情報処理学会論文誌 Vol.61 No.4 884–894 (Apr. 2020)

設定ファイルを考慮したFault Localizationの拡張

肥後 芳樹[1,a)] 柗本 真佑[1] 内藤 圭吾[1] 谷門 照斗[1] 楠本 真二[1]
切貫 弘之[2] 倉林 利行[2] 丹野 治門[2]

受付日 2019年7月29日, 採録日 2020年1月16日

概要：Fault Localization（以降，FL）とは，バグの原因箇所を推定する手法である．既存手法として，テストケースによる実行経路を用いてFLを行う手法が提案されている．この手法では，失敗テストケースによって実行されるプログラム文はバグの原因箇所である可能性が高く，成功テストケースによって実行されるプログラム文はバグの原因箇所である可能性が低い，というアイデアに基づいて推定を行う．既存手法はテストケースを実行することによりFLを行うため，設定ファイル等の直接実行されない箇所にバグが存在していた場合には，正しくFLできないという課題がある．そこで本研究では，ソースコード以外の直接は実行されない箇所がバグの原因である場合でもより正しくFLできるように既存手法を拡張した．拡張した手法の有効性を確かめるため，Java言語のプロパティファイルを対象とした実装を行った．実験では，既存手法（Javaのソースコードのみを対象としたFL手法）と提案手法（Javaソースコードとプロパティファイルを対象としたFL手法）を比較した．実験の結果，バグの原因箇所がプロパティファイルにある場合には，提案手法は既存手法に比べてより正確にFLができていること，およびバグの原因箇所がソースコードにある場合には，提案手法は既存手法に比べてFLの精度がほとんど変わらないことを確認した．

キーワード：Fault Localization，ソースコード解析，産学連携

An Extended Fault Localization regarding Property Files

Yoshiki Higo[1,a)] Shinsuke Matsumoto[1] Keigo Naitou[1] Akito Tanikado[1] Shinji Kusumoto[1]
Hiroyuki Kirinuki[2] Toshiyuki Kurabayashi[2] Haruto Tanno[2]

Received: July 29, 2019, Accepted: January 16, 2020

Abstract: Fault Localization (in short, FL) is a technique to automatically infer which lines of source code are the root cause of a given bug. Spectrum-based FL is a technique that utilizes results and execution paths of test cases. The basic idea of spectrum-based FL is that, program elements executed by failed test cases are more suspicious while program elements executed by passed test cases are less suspicious. Existing spetrum-based FL techniques look for suspicious code by executing test cases, so that program elements outside the source code are out of scope of the FL targets. However, software systems often include other program elements than the source code such as configuration files and databases. In this research, we try to make an extension of spectrum-based FL techniques regarding property files, which are simple yet often used configuration files in software systems. We also implemented a prototype tool based on the extended FL technique. We conducted an experiment on both industrial systems and open source software systems. As a result, we confirmed that: in the case that a bug is in a property file, our technique outperforms the existing technique; in the case that a bug is in a Java source file, there is little to distinguish our technique from the existing technique.

Keywords: Fault Localization, source code analysis, academic-industrial alliance

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[2] 日本電信電話株式会社
NTT Software Innovation Center, Nippon Telegraph and Telephone Corporation, Minato, Tokyo 108–0023, Japan
[a)] higo@ist.osaka-u.ac.jp

1. はじめに

ソフトウェア開発において，デバッグは多大な労力とコストを必要とする作業である．1デバッグ作業がソフトウェア開発コストの過半数を占めるとの報告もある [4], [7].

NIST[*1]は，米国においてソフトウェアのバグが原因となり，年間約 595 億円の損失が生じていると報告している [12]．このような状況から，デバッグを支援する研究が活発に行われている．

デバッグ支援における研究のトピックとして Fault Localization（以降，FL）がある．FL とは，なんらかの情報を用いることにより，発生したバグの原因箇所を推定する技術である．これまで様々な FL 手法が提案されている [8], [10], [11]．その中で近年最もさかんに研究されている手法の 1 つとして，テストケースによる実行経路の情報を用いて FL を行う Spectrum-Based Fault Localization（以降，SBFL と略す）がある [16]．SBFL は，失敗テストケースで実行されたプログラム文はバグの原因箇所である可能性が高く，成功テストケースで実行されたプログラム文はバグの原因箇所である可能性が低い，というアイデアに基づいて FL を行う．具体的には，各プログラム文が失敗テストケース，および成功テストケースで何回実行されたかという情報を用いて，そのプログラム文がバグの原因箇所である可能性（以下，疑惑値）を計算する．これまでに多くの SBFL 手法が提案されている [5], [6], [9]．

しかし，SBFL はテストケースによる実行経路に基づきバグの原因箇所の推定を行うため，テストケースによって直接実行されるプログラム文のみが疑惑値の付与対象である．そのため，設定ファイル等にバグが含まれていた場合，SBFL では正しく FL することができない．そこで本研究では，既存手法を拡張することで，ソースコード以外のファイルに対する疑惑値の計算を試みる．提案手法の評価実験として，2 つ企業のシステムと 2 つのオープンソースソフトウェアに対して提案手法を適用した．その結果，3 つのバグについて提案手法は既存手法より正確にバグの原因箇所を推定することに成功した．また，既存手法と比較して提案手法の実行時間は最大で 4.5%の増加であった．

2. 研究動機

本研究を行うに先立って，バグ修正においてどのようなファイルが変更されるのかを調査した．調査対象は，ある企業の 2 つのシステム開発プロジェクトにおける 1,245 のコミット，および 100 個のオープンソースソフトウェア[*2]（以降，OSS）開発プロジェクトにおける約 830,000 のコミットである．いずれのプロジェクトもソースコードは Java で記述されている．

それらのコミットから，まずバグ修正コミットを特定し，そして特定されたコミットにおいて変更されているファイルの拡張子を調査した．バグ修正コミットの特定は，企業システムと OSS では異なる方法で行った．

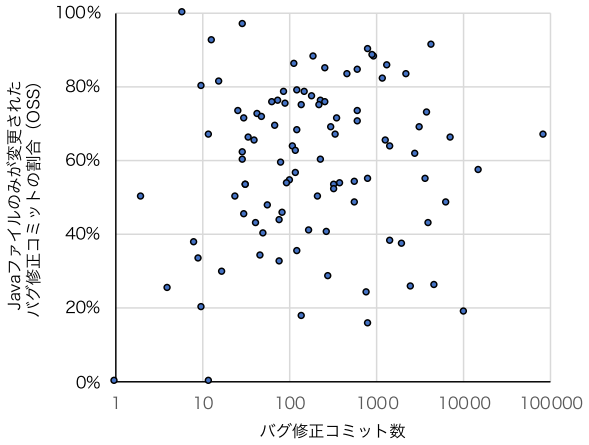


図 1 OSS プロジェクトにおいて，Java ファイルのみが変更されたバグ修正コミットの割合

Fig. 1 Ratio of commits where only Java files are changed on each target project.

