修士学位論文

題目

ソースコード以外のファイルを考慮した自動バグ限局手法の提案

指導教員

楠本 真二 教授

報告者

内藤 圭吾

平成 31 年 2 月 6 日

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

平成 30 年度 修士学位論文

ソースコード以外のファイルを考慮した自動バグ限局手法の提案

内藤 圭吾

内容梗概

デバッグ支援の 1 つに，自動バグ限局という手法がある．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．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バグの原因箇所を推定する．ただし既存手法は，テストケースによる実行回数に基づいてバグ限局を行うため，設定ファイル等直接は実行されない箇所に対しては，バグ限局を行うことができない．という課題が存在している．

本研究では，Properties ファイルと Java ファイルを対象として自動バグ限局を行うことを試みる．Properties ファイルとは，キーとバリューが対応する形式で記述する設定ファイルである．提案手法では，Properties ファイルの読み込みを監視し，読み込み回数に基づき各キーや各バリューがバグの原因箇所である可能性を計算する．Java ファイルのバグ限局には既存手法を利用した．提案手法の評価実験として，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提案手法を適用した．その結果，3 件のバグについて提案手法は既存手法より正確なバグ限局を行うことに成功した．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% 増加と大きな違いは見られなかった．

主な用語

デバッグ支援，自動バグ限局，設定ファイル

目次

1 はじめに 1

2 関連研究 2

2.1 Spectrum-Based Techniques 2

2.2 Slice-Based Techniques 3

2.3 Statistics-Based Techniques 3

2.4 Program State-Based Techniques 3

2.5 Machine Learning-Based Techniques 3

2.6 Data Mining-Based Techniques 4

2.7 Model-Based Techniques 4

2.8 その他の手法 4

3 既存手法の課題調査 6

3.1 調査方法 6

3.2 調査結果 8

4 研究動機 11

5 提案手法 15

5.1 キーアイデア 15

5.2 実装 15

6 実験 18

6.1 実験準備 18

6.2 実験手順 19

6.3 結果 19

7 考察 23

7.1 提案手法により順位が改善したバグ 23

7.2 提案手法により順位が悪化したバグ 24

7.3 実行時間 26

8 妥当性の脅威 28

8.1 追加したテストケース 28
8.2 実験対象 28

9 おわりに 29

10 謝辞 30

参考文献 31

図目次

1 テストによるプログラムの実行経路 (網掛け部分が実行されたコード) 9
2 成功テストケース数の違いによる，各行の疑惑値の変化 9
3 修正内容に Java 以外が含まれるバグ修正コミット 12
4 各拡張子のファイルが修正されたプロジェクト数 13
5 XML ファイルの一例 13
6 Properties ファイルの一例 13
7 既存手法および提案手法でバグ原因箇所が存在した順位の上位からの割合．グラフの上の数字は，バグ原因箇所が存在した順位を表す． 20
8 バグ原因箇所が Java のみのバグに対する実験結果．図中のラベルや軸は図 7 と同じである． 20
9 提案手法のオーバーヘッド 21
10 提案手法および既存手法による実行時間 21
11 提案手法により順位が改善した例 23
12 提案手法により順位が悪化した例 25
13 提案手法が有用な例 26
14 実行時間の割合 27

表目次

1 企業のバグへの Ochiai 適用結果 8
2 調査対象 11

1　はじめに

今日，ソフトウェアは人々の生活に根付いている．多くのセキュリティも，原子力発電の制御もソフトウェアによって支えられ，その影響は計り知れない．そのため，ソフトウェアに含まれるバグは，時間や資産，さらに人命に対して多大な損失を生じさせる [1]．NIST は，アメリカ国内でソフトウェアのバグによって年間約 595 億円の損失が生じていると報告した [2]．一方で，バグの修正にも多大なコストがかかる．開発にかかるコストの内，半数以上はデバッグが占めているという報告もある [3, 4]．そのため，デバッグ支援が必要である．

デバッグ支援の 1 つに自動バグ限局という手法がある．デバッグは以下の 2 つの工程に分けられる．

- バグ原因箇所の特定
- バグ原因箇所の修正

自動バグ限局は上記の工程の内，“バグ原因箇所の特定” を支援する手法である．これまで様々な自動バグ限局手法が提案されている [5, 6, 7]．その中で近年最も盛んに研究されている手法が，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 Techniques である [8]．Spectrum-Based Techniques は，失敗テストケースで実行された行はバグ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行はバグ原因箇所である可能性が低い，アイデアに基づいてバグ限局を行う．Spectrum-Based Techniques では，各行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，その行がバグ原因箇所である可能性 (以下，疑惑値) を計算する．これまで，多くの Spectrum-Based Techniques が提案されてきた [9, 10, 11]．しかし，Spectrum-Based Techniques は，テストケースによる実行経路に基づき各行の疑惑値を計算するため，テストケースによって直接実行されることのない，設定ファイル等に含まれるバグ原因箇所に対しては疑惑値を計算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 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提案手法を適用した．その結果，3 件のバグについて提案手法は既存手法より正確なバグ限局を行うことに成功した．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% の増加と大きな差は見られなかった．

2 関連研究

自動バグ限局手法は以下の 8 つに分けられる [8].

- Spectrum-Based Techniques
- Slice-Based Techniques
- Statistics-Based Techniques
- Program State-Based Techniques
- Machine Learning-Based Techniques
- Data Mining-Based Techniques
- Model-Based Techniques
- その他の手法

以下，それぞれについて述べていく.

2.1 Spectrum-Based Techniques

Spectrum-Based Techniques は，プログラムの実行情報を用いてバグ限局を行う手法である．プログラムの実行情報とは，テストケースによって実行された行の情報とそのテストケースの成否である．Spectrum-Based Techniques の基本的な考えは，失敗テストケースで実行された行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行は正しい可能性が高い，というものである.

Spectrum-Based Techniques の 1 つに Ochiai がある [12]．Ochiai はソースコードの各行に対して，その行にバグが含まれている可能性 (以下，疑惑値) を 0 〜 1.0 で計算する．Ochiai による i 行の疑惑値 $suspiciousness(i)$ は以下の計算式で表される．f

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

式中の変数はそれぞれ以下の意味合いである.

$totalFail$: 失敗テストケースの総数.
$fail(i)$: i 行を実行する失敗テストケースの数.
$pass(i)$: i 行を実行する成功テストケースの数.

Ochiai は複数の論文 [13, 14] で，他の Spectrum-Based Techniques より性能が優れていることが指摘されている.

