情報処理学会論文誌　プログラミング　Vol.12　No.3　1–8 (July 2019)

確率密度関数を用いたソフトウェアテスティングのためのテストケース優先順位付け戦略

齋藤 雄太[1,a)]　佐藤 周行[1,2,b)]

受付日 2018年12月14日, 採録日 2019年3月17日

概要：ソフトウェアテスティングは，システムを実行することで意図したとおりに動作するかどうかを検証する手法である．ソフトウェアテスティングにおいては欠陥をいかに早く発見するかが重要であり，そのために研究が進められている分野の 1 つがテストケース優先順位付け（TCP）である．カバレッジベースの TCP では，それぞれのテストがコードのどの部分を網羅したかを表すカバレッジ情報を用いて優先順位付けを行うが，既存の優先順位付け戦略では対象によって最適な戦略やパラメータが異なるという問題がある．本稿では，TCP における新しい戦略アルゴリズムとして，実行した際に新たに検出される欠陥の個数の期待値が最も高くなるテストを順に選んでいくような優先順位付け戦略を提案する．欠陥の期待値はあるユニットを実行した際にある欠陥が検出される確率の分布を表す確率密度関数を用いて計算できるため，この確率密度関数をカバレッジ情報から推定し，その分布がベータ分布に従うという仮定の下で欠陥の期待値を計算した．提案した戦略と既存の戦略について，40 個の C 言語のプログラムと 28 個の Java のプログラムに対してそれぞれ実験を行い，t 検定により比較を行った．実験の結果，statement カバレッジを用いた場合，提案した戦略によって TCP の性能が有意に向上する結果が得られた．

キーワード：ソフトウェアテスティング，テストケース優先順位付け，確率密度関数

A Strategy of Test Case Prioritization for Software Testing Using Probability Density Function

Yuta Saito[1,a)]　Hiroyuki Sato[1,2,b)]

Received: December 14, 2018, Accepted: March 17, 2019

Abstract: Software testing is a method of validating that a software system operates as intended by actually running it. In software testing, it is important how quickly to find defects, and therefore one of the areas under study is test case prioritization (TCP). Coverage-based TCP prioritizes by using coverage information that indicates which part of the code each test covers. However in existing prioritization strategy the optimal strategy and parameters depend on targets. In this paper, we propose a new strategy in TCP, which gives priority to tests with high anticipated value of the number of detection of newly found defects. Assuming the expectation value of the defect can be calculated using the probability density function representing the distribution of the probability that a certain defect is detected when executing a certain unit, we have estimated this probability density function from the coverage information and calculate the expected value of the defect under the assumption that its distribution follows the beta distribution. For the proposed strategy and existing strategy, we have conducted experiments on 40 C programs and 28 Java programs, and compared them. As a result of the experiment, TCP performance is improved in significant difference by t-test by the proposed strategy when using statement coverage.

Keywords: software testing, test case prioritization, probability density function

1. はじめに

ソフトウェアテスティングとは，ソフトウェアシステム

[1] 東京大学工学系研究科電気系工学専攻
Department of Electrical Engineering and Information Systems, The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[2] 東京大学情報基盤センター
Information Technology Center, The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan

[a)] saito@satolab.itc.u-tokyo.ac.jp
[b)] schuko@satolab.itc.u-tokyo.ac.jp

にテストデータを入力し，実行を観察して意図したとおりに動作するかどうかを検証することである [3]．厳密に欠陥がないことを証明することはできないが，厳密な検証と比較して計算量が小さいため，ソフトウェアテスティングはソフトウェア開発に不可欠であり，品質保証のために広く用いられている．ソフトウェアシステムの変更の度に多くのテストを実行する必要があるため，そのコストを削減するためにテストケース優先順位付け（TCP）が用いられる [10]．TCP の多くは，テストケースがソースコードのどの部分を実行するかを表すカバレッジ情報を用いたものである [13]．カバレッジベースの TCP 技術のほとんどは total strategy と additional strategy のいずれかの戦略をベースにしているが，それぞれに長所と短所があり，一概にどちらが良いとはいえない [6]．

本稿では以上の 2 つの戦略に代わる新しい戦略として，テストケースを実行した際に新たに検出される欠陥の個数の期待値を用いた戦略を提案する．コードの各部分を実行した際に欠陥が見つかる確率を確率密度関数の形で表すことで，新たに欠陥が見つかる個数の期待値を計算し，それが最大となるテストケースから順に選択していく．

本稿の以下の構成は次のようになっている．2 章で TCP と優先順位付け戦略，3 章で提案手法についてそれぞれ説明する．4 章では実験内容とその結果について述べる．5 章に関連研究についてまとめる．6 章では本稿のまとめを述べる．