企業システム 企業から提供されたシステムのバグは，バグ票によって管理されていた．そこで，バグ票に記載されたバグ ID と紐づけられているコミットをバグ修正コミットと見なした．

OSS コミットメッセージに “fix” もしくは “repair” を含むコミットをバグ修正コミットと見なした．

企業システムでは，合計で 91 個（79 個＋12 個）のバグ修正コミットが存在しており，そのうちの 46 個（39 個＋7 個）が Java ファイルのみを修正しているバグであった．OSS については結果を**図 1** に示す[*3]．この図より，プロジェクトに存在するバグ修正コミット数の多寡にかかわらず，Java ファイル以外のファイルが修正されているバグ修正コミットが存在していることが分かる．これら 100 の OSS において，バグ修正コミットに占める Java ファイルのみが修正されたコミットの割合の中央値は 62.1%であった．これら以外のコミットにおいて修正されたバグについては，既存手法ではバグの原因箇所に疑惑値を付与できない．そのため，既存手法を利用した FL では不十分な場合がある．たとえば，図 5 のように，メソッドの引数で与えられた変数の値に応じて読み込むプロパティファイルのキーが異なる場合は，ソースコードのキーを読み込む箇所（図 5 の 6 行目）を見ても，どのキーを読み込んでいるか分からないためソースコードのみを対象とした FL では不十分である．

次に，バグ修正コミットで修正された Java ファイル以外のファイルについて，どの種類のファイルが修正されているかを調査した．企業システムでは，最も変更されたファイルはプロパティファイル[*4]であり，バグ修正コミットで

[*1] National Institute of Standards and Technology, アメリカ国立標準技術研究所

[*2] GitHub で “Java” のラベルがついたプロジェクトのうち，2018 年 11 月 8 日時点でスター数が上位 100 件のプロジェクト．

[*3] 対象とした 100 プロジェクトのうち，1 つのプロジェクトからはバグ修正コミットが特定されなかったため，図 1 は残りの 99 のプロジェクトに対する結果である．

[*4] 本論文では，拡張子が.properties のファイルをプロパティファイルと呼ぶ．

```
…
NOT_DECREASING_NUMBER_OF_POINTS = les points {0} et {1} ne sont pas d¥u00e9croissants ({2} < {3})
NOT_DECREASING_SEQUENCE = les points {3} et {2} ne sont pas d¥u00e9croissants ({1} < {0})
NOT_ENOUGH_DATA_FOR_NUMBER_OF_PREDICTORS = pas assez de donn¥u00e9es ({0} lignes) pour {1} pr¥u00e9dicteurs
- NOT_ENOUGH_POINTS_IN_SPLINE_PARTITION = une partition spline n¥u00e9cessite au moins {0} points, seuls {1} ont ¥u00e9t¥u00e9 fournis
+ NOT_ENOUGH_POINTS_IN_SPLINE_PARTITION = une partition spline n¥u00e9cessite au moins {1} points, seuls {0} ont ¥u00e9t¥u00e9 fournis
NOT_INCREASING_NUMBER_OF_POINTS = les points {0} et {1} ne sont pas croissants ({2} > {3})
NOT_INCREASING_SEQUENCE = les points {3} et {2} ne sont pas croissants ({1} > {0})
NOT_MULTIPLICATION_COMPATIBLE_MATRICES = les dimensions {0}x{1} et {2}x{3} sont incompatibles pour la multiplication matricielle
…
```

図 2 プロパティファイルの一例
Fig. 2 An example of property file.

表 1 OSS におけるバグ修正コミットで変更されたファイル（プロジェクト数が多い上位 10 種類のみ）
Table 1 Kinds of files that were changed in bug-fix commits (Only top 10 are shown due to space limination).

拡張子	プロジェクト数	総変更回数（割合）
xml	90	34,923 (12.39%)
拡張子なし	87	22,771 (8.08%)
md	80	8,688 (3.08%)
properties	71	8,674 (3.08%)
gradle	65	4,740 (1.68%)
gitignore	56	515 (0.18%)
png	52	23,701 (8.41%)
yml	51	1,963 (0.70%)
html	44	44,179 (15.67%)
txt	44	14,715 (5.22%)

26 回変更されていた[*5]．2 番目に多く変更されたファイルは XML ファイルであり，その回数は 10 回であった．

表 1 は，100 の OSS プロジェクトに対する調査結果を表す．約 700 種類のファイルがバグ修正コミットにおいて変更されていた．スペースの都合上，表 1 はバグ修正コミットにおいて変更したプロジェクト数が上位 10 件となっているファイルの拡張子のみを示している．この表において，第 2 カラムはその拡張子のファイルを一度でもバグ修正コミットにおいて修正したプロジェクト数を表す．第 3 カラムは，その拡張子のファイルがバグ修正コミットにおいて変更された回数の 100 プロジェクトにおける合計値を表す．また，括弧内の数字は，Java ファイル以外のすべてのファイルがバグ修正コミットで変更された回数のうち，その拡張子のファイルが変更された割合を示す．たとえば，xml ファイルは 90 の OSS プロジェクトにおいてバグ修正コミットで変更されており，それらの変更回数の合計値は 34,923 回である．そしてこの変更回数は，Java 以外の全ファイルの変更回数のうち，12.39%を占める．

本研究では，SBFL 手法のソースコード以外への拡張の第 1 歩として，企業システムで最も多く変更されていたプロパティファイルを対象とする．プロパティファイルはプログラムの設定ファイルとして用いられることが多い．プロパティファイルの例を**図 2** に示す．このプロパティファイルは，Apache Commons Math という OSS において，フランス語の単語や画面に表示するメッセージを格納している．このプロパティファイルは，Apache Commons Math のエラーメッセージの仕様変更にともない，修正が行われた．このようにプロパティファイルは，各行が `key = value` の形式で記述され，その構造は単純である．