2.2 Slice-Based Techniques

Slice-Based Techniques とはプログラムスライシングを用いてバグ限局を行う手法である．プログラムスライシングとは，プログラム中の文に含まれる変数に注目し，その変数に影響を与えるソースコードを抽出する手法である [15]．プログラムスライシングによって抽出されたソースコードをプログラムスライスという．Slice-Based Techniques の基本的な考えは，バグの原因が変数の誤りである場合，その変数に注目したプログラムスライスを探索することで，探索空間を狭めることができる，というものである．楠本らにより，Slice-Based Techniques の評価が行われ，その有用性が示された [16]．

2.3 Statistics-Based Techniques

Statistics-Based Techniques はプログラムの述語に着目してバグ限局を行う手法である．述語とは引数をとり，引数に応じて真偽値を返す処理である． Statistics-Based Techniques は，成功テストケースと失敗テストケースで，述語の結果に偏りがあった場合，その述語はバグに関係がある，というアイデアに基づいてバグ限局を行う．Statistics-Based Techniques の 1 つに，Ben Liblit らによって提案された手法がある [17]．Ben Liblit らはその手法を CCRYPT や THYTHMBOX など，広く利用されている 5 件のシステムに適用し，未知のバグを見つけるなどの形でその有用性を示した．

2.4 Program State-Based Techniques

Program State-Based Techniques は成功テストケースと失敗テストケースにおける “プログラムの状態” の違いを用いてバグ限局を行う手法である．プログラムの状態とは，プログラム実行中の注目しているタイミングにおける変数とその値のことである．Program State-Based Techniques の 1 つに，Zeller による Delta Debugging がある [18]．Delta Debugging はツールとして実装され，自動デバッグのために実際に広く使われている [19, 8]．

2.5 Machine Learning-Based Techniques

Machine Learning-Based Techniques はカバレッジや，その実行結果などの情報に基づいて機械学習のモデルを作成することで，バグ限局を行う手法である．カバレッジとはプログラム全体のうち，テストケースによって実行された命令や分岐の割合のことである．Machine Learning-Based Techniques の 1 つに Wong らが提案したバグ限局手法 EXAM がある [20]．EXAM は，仮にプログラムの 1 行だけを実行するテストケースが存在し，そのテストケースが失敗した場合，そのテストケースによって実行された行はバグ原因箇所である可能性が高い，というアイデアに基づいてバグ限局を行う．EXAM はまず，バグ限局の対象であるプロジェクトの各テストケースの実行結果をラベルとして，各テスト

ケースによるカバレッジをモデルに学習させる．その後，実行された行が 1 行のみのカバレッジを入力としてモデルに与える．その出力が “失敗” であった場合，入力として与えたカバレッジで実行されている行がバグの存在箇所である可能性が高いと推定する．Wong らは EXAM を Spectrum-Based Techniques のひとつである Tarantula と比較し優れた性能を示した．

2.6 Data Mining-Based Techniques

自動バグ限局の問題を抽象化していくと，データマイニングの問題に落とし込むことができる．例えば，完全な実行トレースはバグ限局において価値のある情報である．しかし，情報量があまりにも多いため，利用することは困難である．そういった問題を解決するために，データマイニングを用いる手法が Data Mining-Based Techniques である [21, 22]．Data Mining-Based Techniques の 1 つに，Nessa らによる手法がある [23]．Nessa らの手法は，まずプログラムの実行可能な経路を N-gram を用いて分割する．その後，N-gram によって分割された各コード片が，失敗テストケースの実行経路に出現する頻度を用いてバグ限局を行う．しかし，N-gram による分割では，分割されたコード片の数が非常に多くなる．そこで，データマイニングを用いて，失敗テストケースの実行経路に頻繁に出現するコード片を探す．Nessa らはその手法と Spectrum-Based Techniques の 1 つである Tarantula と比較し，優れたバグ限局の結果を出力することで，その有用性を示した．

2.7 Model-Based Techniques

Model-Based Techniques はプログラムの振る舞いを表すモデルを用いることでバグ限局を行う手法である．この手法では，モデルの振る舞いを正しい振る舞いとして，バグを含むプログラムとモデルとの振る舞いの違いによってバグ限局を行う．Model-Based Techniques の 1 つに Cristine Mateis らによる JADE がある [24]．JADE は Slice-Based Techniques と比較しすることで，その有用性を示した．

2.8 その他の手法

上記のいずれのカテゴリにも分類されない自動バグ限局手法の多くは，対象とする特定のエラーに特化したものなどがあげられる．

2.8.1 Memory leak

Java では自動でガベージコレクションを行うことで，メモリリークが起こらないようにしている．しかし，意図しない参照により，メモリリークが生じることがある．そこで，様々なメモリリークの原因箇所を特定する手法が提案されてきた [25, 26]．その 1 つに，Chen らが提案した FindLeak がある

[27]．FindLeak はメモリ使用量と，プログラム実行中に作成された参照を収集することで，メモリリークの発生原因を特定する手法である．

2.8.2 Information Retrieval-Based Techniques

近年，自然言語処理，特に情報検索 (Information Retrieval) を用いた手法が研究されている [28, 29]．情報検索を用いた手法を Information Retrieval-Based Techniques という．Information Retrieval-Based Techniques は，バグが発生したときに開発者が作成するバグレポートを解析することでバグ限局を行う手法である．Information Retrieval-Based Techniques の 1 つに Le らによって提案されたものがある [30]．その手法は，情報検索と Spectrum-Based Techniques を組み合わせた手法である．Le らはその手法を 4 つのオープンソースソフトウェアに対して適用し，最新の既存手法 [31, 32, 33] と比べて最大 54% 正確にバグ原因箇所を推定することで，その有用性を示した．

2.8.3 Formula-Based Techniques

Formula-Based Techniques という手法がこれまで提案されてきた [34, 35]．Formula-Based Techniques とは，失敗テストケースを命題論理式に変換することでバグ限局を行う手法である．Formula-Based Techniques の 1 つに，Jose らによる BugAssist がある [36]．BugAssist は，まず成功テストケースと失敗テストケースの実行トレースから，満たすべき命題論理式と満たしてはいけない命題論理式を生成する．これにより，バグ限局の問題を MAX-SAT 問題へ帰着させる．MAX-SAT 問題 [37] とは，与えられた命題論理式の最も多くの節を真にする値割り当てを探索する問題であり，NP -困難であることが知られている [38]．その後，BugAssist は生成した命題論理式を解くことでバグ限局を行う．MAX-SAT 問題を解くために， MAX-SAT ソルバ [39] を用いた．

これらの手法の内，Spectrum-Based Techniques が最も盛んに研究されている [8]．そこで，本研究では Spectrum-Based Techniques を対象として研究を行う．

3　既存手法の課題調査

本研究を行うにあたって，既存手法の課題調査を行った．調査対象の既存手法として，Spectrum-Based Techniques の 1 つである Ochiai を用いた．調査には，Ochiai を Java 向けに実装したツール GZoltar を利用した [40].

3.1　調査方法

企業から提供された 2 件のシステムのバグ，合計 408 個に対して Ochiai を適用し，その結果を評価した．ランキングの上位 20% に，バグ原因箇所が順位づけられていた場合，バグ限局に成功したと判定する．ただし，GZoltar は対象にいくつか制約があるため，次のようなフィルタリングを行った．括弧の中は，そのフィルタリングを行った後残ったバグの数である．

- 提供されたバグ (408 個)
- 単体テストで発見されたバグ (170 個)
- 開発者によるバグ修正情報を取得できるバグ (80 個)
- 修正が Java ファイルのみであるバグ (24 個)
- バグにより失敗する単体テストが存在するバグ (12 個)

以下では，それぞれのフィルタリングの必要性とその方法について述べる．

単体テストで発見されたバグ

ソフトウェアの開発中にバグを発見する方法として，単体テストや結合テストなどが挙げられる．今回対象言語として選んだ Java において，単体テストは JUnit というテスト実行，評価の自動化を行うフレームワークが実装されているが，結合テストは自動化がされていない．GZoltar はバグ限局を行うために，テストケースによる実行経路と，そのテストケースの結果を用いる．そのため，テストケースの実行，評価が自動化されている必要がある．そこで，今回は対象とするバグとして単体テストで発見されているバグを選択した．今後，結合テストが自動化された場合，結合テストで発見されたバグについても対象として実験を行うことができると考えられる．