2. TCP と優先順位付け戦略

2.1 Test Case Prioritization（TCP）

一定の指標に基づいてシステムに入力するテストケースの優先順位付けを行う技術を Test Case Prioritization（TCP）と呼ぶ [5]．TCP によってソフトウェアテスティングのテストケースの実行順序を最適化することで，欠陥をより早く発見できる，実行するテストケースを削減できるなどの利点がある．

テストスイート（テストケースの集合）T の可能なすべての置換の集合を PT，目的関数を f として，

$$(\forall P'')\,(P'' \in PT)\,(P'' \neq P')\ ,\ f\,(P') \geq f\,(P'') \qquad (1)$$

を満たすような置換 P' を求めることが TCP 技術の目標である [14]．

2.1.1 目的関数

TCP の目的関数として用いられている指標の 1 つに Average Percentage Faults Detected（APFD）[11] がある．APFD は fault によって生じる欠陥がどれだけ早い段階で検出されるかを表す．

Fault について

- $T = \{t_1, t_2, \ldots, t_N\}$：$N$ 個のテストケース t_n を含むテストスイート
- $F = \{f_1, f_2, \ldots, f_L\}$：$L$ 個の fault f_l の集合
- $Fault\,[n, l]$：テストケース t_n を実行したときに fault f_l によって生じる欠陥が発見できるかどうかを表すブール値

と定義すると，テストケースの n 番目までで発見できる欠陥の原因となる fault の数は

$$\mathcal{F}_n = \#\left(\{f_l \in F | \exists n' \in (1, 2, \ldots, n)\,, Fault\,[n', l]\}\right) \qquad (2)$$

となる．ただし，$\#\,(X)$ は集合 X の要素数である．

APFD は，順序付けしたテストケースに対して，

$$APFD = \frac{\sum_{n=1}^{N-1} \mathcal{F}_n}{N \times \mathcal{F}_N} + \frac{1}{2N} \qquad (3)$$

で与えられる．APFD はすべての fault が最初のテストケースで見つかる場合に $F_1 = F_2 = \cdots = F_N$ となるため最大値 $1 - \frac{1}{2N}$ をとり，すべての fault が最後のテストケースで見つかる場合に $F_1 = F_2 = \cdots = F_{N-1} = 0$ となるため最小値 $\frac{1}{2N}$ をとる．

これから分かるように，APFD は fault を特定するための指標ではなく，実際にテストを実行した際に検出される欠陥から fault が存在することを早期に通知するための指標である．

2.1.2 カバレッジベースの TCP

カバレッジベースの TCP とは，カバレッジ情報を用いてテストケースの優先順位を決める手法のことである [13]．カバレッジ情報とは，一定の単位に分けられたソースコードに対して，各テストケースが実行されたときにそれぞれの単位が実行されるかという情報のことである．カバレッジを分割するユニットとして statement，method，class などが用いられる．カバレッジ情報の取得にはコールグラフが用いられ，静的にコードを分析する静的コードカバレッジと，以前のバージョンのテスト実行時に得られたカバレッジ情報を用いる動的コードカバレッジに大別される [8]．

カバレッジ情報は本来各ユニットが実行されるか否かの 2 値であるが，動的コードカバレッジによって各ユニットを実行した回数を得ることができるため，非負整数に拡張することができる．本研究では拡張したカバレッジ情報を用いる．

2.1.3 優先順位付け戦略

得られたカバレッジ情報からテストケースの優先順位を決める優先付け戦略としてよく用いられるものは total strategy と additional strategy である [11]．total strategy は，カバーするユニットの多いテストケースから順に選択していく戦略であり，その最悪時間計算量はユニットの数を M，テストケースの数を N として $O\,(MN)$ である [11]．また，additional strategy はそれまでに選択した

Algorithm 1 Prioritization in the unified strategy [6]
Require: coverage information $cover\,[n,m]$
Ensure: prioritization $Priority\,[i]$

```
initialize Prob[m] ← 1
initialize Selected[n] ← false
for each i (1 ≤ i ≤ N) do
    for each n (1 ≤ n ≤ N) do
        if not Selected[n] then
            Sum[n] ← Σ_{m=1}^{M} (1 − (1 − c)^{Cover[n,m]}) × Prob[m]
        end if
    end for
    Priority[i] ← arg max_n Sum[n]
    Selected[Priority[i]] ← true
    for each m (1 ≤ m ≤ M) do
        Prob[m] ← Prob[m] × (1 − c)^{Cover[Priority[i],m]}
    end for
end for
```

テストケースがカバーしていないユニットをより多くカバーするテストケースを順に選択していく戦略であり，その最悪時間計算量は $O\left(MN^2\right)$ である [11].