3. プログラムの実行経路情報を利用した Fault Localization

バグの原因箇所を類推する FL 手法が研究されており，そのなかの一手法としてプログラムの実行経路情報を利用した FL 手法[*6]がある [16]．SBFL 手法を利用するためにはテストケースが必要である．テストケースを利用して，対象プログラムを実行し実行経路情報を収集する．プログラムの実行経路情報とは各テストケースにおいて対象プログラムの各文が実行されたか否かを表す情報である．実行経路情報と各テストケースの成否情報を利用し，対象プログラムの各文に対してバグの原因箇所である可能性を表す数値[*7]を付与する．なお，テストが成功するとはそのテストにおいて実行された対象プログラムの計算結果が期待する値と同じになることであり，テストが失敗するとはそのテストにおいて実行された対象プログラムの計算結果が期待する値とは異なる値となることである．SBFL 手法の基本的な考え方は，失敗テストケースで実行されたプログラム文はバグが存在する可能性が高く，成功テストケースで実行されたプログラム文は正しい可能性が高い，である．

Ochiai と呼ばれている疑惑値の計算方法がある [3]．式 (1) は Ochiai の計算式を表す．なお，s は疑惑値の計算対象の文である．

$$suspicious(s) = \frac{fail(s)}{\sqrt{totalFail * (fail(s) + pass(s))}} \quad (1)$$

式中の変数の意味を以下に示す．
$totalFail$：失敗テストケースの総数
$fail(s)$：文 s を実行する失敗テストケースの数
$pass(s)$：文 s を実行する成功テストケースの数

[*5] 各バグ修正コミットにおいて変更されたファイル数をカウントし，すべてのバグ修正コミットにおける変更回数をその合計値として算出した．
[*6] Spetrum-Based Fault Localization．以降，SBFL 手法と呼ぶ．
[*7] 以降，疑惑値と呼ぶ．疑惑値は一般的に 0〜1 の値をとる．

Ochiai の式は，その文を実行する成功テストが多いほど疑惑値は低くなり，その文を実行する失敗テストが多いほど疑惑値が高くなる．

Ochiai は SBFL 手法の他の計算式よりも FL の性能が優れていることが示されている [1], [2]．本研究でも FL に Ochiai を用いる．

4. 提案手法

本研究では，既存の FL 手法を拡張し，プロパティファイルの各キーに対しても疑惑値の計算を行う手法を提案する．提案手法の入力は Java プロジェクトとテストケースである．提案手法の出力は，対象プロジェクトに含まれる各プログラム文およびプロパティファイルに含まれる各キーに対する疑惑値である．

提案手法は，SBFL の疑惑値を計算する式を Java ソースコードとプロパティファイルに適用することで，それらのどこにバグの原因箇所があるのかを推測する．疑惑値の計算には Ochiai [3] の計算式を用いた．プロパティファイルのキー k の疑惑値 $suspicious(k)$ を計算する際，以下に示すように Ochiai の計算式を応用した．

$$suspicious(k) = \frac{fail(k)}{\sqrt{totalFail * (fail(k) + pass(k))}}$$

式中の変数の意味を以下に示す．

$totalFail$：失敗テストケースの総数

$fail(k)$：キー k を読み込む失敗テストケースの数

$pass(k)$：キー k を読み込む成功テストケースの数

提案手法では，Ochiai を利用しているが，SBFL 手法において利用される他の式（たとえば，Zolter や Tarantula 等）を利用することもできる．他の式を利用してもプロパティファイルのキーに疑惑値を付与できるため，既存のソースコードのみを対象とした FL に比べてプロパティファイルのバグを正しく FL できる．

4.1 提案手法の利用場面

開発者が FL の結果を利用する場面においては，プロパティファイルのキーが静的に与えられている場合（ソースコード中に文字列として明記されている場合）は，既存手法を利用した場合に比べて提案手法の優位性は高くない．そのような場合は，既存手法を利用してソースコード中においてプロパティファイルのキーを読み込んでいる行をバグ原因箇所と見なすことで，十分にプロパティファイルの原因箇所を特定できるからである．しかし，プロパティファイルの読み込むキーが静的には与えられていない場合（つまりキー文字列がプログラムの実行中に決定されるためソースコードを読むだけではどのキーが指定されているのか分かりづらい場合）は，ソースコード中においてキーを読み込んでいる箇所がバグの可能性が高いとなった場合でも，プロパティファイルのどのキーが指定されているのか自明ではないため，提案手法は既存手法に比べて有効であると著者らは考える．また，自動プログラム修正おいて FL の結果を利用する場面では，プロパティファイルのキーが静的に与えられる場合であっても提案手法は既存手法に対して優位性があると著者らは考える．その理由は，プロパティファイルの読み込まれる行に高い疑惑値が付与されりうことにより，自動プログラム修正においてその行が変更対象となるからである．

4.2 実装

提案手法を実現するためには，以下の項目を行う機能が必要である．

機能 A 各テストの実行において，プロパティファイルの各キーの読込の有無を記録する機能．

機能 B 各テストの実行において，ソースコードの各プログラム文の実行の有無を記録する機能．

機能 C 与えられたテスト群を順次実行する機能．

機能 D 各キーの読込の有無および各プログラム文の実行の有無の情報から疑惑値を計算する機能．

以下，本節では，著者らが実装したツールにおいて上記の各機能の実現方法について述べる．

機能 A

Java では，プロパティファイルの読み込みには，`Properties` クラス[*8]，もしくは `ResourceBundle` クラス[*9]が利用されることが多い．しかし，これらのライブラリにはテスト実行時にどのキーが何回読み込まれたか記録する機能はない．そこで，これらのライブラリクラスを継承したクラスを作成し，そのクラスで各キーの読み込みの回数を記録する機能を実装した．

機能 B

各プログラム文の実行回数や，各キーの読み込み回数を計測するために，対象プロジェクトへコードの埋め込みを行う．しかし，提案手法を適用するたびに対象プロジェクトにコードを手動で埋め込むのは現実的ではない．提案手法では，対象プロジェクトの抽象構文木を構築し，その抽象構文木に対して自動で加工を行う．加工は以下の 2 つの種類がある．