適用対象のシステムにおいてバグはバグ票で管理されていた．バグ票とは，発生したバグの ID，バグの内容，バグを発見した工程などが記載されたファイルである．そこで，バグ票に単体テストの工程で発見されたと記載されているバグを手動で抽出した．

開発者によるバグ修正情報が取得できるバグ

GZoltar により出力された疑惑値のランキングのどの順位にバグ原因箇所が位置しているか評価するために，開発者によるバグ修正情報が必要である．そこで，企業で利用されているバージョン管理システムから，そのバグの修正内容が取得できるかどうかでフィルタリングを行った．

バージョン管理システムには以下の情報が含まれている．

- 修正日時
- 修正した開発者名
- 修正したファイル名
- 修正内容

バグ票に記載されているバグであっても，修正したリビジョンが確認できないもの，複数のバグが 1 つのリビジョンで修正され修正内容とバグが紐付けられないもの，機能の実装とバグの修正が同時に行われているものなどが存在していた．そのため，バグの修正内容が取得でき，バグと修正内容が一意に紐づけられるかどうか手作業で確認することでフィルタリングを行った．

修正が **Java** ファイルのみであるバグ

GZoltar は Java ファイルに含まれるバグしかバグ限局を行うことができない．しかし，今回実験対象としたシステムには，ソースコードである Java ファイル以外にも，設定ファイルである Properties ファイルや，入出力ファイルである XML ファイルなどが含まれていた．そこで，修正内容が Java ファイルのみであるかどうかのフィルタリングが必要である．

前述した“開発者によるバグ修正情報が取得できるバグ”のフィルタリングと同様に，バージョン管理システムから，その修正ファイルが Java ファイルのみかどうか手作業でフィルタリングを行った．

3.1.1 バグにより失敗する単体テストが存在するバグ

GZoltar は失敗テストケースと，成功テストケースの実行経路を用いてバグ限局を行う．そのため，バグによって失敗テストケースが存在しない場合，バグ限局を行うことが出来ない．多くのバグで，そのバグを原因として失敗テストケースが存在しなかった．そこで本研究では，テストスイートにバグを原因として失敗テストケースの追加を行った．ただし，対象のバグの原因が複雑だった場合など，失敗テストケースの作成が出来なかったものについては，調査の対象から外した．

3.2 調査結果

調査結果を表 1 に示す．

調査結果および，調査の過程より，次のような課題が明らかになった．

- 対象とするバグの少なさ
- 精度の低さ

以下では，それぞれの課題について詳細を述べていく．

対象とするバグの少なさ

3.1 で述べた通り，今回用いたツールには適用するうえでの制約がいくつかある．そこでフィルタリングを行い，対象バグの数が 408 個から 12 個にまで減少した．これは，全体のバグの数に対してわずか約 3% である．開発者によるバグ修正情報を取得できるかどうかで，90 個のバグが除外されたが，これは調査を行う上で自動バグ限局が出力したランキングを評価するためのフィルタリングである．したがって，その 90 個のバグが GZoltar によってバグ限局可能であるかは未確認である．しかし，90 個のバグ全てがバグ限局可能なバグであった場合でも，その数は 25% にとどまる．このバグ限局可能なバグの数の少なさが，既存手法の課題の 1 つである．

しかし，“単体テストで発見されたバグ” は，結合テストが自動化されていないという，既存手法以外の問題による制約である．したがって既存手法の課題は，“修正が Java ファイルのみのバグ” という制約にある．システムはソースコードだけで構成されている場合は稀で，ほとんどの場合は設定ファイルや入出力ファイルも含まれている．そうであるにも関わらず，自動プログラム修正の修正対象がソースコードのみであるという点が既存手法の課題である．今回の実験でも，“修正が Java ファイルのみのバグ” のフィルタリングでバグの数が 80 個から 24 個，70% ものバグが除外されてしまっている．また，今回の実験では，Spectrum-Based Techniques を用いたが，2 で述べたいずれの手法も，対象がソースコードのみである．

以上より，今後の 自動プログラム修正の課題は，その修正対象をソースコードのみから，それ以外の設定ファイルにまで拡張することである．

表 1　企業のバグへの Ochiai 適用結果

Bug ID	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
結果	成功	成功	失敗	失敗	失敗	失敗	失敗	失敗	失敗	失敗	失敗	失敗

```
11: int b = 5;
12: if (a > b) {
13:     return b;
14: } else {
15:     return b;
16: }
```

(a)成功テストケース

```
11: int b = 5;
12: if (a > b) {
13:     return b;
14: } else {
15:     return b;
16: }
```

(b)失敗テストケース

図 1　テストによるプログラムの実行経路 (網掛け部分が実行されたコード)

	成功テストケース数		
	1個	9個	99個
11: int b = 5;	0.7	0.3	0.1
12: if (a > b) {	0.7	0.3	0.1
13: return b;	1.0	1.0	1.0
14: } else {			
15: return b;	0.0	0.0	0.0
16: }			

図 2　成功テストケース数の違いによる，各行の疑惑値の変化

3.2.1　精度の低さ

12 件のバグに対して調査を行い，バグ限局に成功したバグは 2 件であった．精度の低さの原因としてテストケースの不足が考えられる．

カバレッジを指標としてテストケースを作成した場合，一定数のテストケースが作成されるが，自動バグ限局を行うためにはテストケースの数が不十分な場合がある．以下で，テストケースの数が疑惑値に大きく寄与する例を述べる．

図 1 に 2 つのテストケースによる実行経路を示す．この例は，変数 a と b を比較して，値が大きな方を返す処理を行っている．しかし，13 行目が本来ならば “return a;” であるところが，“return b;” なっている．このバグのために (b) のテストケースが失敗している．

テストケースがこの 2 つだけの場合と，(a) の成功テストケースと同様の実行経路の成功テストケースが 9 個，99 個の場合について，各行の疑惑値を図 2 に示す．

図 1 の 2 つのテストケースだけで，命令カバレッジと分岐カバレッジはともに 100% になっている．しかし，図 2 を見ると，テストケース 2 つだけでは，バグ原因箇所である 13 行目に加えて，11 行目と 12 行目の疑惑値も高い値を示している．11 行目，12 行目は成功テストケースと失敗テストケースの両

方で実行されているため，バグ原因箇所ではないにもかかわらず高い疑惑値になってしまっている．図2を見ると，成功テストケースを9個，99個に増やすと，11，12行目の疑惑値が減っていくことが分かる．

以上の例から分かるように，テストケースの数が疑惑値を決定する大きな要素となっている．そのため，テストケースの数が不十分であった場合，自動バグ限局が正しく行われない．

この課題は少ないテストケースで高いカバレッジを達成する，という戦略をとっている場合のみに限った話ではない．例として，0〜9の自然数を入力として，入力の値を四捨五入するプログラムを作成したとする．そのプログラムに対して，代表的なテスト技法である同値分割や，境界値分析に基づきテストケースを作成したとする．その場合，正常系のテストケースとしては，0〜4の範囲に含まれる自然数を入力とするテストケースと，5〜9の範囲に含まれる自然数を入力とするテストケースを高々2つずつ用意すると十分である．しかしこれは，あくまで人間が正しく機能を実装できているか，バグが存在していないかを確かめるためのテストケースであるので，自動プログラム修正を行うためには不十分である可能性がある．つまり，人間がプログラムの正しさを確かめるためのテストスイートと，自動プログラム修正において，正しくバグ限局を行うためのテストスイートは異なる，ということがこの課題の本質である．