total strategy はユニットの実行回数を最大化するように優先順位付けができる一方，それまでに選択したテストケースのカバレッジ情報がその後の選択に影響しないため，似たようなカバレッジを持つテストケースばかりを実行してしまいトータルカバレッジが向上しない可能性がある．additional strategy は網羅率を最大化するように優先順位付けができる一方，1 度カバーされたユニットについてのカバレッジ情報はその後の選択に影響しないため，1 度欠陥が検出されなかったユニットに欠陥がある場合，検出が遅れる恐れがある．以上の理由から，2 つの戦略のうちどちらが最適かは場合によって異なる．

2.2 優先付け戦略の統合

Hao らは total strategy と additional strategy を統合した戦略として unified strategy を提案した [6]．区間 $(0,1]$ で定義される変数 c の値によって total strategy と additional strategy のどちらに近い戦略かを調整することができるようになっている点が特徴である．コード単位 u_m が未発見の欠陥を含んでいる確率変数 $Prob\,[m]$ を更新し，その重みづけ和によって優先順位の決定を行う．

Hao らが提案した戦略をアルゴリズム 1 に示す．ただし，N 個のテストケース t_n を含むテストスイート $T=\{t_1,t_2,\ldots,t_N\}$, M 個のコード単位 u_m を含むプログラム $P=\{u_1,u_2,\ldots,u_M\}$ に対して，$Cover\,[n,m]$, $Sum\,[n]$, $Selected\,[n]$, $Priority\,[i]$ はそれぞれ

- $Cover\,[n,m]$：テストケース t_n がコード単位 u_m をカバーしている回数を表すカバレッジ情報
- $Sum\,[n]$：テストケース t_n に関する優先順位付けに用いる和
- $Selected\,[n]$：テストケース t_n がすでに選択されたかどうかを表すブール値
- $Priority\,[i]$：i 番目にどのテストケースが順序付けされたかを表す優先度を示す定数

である．

この戦略は $c=1$ のときに additional strategy になり，$c\simeq 0$ のときに total strategy に近づく．$c=0$ のときは $1-(1-c)^{Cover[n,m]}=0$ より，重みづけができないことに注意されたい．$0<c<1$ のときは，c が 0 に近いほど total strategy に，1 に近いほど additional strategy に近い，2 つを複合した戦略となる．また，この戦略の最悪時間計算量は additional strategy と同じ $O\left(MN^2\right)$ である．

3. 提案手法

3.1 確率密度関数による戦略の生成

unified strategy では，$Sum\,[n]$ の計算時に重みづけの確率変数 $Prob\,[m]$ に定数 $(1-c)$ をかけているが，別の更新式の戦略との比較はなされていない．また，対象に応じて c の値をチューニングする必要があるという問題もある．このことから，更新式を変更することでこの戦略を上回る戦略を作ることができる可能性がある．

各コード単位 u_m が実行されたときに欠陥が検出される確率はそれぞれ異なり，p_m と表せるとする．p_m の分布を $f\,(p)$ とおくと，これは欠陥が検出される確率についての確率密度関数である．このとき，x 回カバーされたときに 1 回以上欠陥が検出される確率は

$$\begin{aligned} P\,(x) &= \int_0^1 \left(1-(1-p)^x\right) f\,(p)\,dp \\ &= 1-\int_0^1 (1-p)^x\, f\,(p)\,dp \end{aligned} \tag{4}$$

となる．したがって，あるコード単位 u_m について，これまでに選択したテストケースが合計 $Count\,[m]$ 回，あるテストケース t_n が $Cover\,[n,m]$ 回カバーする t_n によってはじめて u_m の欠陥が検出される確率は

$$P\,(Count\,[m]+Cover\,[n,m])-P\,(Count\,[m]) \tag{5}$$

となるため，t_n によってはじめて検出される欠陥の合計数の期待値

$$\begin{aligned} Sum\,[n] = \sum_{m=1}^{M} &\left(P\,(Count\,[m]+Cover\,[n,m])\right. \\ &\left.-P\,(Count\,[m])\right) \end{aligned} \tag{6}$$

が最大となるテストケースを順に選択することによって既

Algorithm 2 Prioritization in the proposed method
Require: coverage information $cover\,[n,m]$
Ensure: proritization $Priority\,[i]$

```
initialize Count[m] ← 0
initialize Selected[n] ← false
for each i (1 ≤ i ≤ N) do
    for each n (1 ≤ n ≤ N) do
        if not Selected[n] then
            Sum[n] ← Σ_{m=1}^{M} P(Count[m]+Cover[n,m]) − P(Count[m])
        end if
    end for
    Priority[i] ← arg max_n Sum[n]
    Selected[Priority[i]] ← true
    for each m (1 ≤ m ≤ M) do
        Count[m] ← Count[m] + Cover[Priority[i], m]
    end for
end for
```