(1) 各プログラム文の後に，その文が実行されたことを記録するプログラム文を追加する．

(2) プロパティファイルを読み込むライブラリのインスタンスを，本手法で拡張したクラスのインスタンスに換える．

機能 C

提案手法では，各テストにおいてプロパティファイルの各キーの読込の有無の情報が必要である．しかし，複数のテストが存在する場合にそれらで利用する情報を一度にま

[*8] `java.lang.Properties`
[*9] `java.util.ResourceBundle`

とめて取得し（プロパティファイルから読み込み），そのあとで各テストを順次実行するというテストの実装が行われることがある．そのような場合は，プロパティファイルから取得した情報は静的なフィールド変数[*10]に格納される．しかし，それでは静的なフィールドの初期化で読み込まれたキーの疑惑値が正しく計算されない．そこで，提案手法では各テストメソッドを別プロセスで実行する．このように変更することで，各テストケースの実行前に静的なフィールド変数に対してプロパティファイルの読み込みが行われるため，キーの疑惑値を計算できる．

機能 D

提案手法では 2 つの方法で疑惑値の計算を行う．1 つ目は，各プログラム文に対する疑惑値の計算である．疑惑値の計算には，SBFL の 1 つ，Ochiai の計算式を用いた．2 つ目は，プロパティファイルの各キーに対する疑惑値の計算である．プロパティファイルの各キーに対する疑惑値の計算には本章の冒頭で述べた，Ochiai の計算式を変更した計算式を用いた．

5. 実験

本章では，提案手法を評価するために行った実験と，その結果について述べる．

5.1 準備

実験対象とするバグの収集を行った．この実験では，2 つの企業システムのバグと，2 つの OSS（Apache Commons Math と Closure Compiler）から，対象となるバグを収集した．合計で 13 のバグを収集した．企業システムのバグは，2 章で特定したバグ修正コミットから，対象バグを収集した．Apache Commons Math と Closure Compiler は，数多くの研究で利用されているバグのデータセット Defects4J [*11]のバグ収集対象であるプロジェクトである．本研究では，プロパティファイルに含まれているバグを実験対象とする．Defects4J に含まれているのは Java ファイルが原因箇所であるバグであるため，それらのバグは利用できない．しかしながら，Defects4J のバグ収集対象になっている OSS プロジェクトはテストケースを多く含むプロジェクトであるため，これらのプロジェクトから著者らがバグを収集した．企業システムと 2 つの OSS は，どちらも Java 言語で記述されている以外の関連性はない．

以下の条件を満たすバグが今回の実験対象である．

- そのバグが修正されたコミットが特定できる．
- Java ファイル（拡張子が .java）の単一行，もしくはプロパティファイル（拡張子が .properties）の単一行のみが修正されている．

これらの条件を満たすプロパティファイルのバグは 4 つ，Java ファイルのバグは 5 つ存在した．提案手法の有効性を確認するためのプロパティファイルのバグが 4 つでは少ないと考え，OSS からもプロパティファイルのバグを収集した．Apache Commons Math と Closure Compiler からのバグ収集手順は以下のとおりである．

(1) コミットを解析し，“fix” もしくは “repair” がコミットメッセージに含まれるコミットをバグ修正コミットとして収集した．
(2) それらのコミットからプロパティファイルの単一行のみが修正されているコミットを抽出した．
(3) 抽出したコミットについて，コミットメッセージおよび修正内容を目視確認し，バグ修正であると著者らが判断できた場合に実験対象のバグとした．

表 2 は収集したバグの情報を表している．第 2 カラムはバグの原因箇所がプロパティファイルか Java ソースファイルかを表している．第 3 および第 4 カラムは FL 対象のソースコード行数およびプロパティファイル行数を表している．単体テストのテストケースを利用して FL を行うため，各バグで FL 対象となるソースコード行数やプロパティファイル行数が異なる．第 5 カラムはそのバグが企業システムのものか OSS のものかを表している．

表 2 対象バグ
Table 2 Target bugs.

バグ ID	バグ箇所	行数		企業/OSS
		Java	プロパティ	
#1	プロパティ	5,848	3	OSS
#2	プロパティ	47	1	OSS
#3	プロパティ	47	1	OSS
#4	プロパティ	53	1	OSS
#5	プロパティ	1,845	707	企業
#6	プロパティ	130	50	企業
#7	プロパティ	73	7	企業
#8	プロパティ	243	56	企業
#9	Java	267	56	企業
#10	Java	305	63	企業
#11	Java	202	55	企業
#12	Java	326	59	企業
#13	Java	115	13	企業

5.2 手順

実験は以下の手順で行った．

(1) バグ修正コミットの 1 つ前のコミットをチェックアウトする．
(2) そのコミットでテストケースを実行する．
(3) バグが原因となる失敗テストケースが含まれない場合，著者らがそのような失敗テストケースを作成する．
(4) 対象プロジェクトに対して，提案手法と既存手法を実行する．

手順 (1) でバグ修正コミットの 1 つ前のコミットを

[*10] `static` キーワードが付与されたフィールド変数
[*11] https://github.com/rjust/defects4J

チェックアウトしたのは，バグを含んだ状態を用意するためである．企業のシステムのバグについて，手順 (3) で作成したテストケースは，企業の開発者にそのテストケースの正しさを確認していただいた．

評価指標として，2 つの指標を用いた．1 つ目は，バグの存在する箇所の疑惑値の順位である．より高い順位にバグ原因箇所が順位付けされている手法が，より FL の精度が高いと評価する．既存手法では プロパティファイルのキーに含まれるバグは見つけることができない．そのため，既存手法の評価では，バグの原因であるプロパティファイルのキーを読み込んでいるプログラム文をバグ原因箇所として評価を行った．この指標は，疑惑値の高いプログラム文から手作業により調べていった場合，バグ原因箇所を見つけるために調査しなければならないプログラム文の数およびキーの数を表している．そこで，複数のプログラム文やキーが同じ疑惑値を持つ場合，それらの順位は最も悪くなるように評価する．たとえば，2 つのキーが 1 位に順位づけられていた場合，それらの順位は 2 位として評価する．

2 つ目の指標は実行時間である．既存手法と提案手法を，それぞれのバグに対し 5 回ずつ実行し，その実行時間の平均を比較した．これにより，提案手法の導入によりどの程度疑惑値の算出に長い時間が必要となるのかを評価する．

5.3 結果

バグ原因箇所の精度および順位に関する結果を**表 3** に示す．ここで精度とは，バグの原因箇所が FL 結果の上位（何%の位置に存在したのか）を表す．なお，既存手法では FL 対象は Java ソースコードのみなのに対して提案手法では Java ソースコードおよびプロパティファイルであるため，FL の精度を出す際に利用する全体行数の値が異なる．つまり既存手法では表 2 の第 3 カラムの値のみを用いるが，提案手法は第 3 カラムと第 4 カラムの合計値を用いる．