4 研究動機

3 で既存手法の問題点を明らかにした．明らかになった問題点は以下の 2 点である．

- 対象とするバグの少なさ
- 精度の低さ

本研究では，“対象とするバグの少なさ” の解決を試みる．理由として，ソースコード以外に含まれるバグの重大さがあげられる．Yin らの調査によると，設定ファイルを原因とするバグの内，9.7% ～ 27.3% のバグによって，システムが使用できない状態になる [41]．また，設定ファイルを原因とするバグの内，20% のバグによってパフォーマンスの低下がみられた．このように，ソースコード以外に含まれるバグの影響が大きいにも関わらず，既存手法ではソースコード以外に含まれるバグについて自動バグ限局を行うことが出来ない．したがって本研究では，自動バグ限局の対象を設定ファイルへ拡張することを試みる．

まず，バグ修正コミットにおいて，修正ファイルに Java ファイル以外が含まれるものがどの程度存在するのかを調査した．調査対象を表 2 に示す．

調査方法は次の通りである．

1. それぞれの調査対象からバグ修正コミットを収集
2. 収集した各バグ修正コミットで修正されていたファイルを調査

バグ修正コミットの収集方法はオープンソースソフトウェアと企業のシステムとでは異なる．以下，それぞれについて，バグの収集方法を述べる．

オープンソースソフトウェア

実験対象のプロジェクトのコミットを解析し，“fix” や “repair” といった言葉がコミットメッセージに含まれるものをバグ修正コミットとして収集する．

表 2　調査対象

調査対象	総コミット数
企業のシステム	1,245
OSS	830,000

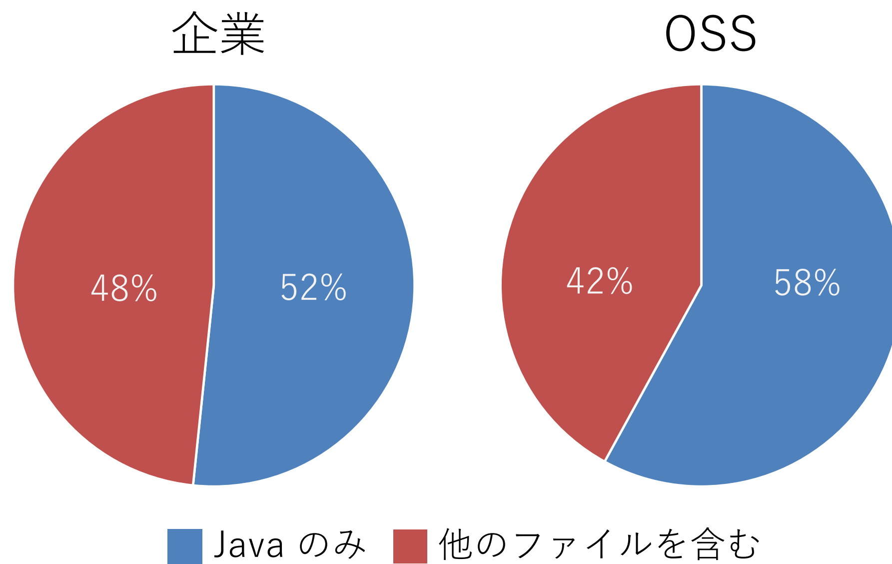


図 3　修正内容に Java 以外が含まれるバグ修正コミット

企業から提供されたシステム

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して収集する．

調査結果を図 3 に示す．図 3 を見ると，オープンソースソフトウェアで 40% 以上，企業のシステムでは約 50% のバグ修正コミットに Java ファイル以外のファイルが含まれていることが分かる．したがって企業のシステム，オープンソースソフトウェア共に 40% 以上のバグについて，既存手法ではバグ原因箇所に対して疑惑値を計算することができない．

次に，バグ修正コミットで修正された Java 以外のファイルについて，どの様な種類のファイルが修正されているかを調査した．調査対象は前述の対象と同じである．オープンソースソフトウェアは，リポジトリ毎にコミット数に大きな違いがあったため，各拡張子のファイルが修正されたコミット数を集計すると，コミット数の多いリポジトリの傾向に全体の結果が大きく影響される．そこで今回の調査では，各拡張子のファイルが一度でも修正されたリポジトリの数を調査した．調査結果を図 4 に示す．

図 4 より，XML ファイルが最も多くのリポジトリで修正され，次に MarkDown ファイル，Properties ファイルと続くことが分かる．XML ファイルは，記述の自由度が高く，ビルドの設定ファイルや，プロジェクトの入出力ファイルなど，様々な用途に利用されている．XML ファイルの例を図 5 に示す．

図 5 はビルドの設定ファイルの一部である．XML ファイルは図 5 の様に，入れ子状にタグで囲まれたデータを記述していく．これにより様々なデータを記述することができるが，その反面構造が複雑になることがある．

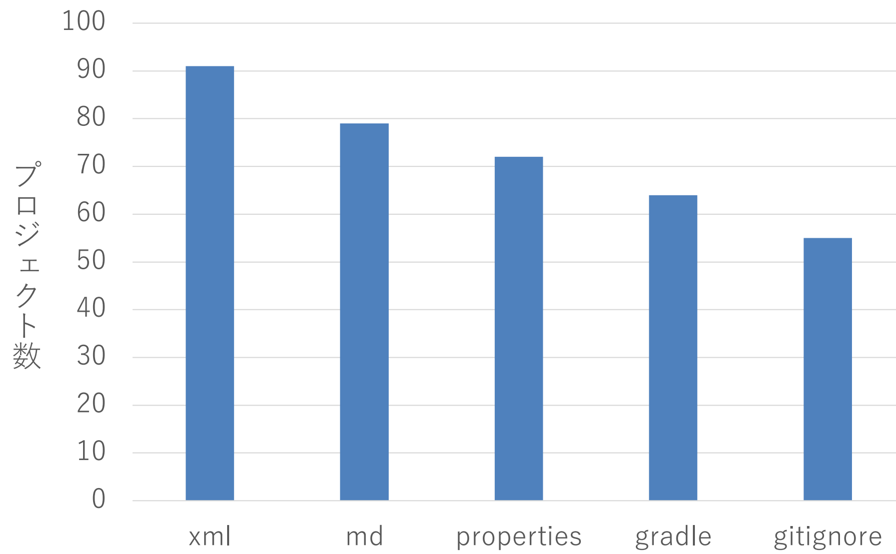


図 4　各拡張子のファイルが修正されたプロジェクト数

```
<target name="jar">
    <mkdir dir="build/jar"/>
    <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
        <manifest>
            <attribute name="Main-Class" value="HelloWorld"/>
        </manifest>
    </jar>
</target>
```

図 5　XML ファイルの一例

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

図 6　Properties ファイルの一例

次に，MarkDown ファイルはドキュメントファイルであるため，バグの原因とはならない．

Properties ファイルは XML ファイルと異なり，用途がプログラムの設定ファイルと限られている．Properties ファイルの例を図 6 に示す．

Properties ファイルは図 6 の様に，“key = value” の形式で記述される．そのため，記述の方法に制限が強いが，その構造は単純なものとなっている．

これまで，Spectrum-Based Techniques を，設定ファイルへと拡張した研究はないため，本研究では第一歩として構造がシンプルな Properties ファイルを対象として研究を行う．今後の課題として，

Properties 以外のファイルへの拡張が考えられる．

5 提案手法

本研究では，既存の自動バグ限局手法を拡張し，Properties 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．提案手法の入力は対象プロジェクトと，対象プロジェクトのテストスイートである．対象プロジェクトは Java で開発されている必要がある．テストスイートは，JUnit で記述され，バグによって失敗テストケースが含まれる必要がある．提案手法の出力は，疑惑値のランキングである．

5.1 キーアイデア

提案手法は，Spectrum-Based Techniques の疑惑値を計算する式を，他のファイルの疑惑値計算にも用いることで，ソースコード以外のファイルについてもバグ限局を行うことができるのではないか，というキーアイデアでに基づいてバグ限局を行う．

疑惑値の計算には Ochiai[12] の計算式を用いた．これは，他の Spectrum-Based な自動バグ限局手法と比較して，Ochiai が優れていることが複数の論文で指摘されているからである [13, 14]．Properties のキー key の疑惑値 $suspicious(key)$ を計算する際，以下の様に Ochiai の計算式を変更した．

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

式中の変数はそれぞれ以下の意味合いである．

$totalFail$: 失敗テストケースの総数.