存手法より良い戦略を生成できると考えられる．

以上をふまえて，提案手法のアルゴリズムをアルゴリズム 2 に示す．この戦略の最悪時間計算量は additional strategy や unified strategy と同じ $O\left(MN^2\right)$ である．

unified strategy [6] の $Prob\,[m]$，$Sum\,[n]$ はそれぞれ $Count\,[m]$ を用いて

$$Prob\,[m] = (1-c)^{Count[m]}, \tag{7}$$

$$\begin{aligned} Sum\,[n] &= \sum_{m=1}^{M} Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]}\right) \\ &= \sum_{m=1}^{M} \left((1-c)^{Count[m]} - (1-c)^{Count[m]+Cover[n,m]}\right) \end{aligned} \tag{8}$$

と表せる．ここで $f\,(p) = \delta\,(p-c)$ とおくと

$$P\,(x) = 1-(1-c)^{x} \tag{9}$$

となるため，unified strategy は提案手法において欠陥の見つかる確率を c に固定した場合と等しいといえる．式 (9) の形で表される確率密度関数が最良であるか否かについては検証する必要がある．そのうえで最適な確率密度関数を予測することが目的になる．

3.2　確率密度関数の推定

3.2.1　最尤推定

$F = f_l \;\; (l = 1, 2, \ldots, L)$ のうちのある fault f_l について，これによって生じる欠陥が検出されるテスト t_n の集合を T_{f_l} とする．コード単位 u_m をカバーした際に f_l による欠陥が検出される確率を $p_{m,l}$ とすると，テスト t_n を実行した際に f_l による欠陥が検出される確率は

$$P_{n,l} = 1 - \prod_{m} (1-p_{m,l})^{Cover[n,m]} \tag{10}$$

となるため，尤度関数は

$$L_l = \prod_{t_n \in T_{f_l}} P_{n,l} \times \prod_{t_n \notin T_{f_l}} (1-P_{n,l}) \tag{11}$$

より，

$$\begin{aligned} &\frac{\partial}{\partial p_{m,l}} \log L_l \\ &= \frac{\partial}{\partial p_{m,l}} \left(\sum_{t_n \in T_{f_l}} \log P_{n,l} + \sum_{t_n \notin T_{f_l}} \log\left(1-P_{n,l}\right) \right) \\ &= \sum_{t_n \in T_{f_l}} \frac{P_{n,l}-1}{P_{n,l}} \frac{Cover\,[n,m]}{p_{m,l}-1} + \sum_{t_n \notin T_{f_l}} \frac{Cover\,[n,m]}{p_{m,l}-1} \end{aligned} \tag{12}$$

となる．これを最大化する $p_{m,l}$ は最急降下法によって最尤推定できる．

推定した $p_{m,l}$ を用いて，確率密度関数は離散的に

$$f\,(p) = \frac{1}{ML} \sum_{m=1}^{M} \sum_{l=1}^{L} \delta\,(p-p_{m,l}) \tag{13}$$

と表現できる．

実験で用いたデータセットから計算した確率密度関数の累積分布を図 **1** に示す．累積分布関数の形状より，確率 $p = 0, 1$ とその付近に分布が集中していることが分かる．また，C オブジェクトと比較して Java オブジェクトの方が分布が $p = 1$ 側に，method カバレッジと比較して statement カバレッジの方が $p = 0, 1$ の両側に寄っている．

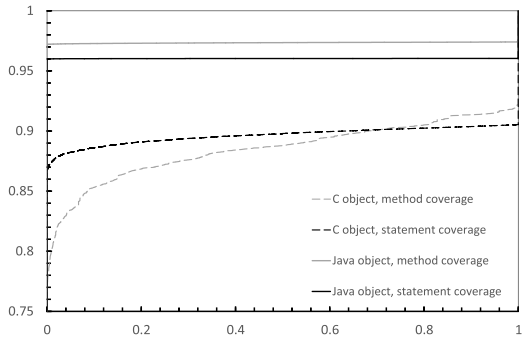

図 1　最尤推定によって計算された累積分布関数
Fig. 1　Cumulative distribution functions calculated by maximum likelihood estimation.

3.2.2　ベータ近似

図 1 の累積分布関数を観察すると，欠陥の検出確率は 0 と 1 に大きく偏って分布していることが分かる．ベータ関数の一部はこの種の関数を表現できることが知られている．ここでは，確率密度関数がベータ関数で近似できると仮定して，その仮定がどの程度妥当かを調べる．

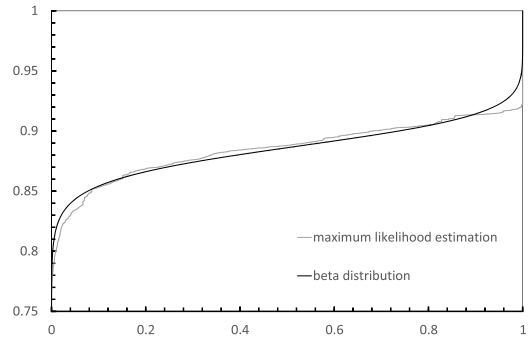

図 2　C 言語，method カバレッジの場合の累積分布関数
Fig. 2　cumulative distribution functions of C programs for test suites with dynamic method coverage.