表 3 (a) に含まれるバグは，プロパティファイルが原因箇所であるバグである．実験の結果，3 件のバグ（#6，#7，#8）について，提案手法により順位が改善した．提案手法によって最も順位が改善したもので，34%の改善が見られた．一方で，提案手法によって最も順位が悪化したバグで，1.3%の悪化が見られた．この理由は，プロパティファイルのバグとは関係のないキーがバグの原因箇所であるキーと同じ疑惑値を持ったためである．提案手法を利用することによりバグとは関係のないソースコード内のプログラム文の疑惑値が，既存手法を利用した場合に比べて高くなることはなかった．

次に，バグ原因箇所が Java ファイルであるバグに対する実験の結果を表 3 (b) に示す．表 3 (b) より，いずれのバグについても，順位の大きな違いは見られなかった．#10 や #13 のバグでは若干ではあるが，提案手法は既存手法に比べてうまく FL できなかった．これは，プロパティファイルを読み込む周辺の Java コードにバグがあったことにより，プロパティファイルにおいて読み込まれたキーについても高い疑惑値が付与されてしまったことが原因である．

表 3 提案手法および既存手法を用いた FL の結果
Table 3 The results of fault localization with the extended technique or the existing one.

(a) プロパティファイルのバグに対する結果

バグ ID	提案手法		既存手法	
	上位%	順位	上位%	順位
#1	0.034%	2	0.017%	1
#2	58.33%	28	57.45%	27
#3	58.33%	28	57.45%	27
#4	29.63%	16	28.30%	15
#5	0.078%	2	0.039%	1
#6	65.56%	118	100.0%	130
#7	5.00%	4	21.92%	16
#8	1.320%	4	3.239&	8

(b)Java ソースコードのバグに対する結果

バグ ID	提案手法		既存手法	
	上位%	順位	上位%	順位
#9	1.548%	5	1.498%	4
#10	18.75%	69	19.02%	58
#11	8.171%	21	8.911%	18
#12	8.571%	33	10.12%	33
#13	35.94%	46	35.65%	46

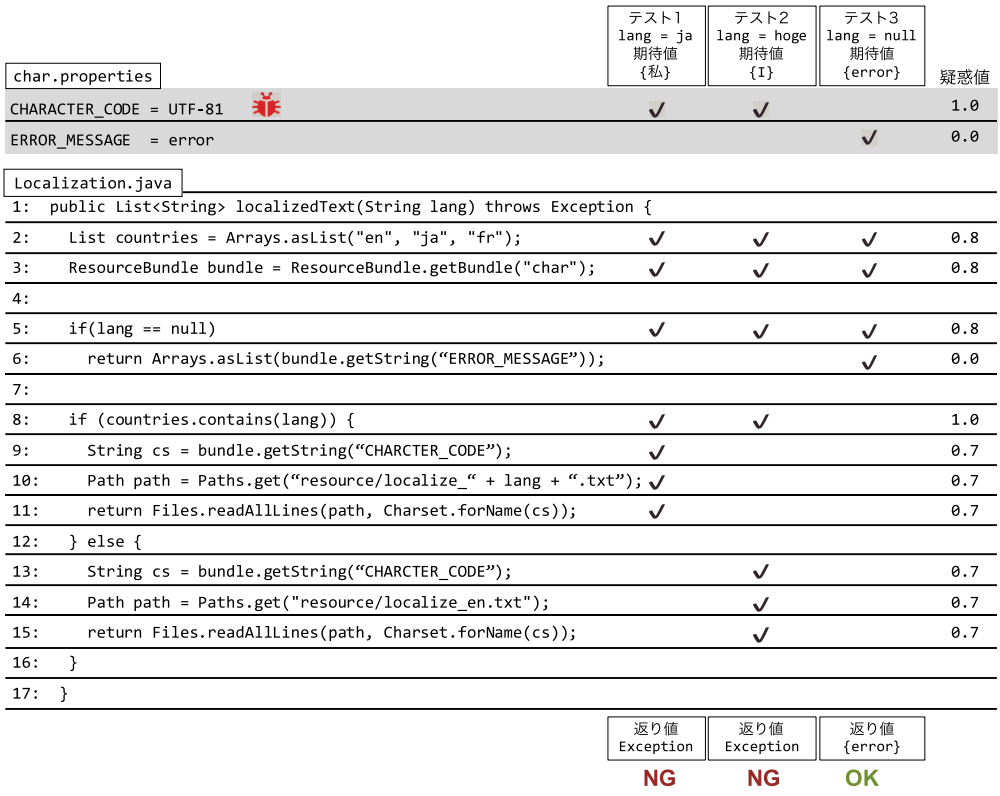
表 4 提案手法および既存手法の実行時間（秒）
Table 4 Execution time of the extended and existing techniques (sec.).

バグ ID	提案手法	既存手法
#1	1,914.2	1,905.7
#2	15.89	15.61
#3	15.99	15.28
#4	9.07	8.82
#5	12.14	11.89
#6	523.0	519.1
#7	222.9	222.3
#8	553.8	545.8

次に，実行時間についての結果を**表 4** に示す．いずれのバグについても，提案手法と既存手法で実行時間に大きな差は見られなかった．提案手法による時間の増加が最も大きかったバグで，その増加量は約 8.5 秒であった．実行時間が長くなっている割合は最大で約 4.5%，最小で 0.3%と少ない．つまり，プロパティファイルに対する疑惑値の付与に必要な時間は全体の実行時間のうちの小さい割合であることが分かる．

6. 考察

本章では，提案手法により順位が改善したバグおよび順位が悪化したバグについて考察を述べる．なお，説明を簡



char.properties

	テスト1 lang = ja 期待値 {私}	テスト2 lang = hoge 期待値 {I}	テスト3 lang = null 期待値 {error}	疑惑値
CHARACTER_CODE = UTF-81	✓	✓		1.0
ERROR_MESSAGE = error			✓	0.0

Localization.java

1: public List<String> localizedText(String lang) throws Exception {				
2: List countries = Arrays.asList("en", "ja", "fr");	✓	✓	✓	0.8
3: ResourceBundle bundle = ResourceBundle.getBundle("char");	✓	✓	✓	0.8
4:				
5: if(lang == null)	✓	✓	✓	0.8
6: return Arrays.asList(bundle.getString(“ERROR_MESSAGE”));			✓	0.0
7:				
8: if (countries.contains(lang)) {	✓	✓		1.0
9: String cs = bundle.getString(“CHARCTER_CODE”);	✓			0.7
10: Path path = Paths.get(“resource/localize_“ + lang + “.txt”);	✓			0.7
11: return Files.readAllLines(path, Charset.forName(cs));	✓			0.7
12: } else {				
13: String cs = bundle.getString(“CHARCTER_CODE”);		✓		0.7
14: Path path = Paths.get("resource/localize_en.txt");		✓		0.7
15: return Files.readAllLines(path, Charset.forName(cs));		✓		0.7
16: }				
17: }				
	返り値 Exception	返り値 Exception	返り値 {error}	
	NG	NG	OK	

(a) 対象のソースコード

localize_en.txt

```
1: I
```

localize_ja.txt

```
1: 私
```

localize_fr.txt

```
1: je
```

(b) 入力ファイル

図 3 提案手法により順位が改善した例

Fig. 3 A simplified example where the extended technique worked well.

