設定ファイルを考慮した自動バグ限局手法の拡張

肥後 芳樹[1]　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]

概要：自動バグ限局とは，バグの原因箇所を自動で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バグ限局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行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行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バグ限局を行うため，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，正しくバグ限局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バグ限局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バグ限局手法）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バグ限局手法）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バグ限局ができていること，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バグ限局の精度がほとんど変わらないことを確認した．

1. はじめに

ソフトウェア開発において，デバッグは多大な労力とコストを必要とする作業である．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自動バグ限局がある．自動バグ限局とは，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．これまで様々な自動バグ限局手法が提案されている [8,10,11]．その中で近年最も盛んに研究されている手法のひとつとして，テストケースによる実行経路の情報を用いて自動バグ限局を行う 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バグ限局を行う．具体的には，各プログラム文が失敗テストケース，及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 では正しくバグ限局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

[1] 大阪大学大学院情報科学研究科 大阪府吹田市
[2] 日本電信電話株式会社 東京都港区
[*1] National Institute of Standards and Technology, アメリカ国立標準技術研究所

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．

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みなした．

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みなした．

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．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関わらず，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わかる．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バグ限局では不十分な場合がある．例えば，図 7 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 7 の 6 行目）を見ても，どのキーを読み込んでいるかわからないためソースコードのみを対象としたバグ限局では不十分である．

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であり，バグ修正コミットで 26 回変更されていた[*3]．二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，第二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．第三カラムは，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 ファイル以外の全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．例えば，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一歩として，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．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

[*2] GitHub で “Java” のラベルがついたプロジェクトのうち，スター数が上位 100 件のプロジェクトが対象．

[*3] 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，全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．

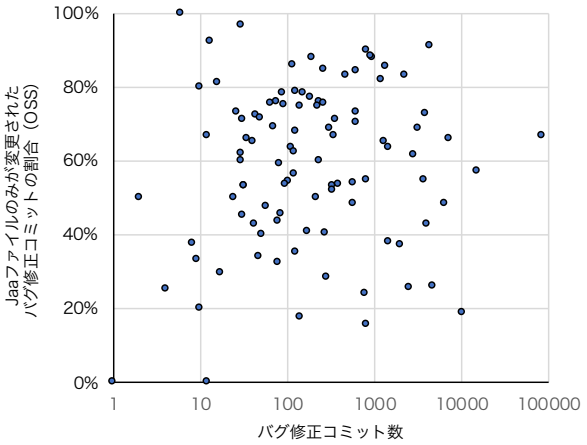


図 1　各 OSS プロジェクトにおいて，Java ファイルのみが変更されたバグ修正コミットの割合

```
PORT_NUMBER = 4000
HOST_NAME = local host
```

図 2　プロパティファイルの一例

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）

拡張子	プロジェクト数	総変更回数（割合）
xml	90	34,923 (12.39%)
拡張子無し	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%)

純である．

3. プログラムの実行経路情報を利用した自動バグ限局

バグの原因箇所を自動的に特定する自動バグ限局手法が研究されており，その中の一手法としてプログラムの実行経路情報を利用したバグ限局手法[*4]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[*5]を付与する．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

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

4. 提案手法

本研究では，既存の自動バグ限局手法を拡張し，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

4.1 実装

提案手法の実装は以下の 4 つの要素に分けられる．

- ライブラリの拡張
- ソースコードの加工
- テストスイートの実行
- 疑惑値の計算

以降ではそれぞれの実装方法を述べる．

ライブラリの拡張

Java では，プロパティファイルの読み込みには，`Properties` クラス[*6]，もしくは `ResourceBundle` クラス[*7]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回数を記録する機能を実装した．

ソースコードの加工

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．

テストケースの実行

プロパティファイルの読み込みはプログラムの処理中に行われることもあるが，フィールド変数の初期化として読み込まれることもある．静的なフィールド変数[*8]に対してプロパティファイルの読み込みが行われた場合，その処理は全てのテストケースの実行前に一度だけ行われる．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個別に実行する．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