ベータ分布は範囲 $(0,1)$ で定義され，正の実数 α，β を用いて以下の式で表される．

$$f(p) = \frac{p^{\alpha-1}(1-p)^{\beta-1}}{B(\alpha,\beta)} \tag{14}$$

ただし，$B(\alpha,\beta)$ はベータ関数

$$B(\alpha,\beta) = \int_0^1 p^{\alpha-1}(1-p)^{\beta-1}\,dp \tag{15}$$

であり，ガンマ関数を用いて

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{16}$$

と表される．このとき，x 回の試行で 1 回以上欠陥が見つかる確率 $P(x)$ は

$$\begin{aligned} P(x) &= 1 - \int_0^1 (1-p)^x \frac{p^{\alpha-1}(1-p)^{\beta-1}}{B(\alpha,\beta)}dp \\ &= 1 - \frac{B(\alpha,\beta+x)}{B(\alpha,\beta)} \end{aligned} \tag{17}$$

となる．

ベータ関数の期待値と分散はそれぞれ

$$E[X] = \frac{\alpha}{\alpha+\beta}\ ,\quad var[X] = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} \tag{18}$$

と表せるので，逆に期待値と分散を用いてパラメータ α，β を

$$\begin{aligned} \alpha &= E[X]\left(\frac{E[X](1-E[X])}{var[X]} - 1\right) \\ \beta &= (1-E[X])\left(\frac{E[X](1-E[X])}{var[X]} - 1\right) \end{aligned} \tag{19}$$

と推定することができる．

C 言語オブジェクトの method カバレッジについて，最尤推定によって求めた確率密度関数と，それを近似したベータ分布の累積分布を**図 2** に示す．$\alpha = 0.0175$，$\beta = 0.135$ となった．各関数の形状の比較より，最尤推定で求めた離散的な確率密度関数を連続なベータ分布で近似できていることが分かる．

4. 実験

4.1 実験設定

- Total Strategy
- Additional Strategy
- Unified Strategy の $c = 0.05$ から 0.95 まで，0.05 刻みで 19 通り
- 提案手法による戦略

という複数の戦略について，データセットを用いて APFD を計算した．

データセットとしては，Hao らの公開するもの[*1]を用いた．このデータセットは 40 個の C 言語オブジェクトと 28 個の Java オブジェクトの method，statement 単位の動的カバレッジ情報と fault の情報からなる．C 言語オブジェクトは GNU Core Utilities のバージョン 6.11 から，Java オブジェクトは Software artifact Infrastructure Repository（SIR [*2]）からそれぞれ取得されたものである．

C 言語オブジェクトでは MutGen [1]，Java オブジェクトでは Javalanche [12] もしくは MuJava [9] によって fault を自動生成している．いずれのツールにおいてもソースコードの一部分の定数や演算子を書き換えたり，文を 1 行削除するなどの局所的な変更によって fault を生成している．テストケースは Java オブジェクトについては SIR に公開されているものを用い，C 言語オブジェクトでは KLEE [4] によって自動生成されている．fault を 1 つだけ加えた状態でテストケースを実行し，意図した出力が得られなかったテストケースの集合が fault の情報となる．

提案手法の確率密度関数には，最尤推定で求めた離散分布をベータ近似したものを用いた．最急降下法の学習率には Adam [7] を用いた．提案手法と既存手法のそれぞれについて APFD を計算し，対応のある 2 標本 t 検定によって検証した．

4.2 実験結果と考察

実験結果をそれぞれ**図 3**，**図 4**，**図 5**，**図 6** に示す．図 3 と図 4 は C 言語オブジェクト，図 5 と図 6 は Java オブジェクトに対して，method カバレッジと statement カバレッジを用いて APFD を計算した結果である．横軸は戦略の種類を表しており，Beta は提案手法，Tot は Total Strategy，Add は Addirional strategy，Pro は提案手法の戦略，Uxx は Unified Strategy の $c = \frac{xx}{100}$ の場合を表している．縦軸が APFD の値を表し，箱の上下はそれぞれ第 3 四分位点と第 1 四分位点，中央の線は中央値，上下のひげはそれぞれ 95 パーセンタイルと 5 パーセンタイル，丸印は平均値，上下の×印はそれぞれ最大値と最小値を意味する．また，それぞれの実験において APFD の分布は正規