単化するため，実際のバグそのものを用いて考察するのではなく，実際のバグの特徴から作成した疑似バグを用いて説明を行う．

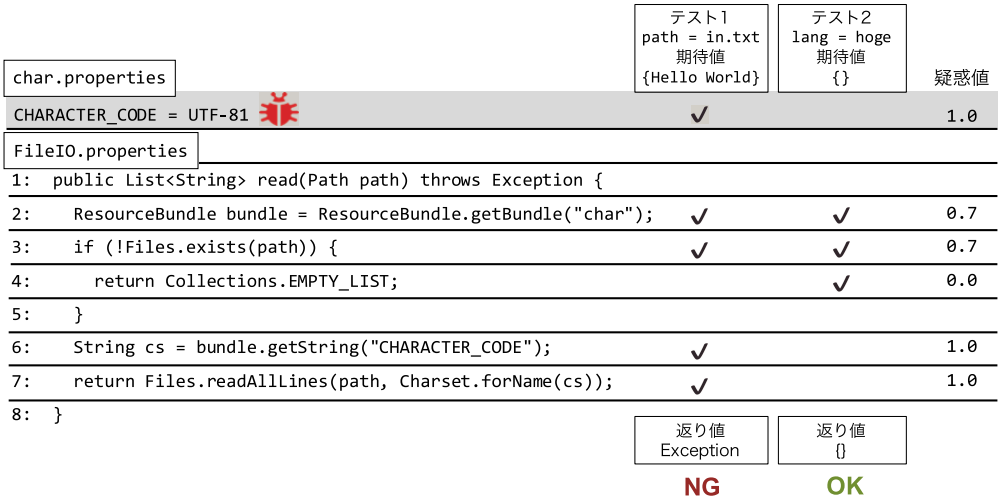
6.1 提案手法により順位が改善したバグ

提案手法により順位が改善した例を**図 3** に示す．図 3 (a) のメソッドは，引数に言語の略称を取り，その言語に対応したファイルの中身を返すメソッドである．各言語に対応するファイルには，図 3 (b) に示したように，その言語における “私” を意味する単語が格納されている．この例には，“char.properties” に含まれているキー，“CHARACTER_CODE” が本来は “UTF-8” であるべきだが，“UTF-81” になっている，というバグが含まれている．そのため，“CHARACTER_CODE” を用いて，入力ファイルを読もうとするテスト 1 とテスト 2 が失敗している．テスト 1 では 9 行目で，テスト 2 では 13 行目で “CHARACTER_CODE” が読み込まれている．

式 (1) より，失敗テストケースのみで実行されたプログラム文であっても，そのプログラム文を実行しない失敗テストケースも存在していると疑惑値が低下することが分かる．そのため，“CHARACTER_CODE” を読み込んでいる 9 行目および 13 行目の疑惑値は，テスト 1 およびテスト 2 で実行される 8 行目よりも低くなってしまう．その一方で，提案手法では，9 行目および 13 行目のどちらでも “CHARACTER_CODE” を読み込んでいるため，バグを含むキーが最も高い疑惑値となる．このように，バグの原因箇所となっているプロパティファイルのキーが複数箇所で読み込まれていた場合，提案手法により順位の改善が見られた．

6.2 提案手法により順位が悪化したバグ

提案手法により順位が悪化した例を**図 4** に示す．図 4 (a)



(a) 対象のソースコード

(b) 入力ファイル

図 4 提案手法により順位が悪化した例
Fig. 4 A simplified example where the extended technique did not work well.

のメソッドは，Path を引数にとり，そのパスのファイルが存在する場合は，そのファイルの中身を返し，存在しない場合は，空のリストを返すメソッドである．図 4 (a) のテスト 1 で読み込まれている，“in.txt” の中身を図 4 (b) に示す．この例も，6.1 節と同様に，“char.properties” の “CHARACTER_CODE” がバグ原因箇所である．この例では，“CHARACTER_CODE” を読み込んでいるプログラム文は 6 行目のみである．そのため，“CHARACTER_CODE” が読み込まれる回数と，“CHARACTER_CODE” を読み込むプログラム文の実行回数が一致する．したがって，“CHARACTER_CODE” と，“CHARACTER_CODE” を読み込むプログラム文が同じ疑惑値，順位になる．5.2 節で述べたように，同一順位に複数順位付けされていた場合，その順位は最も悪くなるように評価する．そのため図 4 のような例では順位が悪化した．

しかし，順位が悪化したにもかかわらず，提案手法が有用であると考えられる例もあった．その例を**図 5** に示す．

図 5 のメソッドは，引数に国名をとり，その国における “こんにちは” を意味する単語を返すメソッドである．この例では，キー “USA” が本来は “Hello” であるべきところが，“Hell” になっている，というバグが含まれている．キー “USA” は，6 行目だけで読み込まれている．そのため，前述の理由でキー “USA” と，6 行目の疑惑値，順位が一致している．しかし，6 行目は，引数で与えられた文字列をキーとしてプロパティファイルを読み込んでいるため，6 行目がバグ原因箇所と推定された場合でも，プロパティファイルのどのキーの値がバグ原因箇所であるか開発者はわからない．提案手法では，“USA” というキーをバグ原因箇所として推定するため，開発者はバグ原因箇所がどこであるか明確に分かる．このように，プロパティファイルを読み込むキーが変数であった場合，提案手法により順位が悪化しても，提案手法が有用であると考えられる．

6.3 実行時間

表 4 で示したように，既存手法と比較して提案手法の時間の増加は十分に小さなものであった．提案手法の実行時間をさらに短縮する方法について考察を行っていく．最もオーバヘッドが大きかったバグ#3 を 5 回実行して，提案手法の各工程が占める実行時間を調査した．**表 5** にその結果を示す．