[*4] Spetrum-Based Fault Localization．以降，SBFL 手法と呼ぶ．
[*5] 以降，疑惑値と呼ぶ．疑惑値は一般的に 0〜1 の値をとる．
[*6] `java.lang.Properties`
[*7] `java.util.ResourceBundle`
[*8] `static` キーワードが付与されたフィールド変数

め，キーの疑惑値を計算できる．

疑惑値の計算

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，プログラムの各プログラム文に対する疑惑値の計算である．プログラムの各行に対す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．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[*9]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ため，Defects4J に含まれているバグそのものは利用できないが，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を利用した．

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 ファイルおよびプロパティファイルのみが修正されている．

Apache Commons Math と Closure Compiler からのバグ収集手順は以下の通りである．

(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，修正ファイルが Java ファイルおよびプロパティファイルのみの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．第二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．第三および第四カラムはバグ限局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バグを限局するため，各バグで限局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．なお，全てのバグは，プロパティファイルもしくは Java ソースファイルの単一行の変更で修正可能であり，その行をバグの原因箇所とした．

[*9] https://github.com/rjust/defects4j

5.2 手順

実験は以下の手順で行った．

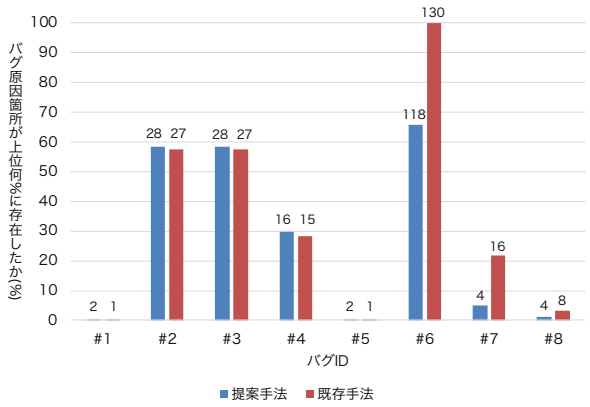
(1) バグ修正コミットの 1 つ前のコミットをチェックアウトする．
(2) そのコミットでテストケースを実行する．
(3) バグが原因となる失敗テストケースが含まれない場合，著者らがそのような失敗テストケースを作成する．
(4) 対象プロジェクトに対して，提案手法と既存手法を実行する．

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頂いた．

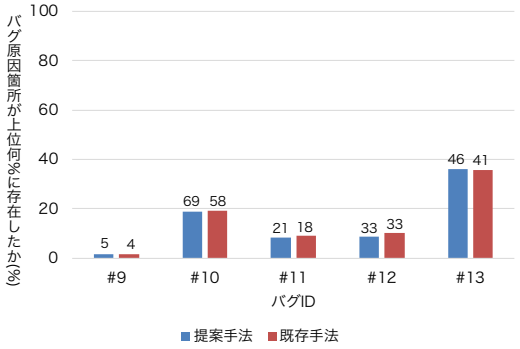
評価指標として，2 つの指標を用いた．1 つ目は，バグの存在する箇所の疑惑値の順位である．より高い順位にバグ原因箇所が順位付けされている手法が，よりバグ限局の精度が高いと評価する．既存手法では プロパティファイルのキーに含まれるバグは見つけることができない．そのため，既存手法の評価では，バグの原因であるプロパティファイルのキーを読み込んでいる行をバグ原因箇所として評価を行った．この指標は，疑惑値の高いプログラム文から手作業により調べていった場合，バグ原因箇所を見つけるために調査しなければならないプログラム文の数およびキーの数を表している．そこで，複数のプログラム文やキーが同じ疑惑値を持つ場合，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

表 2 対象バグ

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



(a) 設定ファイルのバグに対する結果



(b) Java ソースコードのバグに対する結果

図 3 提案手法および既存手法を用いたバグ限局の結果．グラフの Y 軸はバグ限局の精度（バグの原因箇所がバグ限局結果の上位何％の位置に存在したのか）を表す．なお，既存手法では限局対象は 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，バグ限局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 の第三カラムの値のみを用いるが，提案手法は第三カラムと第四カラムの合計値を用いる．また，各棒グラフの上の数値はバグ限局におけるバグの原因箇所の順位を表す．