$fail(key)$: Properties のキー key を読み込む失敗テストケースの数.

$pass(key)$: Properties のキー key を読み込む成功テストケースの数.

5.2 実装

提案手法の実装は以下の 4 つの要素に分けられる．

- 読み込みライブラリの拡張
- 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ていく．

5.2.1 読み込みライブラリの拡張

Properties ファイルは “java.lang.Properties” や “java.util.ResourceBundle” などのライブラリを用いて読み込まれる．こういったライブラリには，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存在しない．そこで，読み込みライブラリを継承し，各キーの読み込み回数を記録するように拡張したライブラリを実装した．

5.2.2 ソースコードの加工

各行の実行回数や，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必要がある．しかし，利便性の点で手作業による書き換えは望ましくない．また，提案手法適用により対象プロジェクトのファイルに変更があることも望ましくない．そこで，提案手法では内部で対象プロジェクトの AST を構築し，その AST に対して加工を行う．これにより，自動かつ対象プロジェクトのファイルに影響を与えることなくソースコードの加工を行うことができる．

加工は以下の 2 つの種類がある．

1. 各行の後に，その行が実行されたかを記録する文を追加する．
2. Properties を読み込むライブラリのインスタンス生成を拡張したものに置換する．

5.2.3 テストスイートの実行

Properties ファイルの読み込みはプログラムの処理中で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．static フィールド変数に対して，Properties ファイルの読み込みが行われた場合，その処理は 1 回のテストスイート実行につき，1 回だけ実行される．しかし，それでは static 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，提案手法ではテストスイートの実行を 1 回につき 1 つのテストメソッドのみ実行するようにしている．これにより，static フィールドで読み込まれている Properties のキーについても正しく疑惑値の計算ができる．ただし，JUnit3 は機能としてテストメソッドごとの実行を提供していない．そのため，対象プロジェクトが JUnit3 で書かれており，static フィールドの初期化で Properties の読み込みを行っていた場合，正しく疑惑値の計算が出来ない．

5.2.4 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，プログラムの各行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，Spectrum-Based Techniques の 1 つ，Ochiai

の計算式を用いた．目は，Properties の各キーに対する疑惑値の計算である．Properties の各キーに対する疑惑値の計算には 5.1 で述べた，Ochiai の計算式を変更した計算式を用いた．

6 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．

6.1 実験準備

実験を行うにあたって，実験対象とするバグの収集を行った．実験対象には 2 件のオープンソースソフトウェアと，2 件の企業から提供されたシステムのバグを用いた．実験対象としたオープンソースソフトウェアを以下の 2 件である．

- apache-commons-math
- Closure-compiler

上記のプロジェクトを選んだ理由は，既存研究の評価に使われているベンチマーク Defects4J に含まれているからである．

実験対象のバグが満たすべき条件は以下の 2 点である．

- そのバグが修正されたコミットが確認できる
- Java ファイルおよび，Properties ファイルのみで修正されている

オープンソースソフトウェアと企業から提供されたシステムとでは，バグの収集方法が異なる．以下それぞれについて，バグの収集方法を述べる．

オープンソースソフトウェア

実験対象のプロジェクトのコミットを解析し，“fix” や “repair” といった言葉がコミットメッセージに含まれるものをバグ修正コミットとして収集する．その後，それらのコミットの内．修正ファイルが Java ファイルおよび Properties ファイルのみのコミットを抽出する．それらのコミットについて，コミットコメントおよび修正内容を目視確認し，条件を満たすと判断できたものを実験対象とした．

企業から提供されたシステム

企業から提供されたシステムはバグが，バグ票によって管理されていた．そこで，バグ票から条件を満たすものを抽出し，それらのバグを実験対象とした．

6.2 実験手順

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を実行する．
3. バグにより失敗テストケースが含まれない場合，テストケースを作成する．
4. 失敗テストケースを含むテストスイートと，プロジェクトを入力と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，企業の開発者により正しいことを確認した．

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 Properties のキーに含まれるバグは見つけることが出来ない．そのため，既存手法の評価では，バグの原因である Properties 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，疑惑値のランキングを上位から調べていった場合，バグ原因箇所を見つけるためにいくつのソースコードおよび Properties のキーを調べる必要があるかを表している．そこで，複数のソースコードや Properties のキーが同じ順位に位置付けられていた場合，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 Properties のキーがランキングの 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，提案手法によるオーバーヘッドの評価を行う．

6.3 結果

まず，バグ原因箇所の順位に関する結果を図 7 に示す．図 7 に含まれるバグは，全てバグ原因箇所として Properties ファイルを含んでいる．図 7 では，縦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．実験の結果，3 件のバグについて，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．