[*1] https://sites.google.com/site/unifiedtestprioritization
[*2] http://sir.unl.edu/

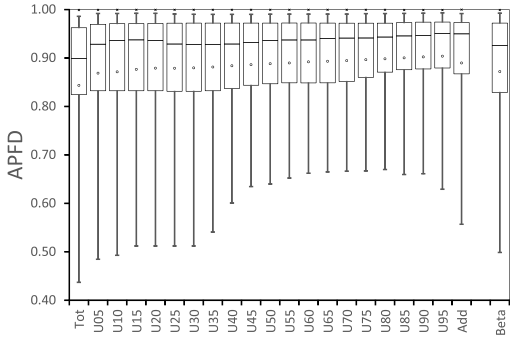

図 3　C 言語，method カバレッジの場合の優先順位付け結果
Fig. 3　Results of C programs for test suites with dynamic method coverage.

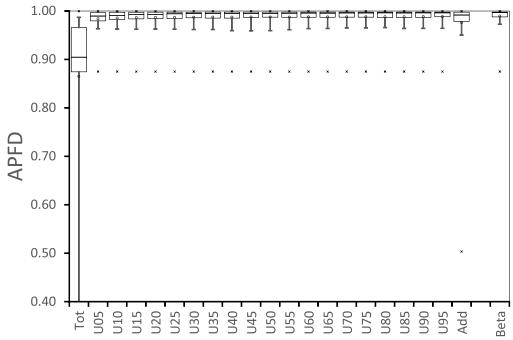

図 4　C 言語，statement カバレッジの場合の優先順位付け結果
Fig. 4　Results of C programs for test suites with dynamic statement coverage.

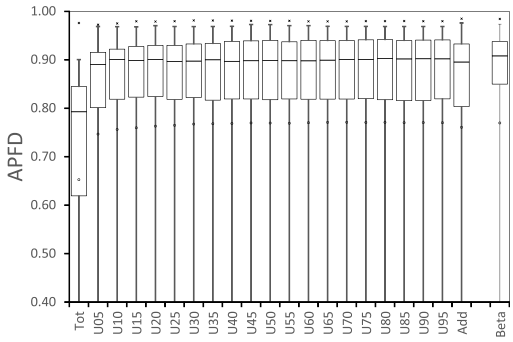

図 5　Java，method カバレッジの場合の優先順位付け結果
Fig. 5　Results of Java programs for test suites with dynamic method coverage.

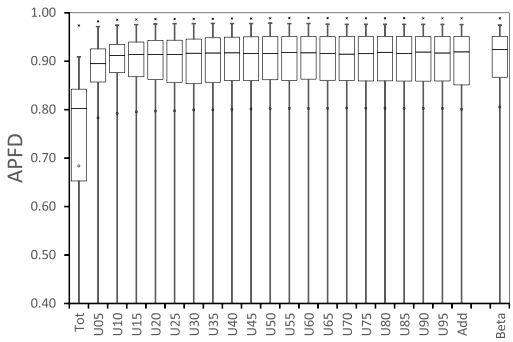

図 6　Java，statement カバレッジの場合の優先順位付け結果
Fig. 6　Results of Java programs for test suites with dynamic statement coverage.

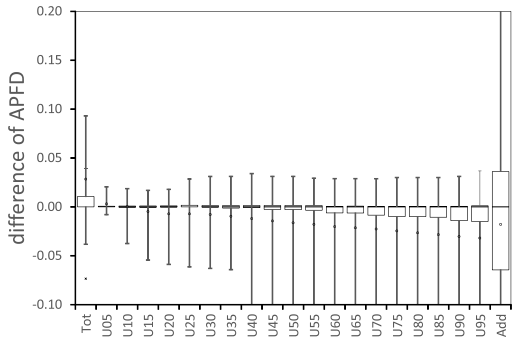

図 7　C 言語，method カバレッジの場合の APFD の差
Fig. 7　Differences of APFD of C programs for test suites with dynamic method coverage.

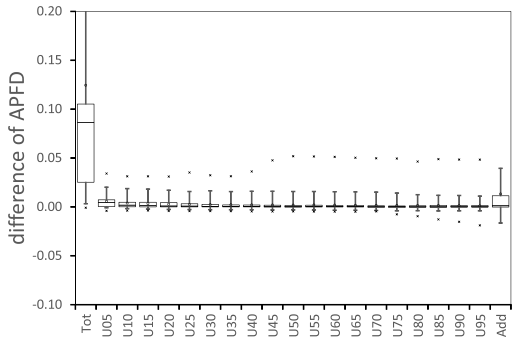

図 8　C 言語，statement カバレッジの場合の APFD の差
Fig. 8　Differences of APFD of C programs for test suites with dynamic statement coverage.