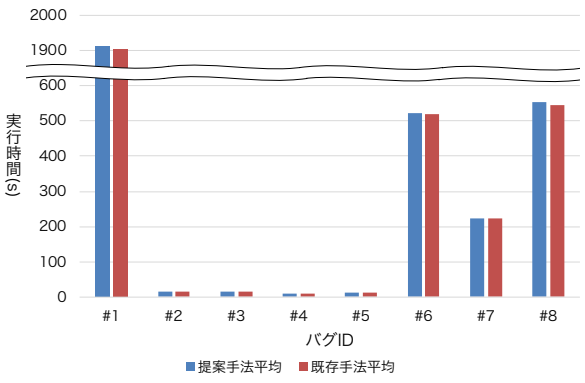


図 4 提案手法および既存手法による実行時間

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

まず，バグ原因箇所の順位に関する結果を図 3 に示す．この図では，Y 軸はバグ原因箇所がランキングの上位何%に存在していたかを示している．棒の上にある数字は，それぞれ何位にバグ原因箇所が位置していたかを示している．図 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．しかし，この悪化は提案手法がプロパティファイルの行数も加味して精度を計算していることに依るものであり，実質的な悪化ではない．

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図 3(b) に示す．図 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限局できなかった．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．

次に，実行時間についての結果を図 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わかる．

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，説明を簡単化するため，実際のバグそのもの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 5 に示す．図 5 (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 5 (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-

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

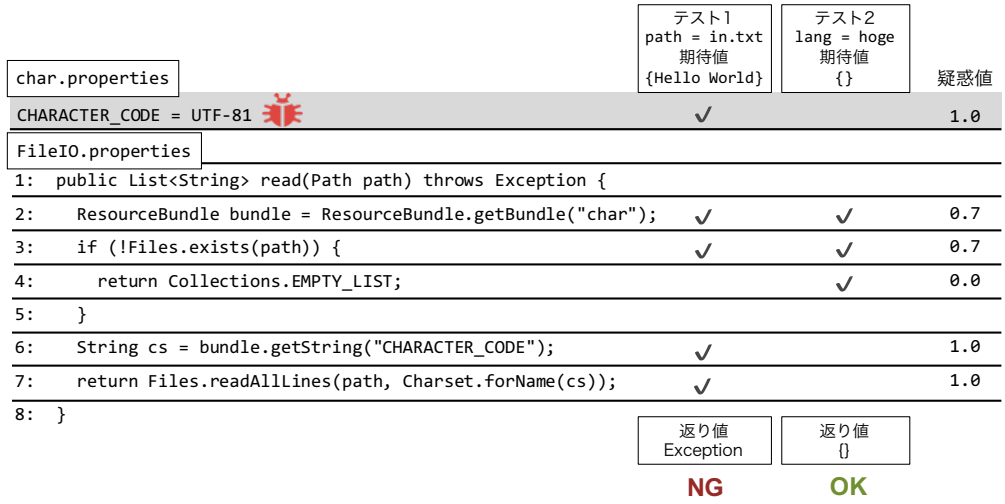
図 5 提案手法により順位が改善した例

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行であっても，その行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 6 に示す．図 6(a) のメソッドは，Path を引数に取り，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 6(a) のテスト 1 で読み込まれている，“in.txt” の中身を図 6(b) に示す．この例も，6.1 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” が読み込まれている行が，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” の読み込み行の実行回数が一致する．したがって，“CHARACTER_CODE” と，“CHARACTER_CODE” の読み込み行が同じ疑惑値，順位になる．5.2 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 6 のような例では順位が悪