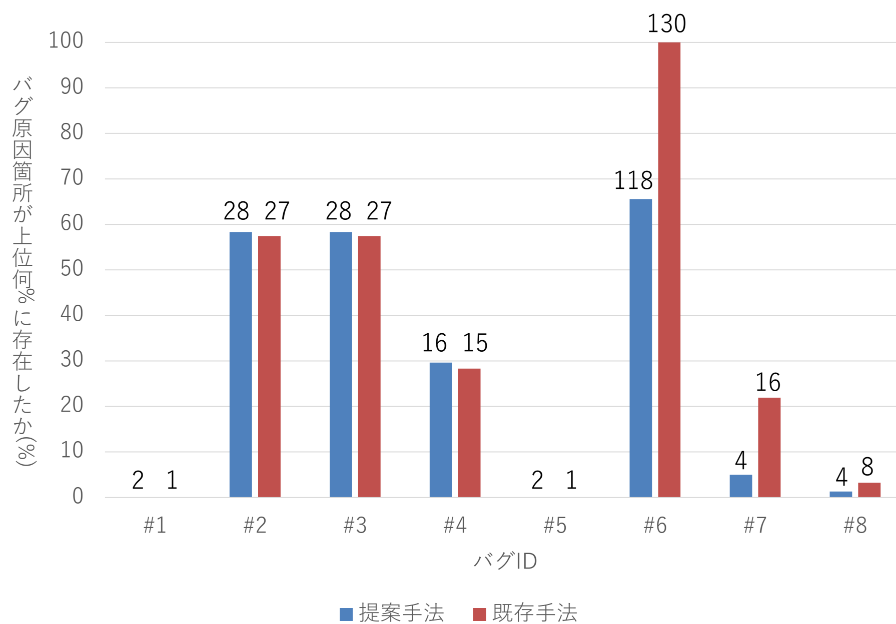


図 7　既存手法および提案手法でバグ原因箇所が存在した順位の上位からの割合．グラフの上の数字は，バグ原因箇所が存在した順位を表す．

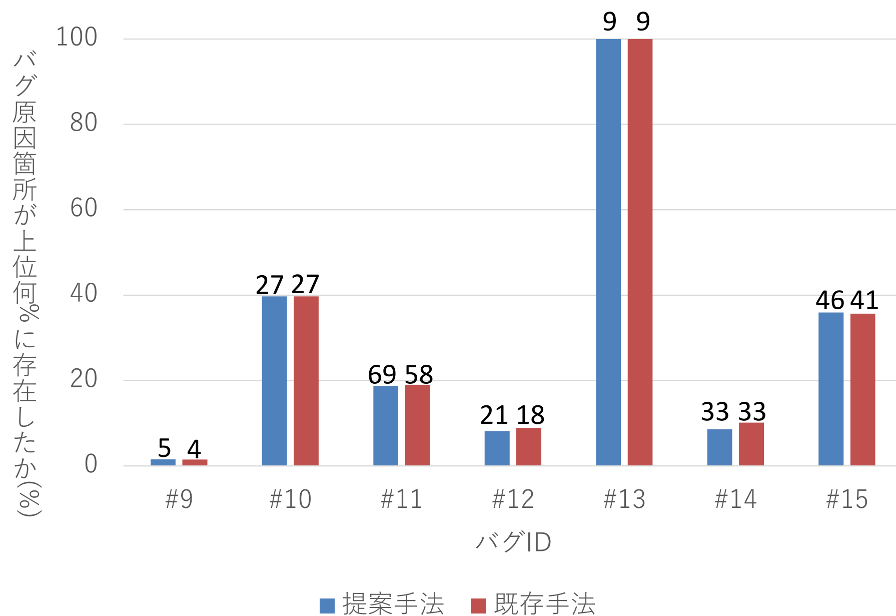


図 8　バグ原因箇所が Java のみのバグに対する実験結果．図中のラベルや軸は図 7 と同じである．

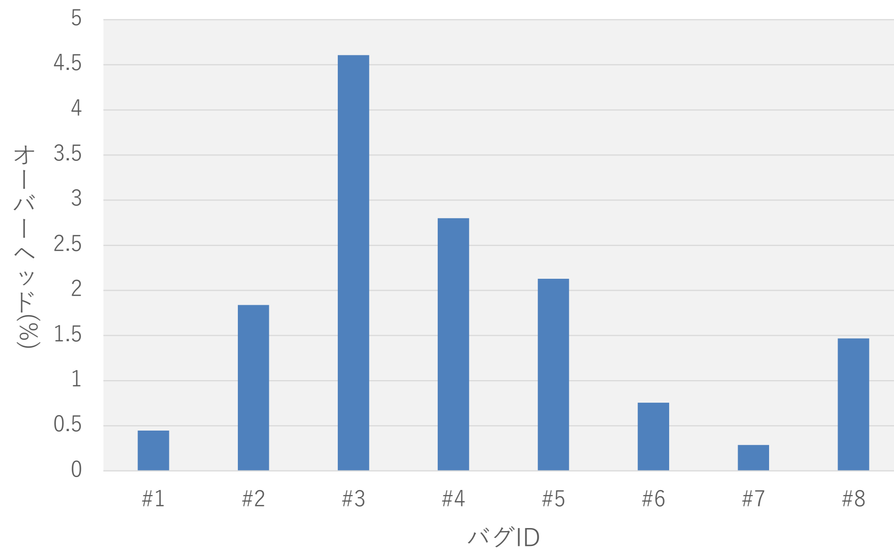


図 9　提案手法のオーバーヘッド

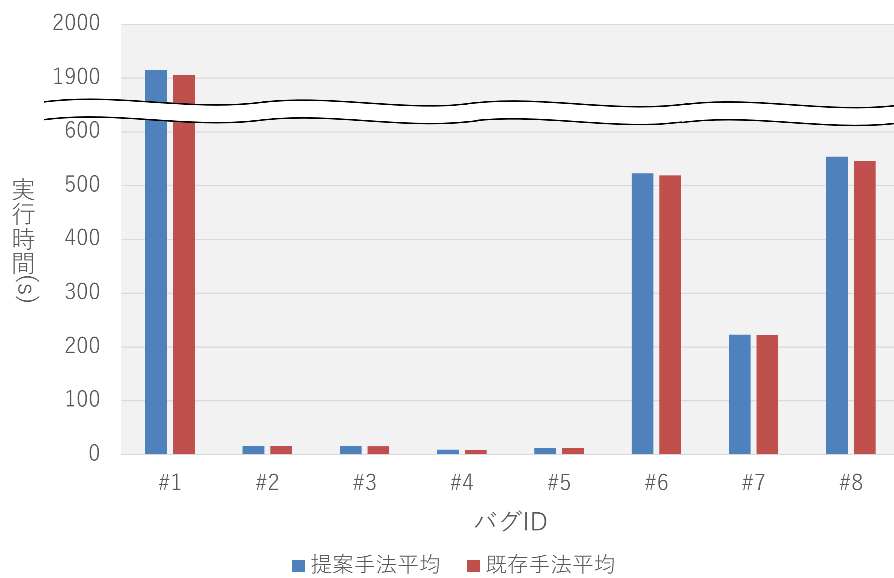


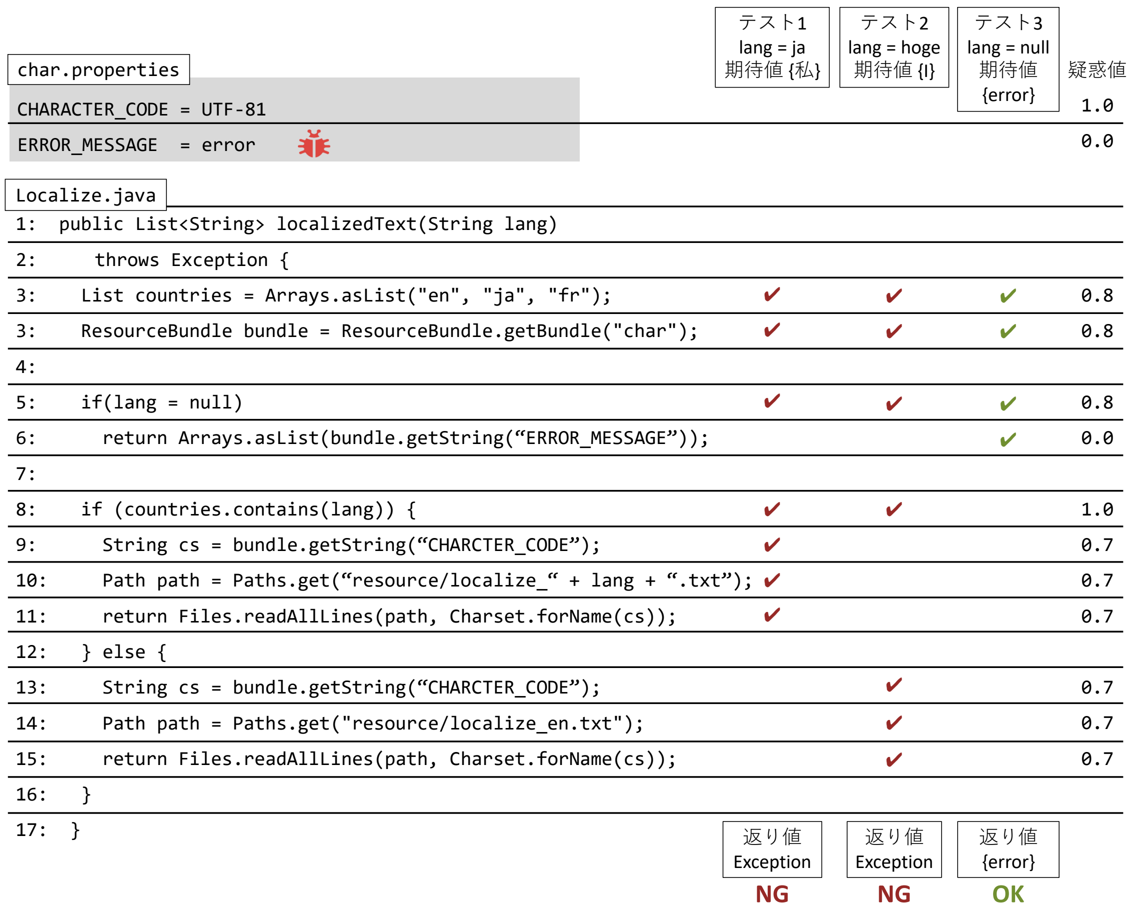
図 10　提案手法および既存手法による実行時間

次に，バグ原因箇所が Java のみであるバグに対する実験の結果を図 8 に示す．図 8 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．

次に，実行時間についての結果を図 10 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．

最後に，それぞれのバグのオーバーヘッドを図 9 に示す．提案手法のオーバーヘッドは最大で約

4.5%，最小で 0.3% であった．



(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 11 提案手法により順位が改善した例

7 考察

7.1 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 11 に示す．

図 11(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 11(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．図 11(a) を見ると，"CHARACTER_CODE" は 9 行目と，13 行目で読み込まれている．
2.1 で述べた疑惑値の計算式を再掲する．

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

式中の変数はそれぞれ以下の意味合いである．

$totalFail$:　失敗テストケースの総数．
$fail(i)$:　i 行を実行する，失敗テストケースの数．
$pass(i)$:　i 行を実行する，成功テストケースの数．

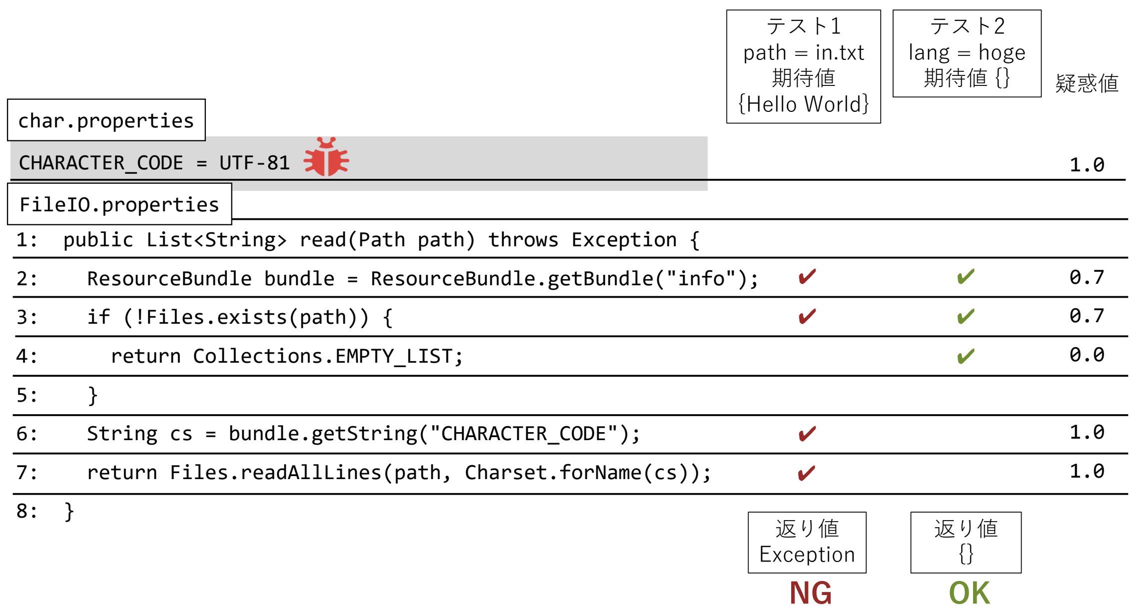