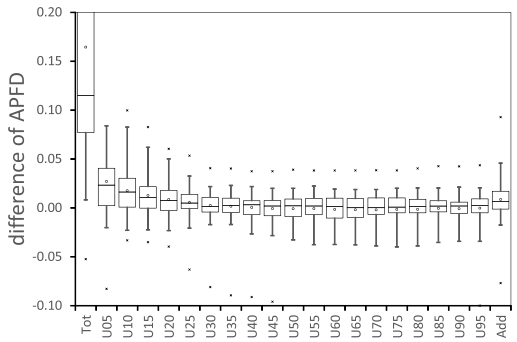

図 9　Java，method カバレッジの場合の APFD の差
Fig. 9　Differences of APFD of Java programs for test suites with dynamic method coverage.

分布に従っているとはいえないが，t 検定の精度に大きく影響を与えるものではないことを確認した．

提案戦略と既存戦略の APFD の差の分布はそれぞれ**図 7**，**図 8**，**図 9**，**図 10** のようになった．図 7 と図 8 は C 言語オブジェクト，図 9 と図 10 は Java オブジェクトに対しての結果である．縦軸はそれぞれのプログラムに対する提案戦略の APFD から各既存戦略の APFD を引いた差を意味する．

t 検定の結果を**表 1** に示す．各オブジェクト，コード単位について，既存手法のそれぞれとの APFD 値の差を母集合とし，提案手法の方が有意に大きいことを “+”，有意

表 1　提案手法と既存手法の t 検定の結果
Table 1　Results of Student's t-test of proposed strategy and existing strategy.

		Tot	U05	U10	U15	U20	U25	U30	U35	U40	U45	U50	U55	U60	U65	U70	U75	U80	U85	U90	U95	Add
C	method	+	+		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
C	statement	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Java	method	+	+	+	+	+	+	+	+													+
Java	statement	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

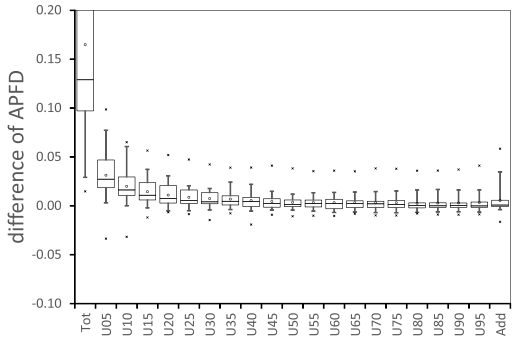

図 10　Java，statement カバレッジの場合の APFD の差
Fig. 10　Differences of APFD of Java programs for test suites with dynamic statement coverage.

に小さいことを "-" で表す．有意差があると判断する基準は，p 値が 0.05 を下回ることとした．

表 1 が示すように，statement カバレッジを用いた実験では，C 言語と Java の両方のオブジェクトについて提案手法が既存手法を有意に上回った．図 8，図 10 を見ると APFD の差の中央値が少なくともほぼ 0 に近い値をとり，分布が正方向により広がっている．このことからも提案手法と既存手法との差分が有意に存在することが読み取れる．以上の結果から，statement カバレッジを用いる場合，提案手法は既存手法と比較して有用であるといえる．一方で，method カバレッジ，C 言語オブジェクトによる実験では $c \geq 0.10$ の場合について既存の戦略を有意に下回った．statement カバレッジと比較して，method カバレッジで提案手法の戦略が有効でなかった理由を考える．

まず，今回の実験で用いたベータ分布は α，β ともに 0 に近く，$\alpha \ll \beta$ が成り立つ．このとき，$P(x)$ は $x = 0$ から 1 になるときに大きく増加し，$x \geq 1$ では緩やかに増加しながら 1 に収束する．つまり，一度もカバーしていないユニットは unified strategy の c が 1 に近い場合に似た働きをし，すでにカバーされたユニットは c が 0 に近い場合に似た働きをする．以上をふまえると，コード単位の粗い method カバレッジの実験では少ないテスト数でユニットのほとんどがカバーされた結果，提案手法の戦略は unified strategy の c が 0 に近い場合に似た結果となったと考えられる．図 3，図 5 の結果もこれを支持している．

5.　関連研究

ソフトウェアテスティングのコストを削減するための手法として，テストケースの最小化，選択，優先順位付けなどがあげられる [14]．このうち，テストケース優先順位付け（TCP）はより少ないテストで欠陥が検出されるようにテストケースを並べ替える手法であり，その多くがカバレッジベースである [13]．基礎的な TCP の戦略として total strategy と additional strategy [11] があり，ほとんどの研究がこれらの重みづけを変更することでこれらを発展させたものである [6]．たとえば Wang らは静的コード解析と欠陥予測モデルによって欠陥が検出される確率が高い部分を推定し [13]，Noor らは類似性に基づく metric などを用いたロジスティック回帰モデルによって [10]，Azizi らは，レコメンドシステムによって過去の欠陥の情報から新たに欠陥が検出される確率が高い部分を推定し [2]，total strategy や additional strategy をベースに重みづけを行っている．