(a) 対象のソースコード

(b) 入力ファイル

図 6 提案手法により順位が悪化した例

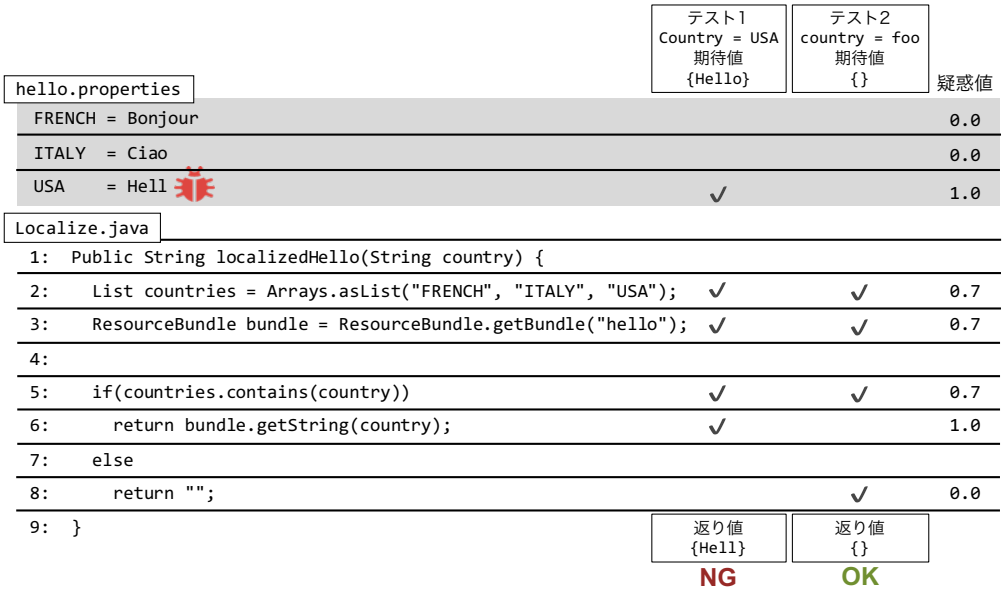


図 7 提案手法が有用な例

化した．

しかし，順位が悪化したにも関わらず，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 7 に示す．

図 7 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わかる．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

図 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ついて考察を行っていく．最もオーバー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 3 にその結果を示す．

表 3 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バグ限局は合計で 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挙げられる．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．

6.4 関連研究

Zhang らはソフトウェアの進化に伴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設定が間違っているがどのキーが間違っているかわからない状態で利用する手法であるのに対して，提案手法はソースコードやプロパティファイルのどこにバグがあるのかわからない状態でも利用できる．

7. 妥当性の脅威

7.1 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．

表 3 実行時間の内訳

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

7.2 プロパティファイルのバグに対する既存手法のバグ限局の扱い

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 7 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全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．

7.3 テストケースの実行

4.1 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，従来は全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伴う場合は，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．

7.4 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. おわりに

本研究では，自動バグ限局の既存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，ソースコードの各行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挙げられる．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 ファイルが挙げられる．

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, *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC)*, IEEE, pp. 39–46 (2006).

[3] Abreu, R., Zoeteweij, P. and Van Gemund, A. J.: On the accuracy of spectrum-based fault localization, *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART)*, IEEE, pp. 89–98 (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, *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 595–604 (2002).

[6] Dallmeier, V., Lindig, C. and Zeller, A.: Lightweight defect localization for java, *Proceedings of European Conference on Object-Oriented Programming (Ecoop)*, Springer, pp. 528–550 (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, *Proceedings of the 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, *Proceedings of the International Conference on Software engineering (ICSE)*, ACM, pp. 467–477 (2002).

[10] Korel, B.: PELAS-Program Error-Locating Assistant System, *IEEE Transactions on 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, ACM, pp. 286–295 (2005).

[12] NIST: Software Errors Cost U.S. Economy $59.5 Billion Annually, `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, *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, IBM Press, p. 8 (2000).

[14] Rabkin, A. and Katz, R.: Precomputing Possible Configuration Error Diagnoses, *Proceedings of the 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, *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, IBM Corp., pp. 214–224 (2010).

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

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