この式を見ると，失敗テストケースのみで実行された行であっても，その行を実行しない失敗テストケースが存在すると，疑惑値が低下することが分かる．そのため，"CHARACTER_CODE" を読み込んでいる 9 行目および 13 行目の疑惑値は，他の行と比べて低くなっている．その一方で，提案手法では，いずれの行でも "CHARACTER_CODE" の読み込み回数が加算されるため，バグを含む Properties のキーが最も高い疑惑値となっている．

このように，バグの原因箇所となっている Properties 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．

7.2　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 12 に示す．

図 12(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 12(a) のテスト 1 で読み込まれている，"in.txt" の中身を図 12(b) に示す．この例も，7.1 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．6.2 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 12 のような



(a) 対象のソースコード

```
In.txt
1: Hello World
```

(b) 入力ファイル

図 12　提案手法により順位が悪化した例

例では順位が悪化した．

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 13 に示す．

図 13 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 Properties 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，どの Properties 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，Properties 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．

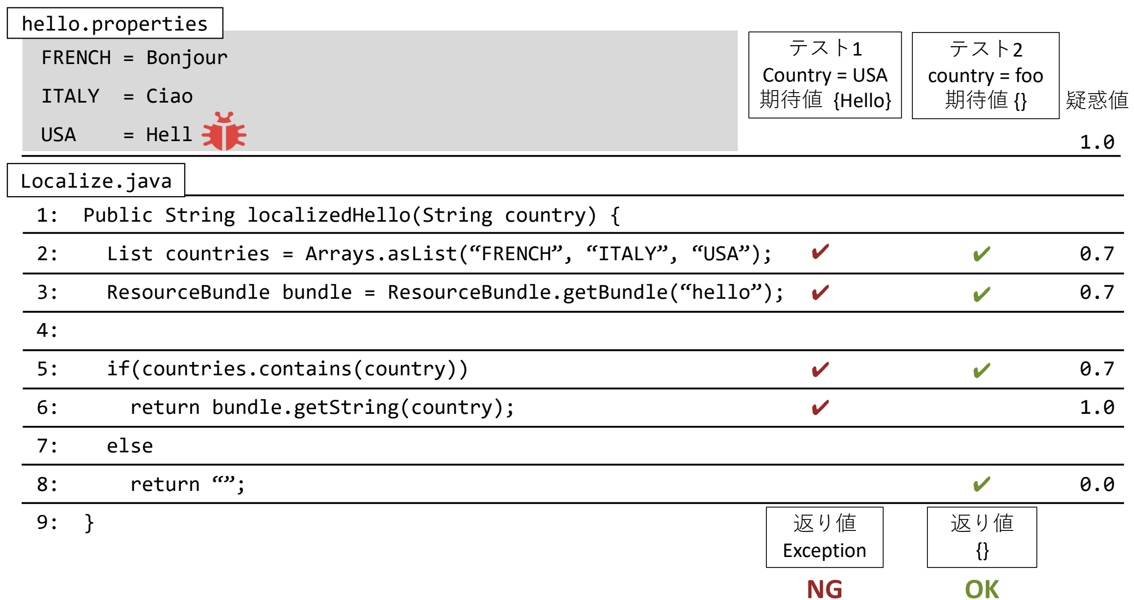


図 13　提案手法が有用な例

7.3　実行時間

図 10 及び図 9 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．実行時間を計測した工程は以下のとおりである．