6.　終わりに

本稿では，total strategy，additional strategy，unified strategy に代わる新しい戦略を提案した．これはコード単位を実行した際に欠陥が検出される確率の確率密度関数を用いてテストケースを実行した際に新たに検出される欠陥の期待値を計算し，それが最大となるものから順に選択していくという戦略である．確率密度関数は，カバレッジ情報に最尤推定を適用し，ベータ近似を用いて推定した．C 言語と Java のプログラムに対して提案手法と既存手法の戦略を適用し実験を行った結果，statement カバレッジを用いた場合について，提案手法が既存手法を有意に上回るという結論が得られた．

ベータ関数に従うモデルの構築が可能か，各種機械学習への適用が可能かは今後の課題である．

参考文献

[1] Andrews, J.H., Briand, L.C. and Labiche, Y.: Is Mutation an Appropriate Tool for Testing Experiments?, *Proc. 27th International Conference on Software Engineering* (*ICSE '05*), pp.402–411, ACM (online), DOI: 10.1145/1062455.1062530 (2005).
[2] Azizi, M. and Do, H.: A Collaborative Filtering Recommender System for Test Case Prioritization in Web Applications, *Proc. 33rd Annual ACM Symposium on Applied Computing* (*SAC '18*), pp.1560–1567, ACM (online), DOI: 10.1145/3167132.3167299 (2018).
[3] Bertolino, A.: Software testing research: Achievements,

challenges, dreams, *2007 Future of Software Engineering*, pp.85–103, IEEE Computer Society (online), DOI: 10.1109/FOSE.2007.25 (2007).
[4] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs, *Proc. 8th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '08*), pp.209–224, USENIX Association (2008) (online), available from ⟨https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf⟩.
[5] Elbaum, S., Malishevsky, A.G. and Rothermel, G.: Test case prioritization: A family of empirical studies, *IEEE Trans. Software Engineering*, Vol.28, No.2, pp.159–182 (online), DOI: 10.1109/32.988497 (2002).
[6] Hao, D., Zhang, L., Zhang, L., Rothermel, G. and Mei, H.: A Unified Test Case Prioritization Approach, *ACM Trans. Softw. Eng. Methodol.*, Vol.24, No.2, pp.10:1–10:31 (online), DOI: 10.1145/2685614 (2014).
[7] Kingma, D. and Ba, J.: Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
[8] Luo, Q., Moran, K. and Poshyvanyk, D.: A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques, *Proc. 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2016*), pp.559–570, ACM (online), DOI: 10.1145/2950290.2950344 (2016).
[9] Ma, Y.-S., Offutt, J. and Kwon, Y.R.: MuJava: An automated class mutation system, *Software Testing, Verification and Reliability*, Vol.15, No.2, pp.97–133 (online), DOI: 10.1002/stvr.308 (2005).
[10] Noor, T.B. and Hemmati, H.: Studying Test Case Failure Prediction for Test Case Prioritization, *Proc. 13th International Conference on Predictive Models and Data Analytics in Software Engineering* (*PROMISE '17*), pp.2–11, ACM (online), DOI: 10.1145/3127005.3127006 (2017).
[11] Rothermel, G., Untch, R.H., Chu, C. and Harrold, M.J.: Test case prioritization: An empirical study, *Proc. IEEE International Conference on Software Maintenance* (*ICSM '99*) pp.179–188, IEEE (online), DOI: 10.1109/ICSM.1999.792604 (1999).
[12] Schuler, D. and Zeller, A.: Javalanche: Efficient mutation testing for Java, *Proc. 7th Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.297–298, ACM (online), DOI: 10.1145/1595696.1595750 (2009).
[13] Wang, S., Nam, J. and Tan, L.: QTEP: Quality-aware Test Case Prioritization, *Proc. 2017 11th Joint Meeting on Foundations of Software Engineering* (*ESEC/FSE 2017*), pp.523–534, ACM (online), DOI: 10.1145/3106237.3106258 (2017).
[14] 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 (online), DOI: 10.1002/stvr.430 (2012).

齋藤 雄太

2017 年東京大学工学部電子情報工学科卒業．2019 年同大学工学系研究科電気系工学専攻修士課程修了，修士（工学）．ソフトウェアテスティング，検証に興味を持つ．

佐藤 周行

1985 年東京大学理学部卒業．1990 年同大学大学院理学系研究科修了，理学博士．同年九州大学大型計算機センター講師．現在，東京大学情報基盤センター准教授．セキュリティ，インターネットトラストの研究に従事．