表 5 より，提案手法の実行時間の 85%はビルドとテストによって占められていることが分かる．また，今回の提案手法で追加された工程である，ソースコードの加工と，プロパティファイルの FL は合計で 6%であった．よって，今後提案手法の実行時間を短くするためには，提案手法で追加された工程ではなく，ビルドとテストの短縮の方が効果的であると考えられる．テスト実行時間の短縮方法として，バイトコードの最適化があげられる．バイトコードの最適化の研究として Raja による SOOT [15] や，Pominville によるもの [13] がある．こういったツールを用いてバイトコードの最適化を行うことで，提案手法のテスト実行時間を短縮できると考えられる．

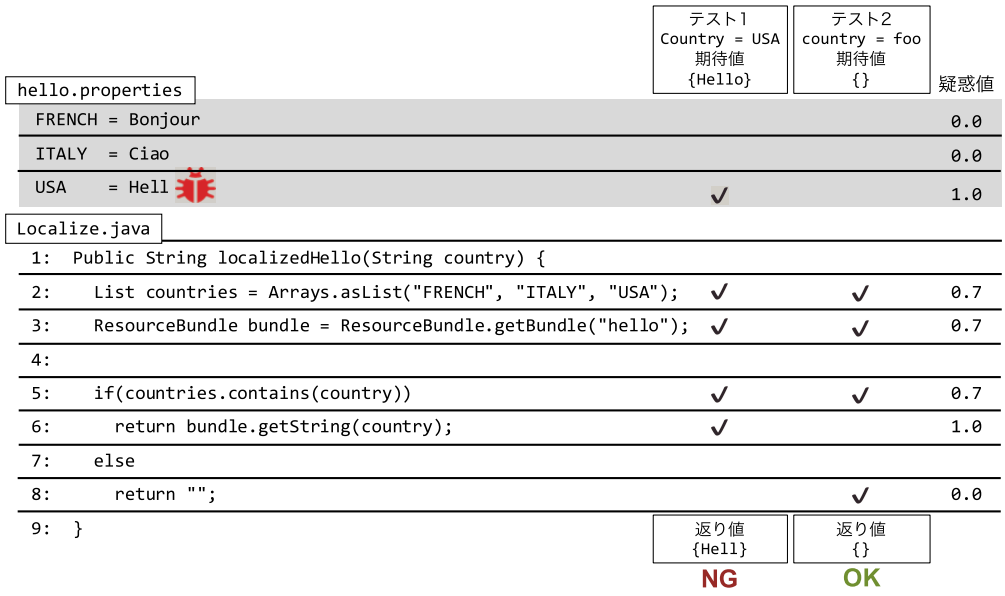


図 5 提案手法が有用な例

Fig. 5 A simplifiled example where the extended technqiue is better than the exiting technique.

表 5 実行時間の内訳

Table 5 Breakdown of execution time.

処理内容	割合
ビルド	53%
テスト実行	32%
ソースコードの FL	9%
ソースコードの加工	5%
プロパティファイルの FL	1%

6.4 プロパティファイルが正しく読み込めない場合

プロパティファイルが正しく読み込めない場合（ファイルそのものが存在しない，ファイル内のフォーマットが正しくない等）は，プロパティファイルの読み込みエラーとなる．そのため，この場合は，提案手法と既存手法に FL の能力の差はない．

7. 関連研究

Zhang らはソフトウェアの進化にともない変更すべき設定ファイルのキーを自動的に特定する手法を提案している [17]．この手法では，新旧のソフトウェアの実行経路を比較することにより，その経路に影響を与えているプロパティファイルのキーを特定している．また，Rabkin らも，バグの原因となっている設定ファイルのキーを自動的に特定する手法を提案している [14]．Zhang らの手法が制御フロー解析を利用しているのに対して，Rabkin らの手法はデータフロー解析を利用している．これら手法はプロパティファイルのみを対象としているのに対して，提案手法はソースコードとプロパティファイルのどちらも対象としているという点で異なる．

また，ソフトウェアによってはその設定ファイルの正しさをチェックする専用ツールが存在している場合もある．たとえば，nginx では “nginx -t 設定ファイル名” とコマンドを実行することにより，設定ファイルの文法の正しさを確認することができる．提案手法はそのような専用のツールとは異なり，プロパティファイルを利用しているソフトウェアにおいて広く利用でき，意味的に誤っている箇所を FL できる．

8. 提案手法および実験の妥当性について留意すべき点

8.1 追加したテストケース

本研究では，提案手法を 2 件の企業のシステムと 2 件のオープンソースソフトウェアに対して適用した．その際に，バグを原因として失敗テストケースが存在しないバグがあった．そういったバグに対して，バグによって失敗テストケースの追加を行って提案手法を適用した．そのため，追加したテストケースは開発者がバグを発見したものとは異なる．開発者の作成したテストケースを用いて実験を行った場合，異なる結果が得られる可能性がある．

8.2 プロパティファイルのバグに対する既存手法の FL の扱い

本実験では，プロパティファイルのバグについては，既存手法はプロパティファイルのキーを読み込んでいる箇所をバグの原因箇所として評価を行った．しかし，図 5 の 6 行目のように，読み込むキーが動的に決まる場合は，ソースコードにおいてプロパティファイルのキーを読み込んでいる箇所をバグの原因箇所として扱うのは不十分である．しかし，プロパティファイルのすべてキーをバグを検出す

るために調査必要な行として計上するのは，提案手法に有利に働きすぎる場合があると著者らは考えたため，このような評価を行った．

8.3 テストケースの実行

4.2 節で述べたように，提案手法では各テストケースは個別に実行される．そのため，従来はすべてのテストケースの実行前に一度だけ実行されていた静的なフィールド変数の初期化が，提案手法では各テストケースの実行時に行われることになる．よって，テストケースが非常に多い場合や，静的なフィールド変数の初期化が時間を要する処理をともなう場合は，提案手法を適用するにあたり長い時間を必要とする場合もありうる．

8.4 実験対象

本研究では，提案手法を 2 件の企業のシステムと 2 件のオープンソースソフトウェアに対して適用した．他の企業のシステムや，オープンソースソフトウェアに対して適用した場合，異なる結果が得られる可能性がある．

8.5 提案手法とソフトウェア開発におけるテスト工程の親和性

プロパティファイルには，実行環境に依存する設定値やユーザがカスタマイズできるようにプログラムの外に定義した定数値が記述される．前者の場合には，ソフトウェア開発におけるテスト環境と本番環境が異なる場合には異なるプロパティファイルを用いることになるので，テスト工程で本番環境におけるプロパティファイルにバグが存在していても見つけることはできない．

