revisiting Template-Based Automated Program Repair, *Proc. ISSTA '19*, pp. 31–42 (2019).
[10] Long, F. and Rinard, M.: An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems, *Proc. ICSE '16*, pp. 702–713 (2016).
[11] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, *Proc. ISSTA '14*, pp. 437–440 (2014).
[12] 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, *Proc. APSEC '18*, pp. 697–698 (2018).
[13] Goues, C. L., Nguyen, T., Forrest, S. and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair, *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72 (2011).
[14] 柗本真佑，肥後芳樹，有馬　諒，谷門照斗，内藤圭吾，松尾裕幸，松本淳之介，富田裕也，華山魁生，楠本真二：高処理効率性と高可搬性を備えた自動プログラム修正システムの開発と評価，情報処理学会論文誌，Vol. 61, No. 4, pp. 830–841 (2020).
[15] Rothermel, G. and Harrold, M. J.: Analyzing Regression Test Selection Techniques, *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, pp. 529–551 (1996).
[16] Biswas, S., Mall, R., Satpathy, M. and Sukumaran, S.: Regression Test Selection Techniques: A Survey, *Informatica*, Vol. 35, No. 3, pp. 289–321 (2011).
[17] Yoo, S. and Harman, M.: Regression testing minimization, selection and prioritization: a survey, *Software Testing, Verification and Reliability*, Vol. 22, No. 2, pp. 67–120 (2012).
[18] Rothermel, G. and Harrold, M. J.: Selecting Tests and Identifying Test Coverage Requirements for Modified Software, *Proc. ISSTA '94*, pp. 169–184 (1994).
[19] Lou, Y., Benton, S., Hao, D., Zhang, L. and Zhang, L.: How Does Regression Test Selection Affect Program Repair? An Extensive Study on 2 Million Patches, *arXiv:2105.07311* (2021).
[20] Long, F. and Rinard, M.: Staged Program Repair with Condition Synthesis, *Proc. ESEC/FSE '15*, pp. 166–178 (2015).
[21] Rothermel, G., Untch, R. H., Chu, C. and Harrold, M. J.: Prioritizing test cases for regression testing, *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, pp. 929–948 (2001).
[22] Voas, J. M.: PIE: a dynamic failure-based technique, *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, pp. 717–727 (1992).
[23] Aggrawal, K. K., Singh, Y. and Kaur, A.: Code Coverage Based Technique for Prioritizing Test Cases for Regression Testing, *ACM SIGSOFT Software Engineering Notes*, Vol. 29, No. 5, pp. 1–4 (2004).
[24] Venugopal, Y., Quang-Ngoc, P. and Eunseok, L.: Modification Point Aware Test Prioritization and Sampling to Improve Patch Validation in Automatic Program Repair, *Applied Sciences*, Vol. 10, No. 5 (2020).
[25] 松田直也，丸山勝久：テストケースが自動バグ修正に与える影響の調査，コンピュータソフトウェア，Vol. 37, No. 4, pp. 31–37 (2020).
[26] Martinez, M. and Monperrus, M.: ASTOR: A Program Repair Library for Java, *Proc. ISSTA '16*, pp. 441–444 (2016).
[27] Abreu, R., Zoeteweij, P. and van Gemund, A. J. C.: An Evaluation of Similarity Coefficients for Software Fault Localization, *Proc. PRDC '06*, pp. 39–46 (2006).
[28] Assiri, F. Y. and Bieman, J. M.: Fault localization for automated program repair: effectiveness, performance, repair correctness, *Software Quality Journal*, Vol. 25, No. 1, pp. 171–199 (2017).
[29] Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J. and Traon, Y. L.: You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems, *Proc. ICST '19*, pp. 102–113 (2019).
[30] Qi, Y., Mao, X., Lei, Y., Dai, Z. and Wang, C.: The Strength of Random Search on Automated Program Repair, *Proc. ICSE '14*, pp. 254–265 (2014).