9. おわりに

本研究では，既存の FL 手法をプロパティファイルまで拡張した手法を提案した．提案手法では，対象プログラムとそのテストスイートを入力として与えることで，ソースコードの各プログラム文とプロパティファイルの各キーへ疑惑値を計算する．

提案手法を，企業のシステムのバグとオープンソースソフトウェアのバグに対して適用し，既存手法と比較した．その結果，3 件のバグについて既存手法より順位の改善が見られた．また，順位が改善しなかったバグでも，実用上提案手法の方が優れていると考えられるバグも存在した．実行時間は，最大 4.5%の増加であったが，増加した時間は十分に小さなものであった．

今後の取り組みとして，対象とするファイルの種類を増やしていくことがあげられる．次の目標としては，最も多くのリポジトリで修正されていた XML ファイルがあげられる．

参考文献

[1] Abreu, R., Zoeteweij, P., Golsteijn, R. and Van Gemund, A.J.: A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software*, Vol.82, No.11, pp.1780–1792 (2009).

[2] Abreu, R., Zoeteweij, P. and Van Gemund, A.J.: An evaluation of similarity coefficients for software fault localization, *Proc. Pacific Rim International Symposium on Dependable Computing* (*PRDC*), pp.39–46, IEEE (2006).

[3] Abreu, R., Zoeteweij, P. and Van Gemund, A.J.: On the accuracy of spectrum-based fault localization, *Proc. Testing: Academic and Industrial Conference Practice and Research Techniques* (*TAIC PART*), pp.89–98, IEEE (2007).

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

[5] Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E.: Pinpoint: Problem determination in large, dynamic internet services, *Proc. International Conference on Dependable Systems and Networks* (*DSN*), pp.595–604, IEEE (2002).

[6] Dallmeier, V., Lindig, C. and Zeller, A.: Lightweight defect localization for java, *Proc. European Conference on Object-Oriented Programming* (*Ecoop*), pp.528–550, Springer (2005).

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

[8] Jin, W. and Orso, A.: F3: fault localization for field failures, *Proc. International Symposium on Software Testing and Analysis* (*ISSTA*) (2013).

[9] Jones, J.A., Harrold, M.J. and Stasko, J.: Visualization of test information to assist fault localization, *Proc. International Conference on Software Engineering* (*ICSE*), pp.467–477, ACM (2002).

[10] Korel, B.: PELAS-Program Error-Locating Assistant System, *IEEE Trans. Software Engineering* (*TSE*), Vol.14, pp.1253–1260 (1988).

[11] Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S.P.: SOBER: statistical model-based bug localization, *ACM SIGSOFT Software Engineering Notes*, Vol.30, No.5, pp.286–295, ACM (2005).

[12] NIST: Software Errors Cost U.S. Economy $59.5 Billion Annually, available from ⟨http://www.abeacha.com/NIST_press_release_bugs_cost.htm⟩.

[13] Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L. and Verbrugge, C.: A framework for optimizing Java using attributes, *Proc. Conference of the Centre for Advanced Studies on Collaborative Research* (*CASCON*), p.8, IBM Press (2000).

[14] Rabkin, A. and Katz, R.: Precomputing Possible Configuration Error Diagnoses, *Proc. 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp.193–202 (2011).

[15] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot: A Java bytecode optimization framework, *Proc. Conference of the Centre for Advanced Studies on Collaborative Research* (*CASCON*), pp.214–224, IBM Corp. (2010).

[16] Wong, W.E., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A survey on software fault localization, *IEEE Trans. Software Engineering* (*TSE*), Vol.42, No.8, pp.707–740 (2016).

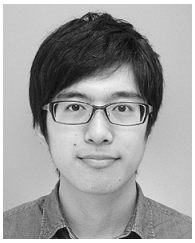
[17] Zhang, S. and Ernst, M.D.: Which Configuration Option Should I Change?, *Proc. 36th International Conference on Software Engineering*, pp.152–163 (2014).

肥後 芳樹 （正会員）

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同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助教．2015 年同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニングおよび自動プログラム修正に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

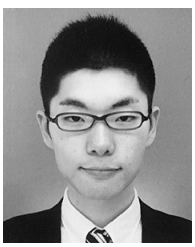
柗本 真佑 （正会員）

2010 年奈良先端科学技術大学院大学博士後期課程修了．同年神戸大学大学院システム情報学研究科特命助教．2016 年大阪大学大学院情報科学研究科助教．博士（工学）．エンピリカルソフトウェア工学の研究に従事．



内藤 圭吾

2019 年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博士前期課程修了．在学時自動プログラム修正および自動バグ検出に関する研究に従事．



谷門 照斗

2017 年大阪大学基礎工学部情報科学科卒業．2019 年同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博士前期課程修了．在学時自動プログラム修正に関する研究に従事．

楠本 真二 （正会員）

1988 年大阪大学基礎工学部卒業．1991 年同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 年同講師．1999 年同助教授．2002 年同大学大学院情報科学研究科助教授．2005 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．

切貫 弘之 （正会員）

2015 年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博士前期課程修了．同年日本電信電話（株）入社．2015 年度コンピュータサイエンス領域奨励賞（情報処理学会）．2018 年度山下記念賞受賞（情報処理学会）．主にリポジトリマイニング，ソースコード解析，ソフトウェアテスト効率化に関する研究開発に従事．

倉林 利行 （正会員）

2012 年慶應大学理工学部システムデザイン工学科卒業．2014 年同大学大学院理工学研究科総合デザイン専攻博士前期課程修了．同年日本電信電話（株）入社．2017 年度コンピュータサイエンス領域奨励賞（情報処理学会）．主にソフトウェアの実装とテスト自動化に関する研究開発に従事．

丹野 治門 （正会員）

2007 年電気通信大学電気通信学部情報工学科卒業．2009 年同大学大学院電気通信学研究科情報工学専攻博士前期課程修了．同年日本電信電話（株）入社．2008 年未踏ユース・スーパークリエータ認定（情報処理推進機構）．2009 年山内奨励賞（情報処理学会）．2013 年山下記念研究賞（情報処理学会）．2016 年企業賞（情報処理学会）．2016 年社長表彰（日本電信電話）．2018 年優秀発表賞．学生奨励賞（日本ソフトウェア科学会）．ソフトウェアテスト，デバッグに関する研究開発に従事．