- ソースコード加工
- ビルド
- テスト実行
- ソースコードのバグ限局
- Properties のバグ限局

調査結果を図 14 に示す．

図 14 を見ると，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，Properties 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 [42]

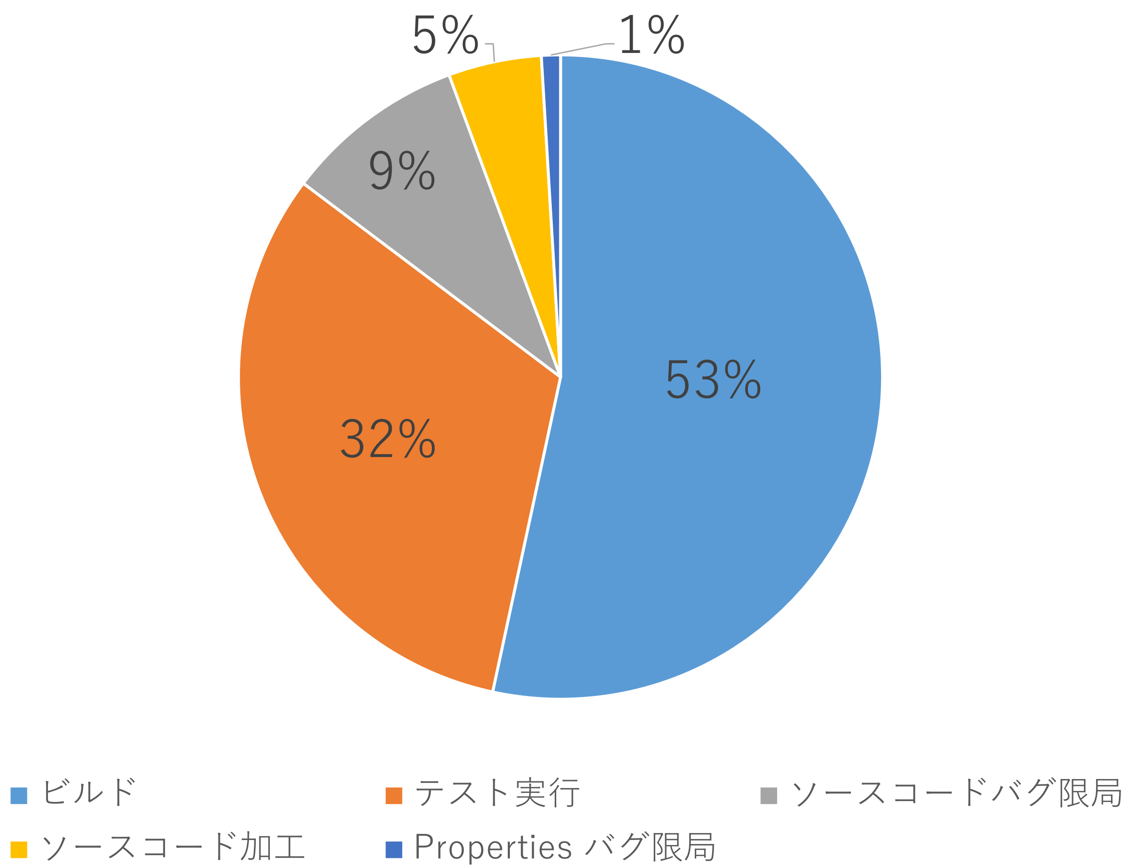


図 14　実行時間の割合

や，Pominville によるもの [43] 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．

8 妥当性の脅威

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 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．

9 おわりに

本研究では，自動バグ限局の既存手法を Properties 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 Properties 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．

10　謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました，桁本真佑助教に深く感謝申し上げます.

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に心より感謝致します.

本研究に至るまでに，講義やセミナー等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます.

最後に，本研究を行う上で，様々な資料やご指導を頂きました企業の皆様に心より感謝申し上げます.

参考文献

[1] W. Eric Wong, Vidroha Debroy, Adithya Surampudi, HyeonJeong Kim, and Michael F. Siok. Recent catastrophic accidents: Investigating how software was responsible. *International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pp. 14–22, 2010.

[2] NIST. Software errors cost u.s. economy $59.5 billion annually. http://www.abeacha.com/NIST_press_release_bugs_cost.htm.

[3] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[4] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers" . 2013.

[5] B. Korel. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering (TSE)*, Vol. 14, pp. 1253–1260, 1988.

[6] Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[7] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30, pp. 286–295. ACM, 2005.

[8] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)*, Vol. 42, No. 8, pp. 707–740, 2016.

[9] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 595–604. IEEE, 2002.

[10] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software engineering (ICSE)*, pp. 467–477. ACM, 2002.

[11] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of European Conference on Object-Oriented Programming (Ecoop)*, pp. 528–550. Springer, 2005.

[12] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based

fault localization. In *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART)*, pp. 89–98. IEEE, 2007.

[13] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 39–46. IEEE, 2006.

[14] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[15] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 439–449. IEEE Press, 1981.

[16] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software engineering (ESE)*, Vol. 7, No. 1, pp. 49–76, 2002.

[17] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, Vol. 40, No. 6, pp. 15–26, 2005.

[18] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10. ACM, 2002.

[19] delta.tigris.org. http://delta.tigris.org/.

[20] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 19, No. 04, pp. 573–597, 2009.

[21] Syeda Nessa, Muhammad Abedin, W Eric Wong, Latifur Khan, and Yu Qi. Software fault localization using n-gram analysis. In *Proceedings of International Conference on Wireless Algorithms, Systems, and Applications (WASA)*, pp. 548–559. Springer, 2008.

[22] Sai Zhang and Congle Zhang. Software bug localization with markov logic. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 424–427. ACM, 2014.

[23] Tristan Denmat, Mireille Ducassé, and Olivier Ridoux. Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering (ASE)*, pp. 396–399. ACM, 2005.

[24] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Jade-ai support

for debugging java programs. In *ictai*, p. 62, 2000.

[25] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 151–160. ACM, 2008.

[26] Maria Jump and Kathryn S McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN Notices*, Vol. 42, pp. 31–38. ACM, 2007.

[27] Kung Chen and Ju-Bing Chen. Aspect-based instrumentation for locating memory leaks in java programs. In *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Vol. 2, pp. 23–28, 2007.

[28] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pp. 155–164. IEEE, 2008.

[29] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 14–24. IEEE, 2012.

[30] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: better together. In *Proceedings of Joint Meeting on Foundations of Software Engineering*, pp. 579–590. ACM, 2015.

[31] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering (TSE)*, Vol. 33, No. 6, 2007.

[32] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering (ASE)*, pp. 234–243. ACM, 2007.

[33] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software engineering (ESE)*, Vol. 18, No. 2, pp. 277–309, 2013.

[34] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. In *Proceedings of International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pp. 189–208, 2013.

[35] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *Proceedings of International Symposium on Formal Methods (FM)*, pp. 187–201. Springer, 2012.

[36] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, Vol. 46, No. 6, pp. 437–446, 2011.

[37] David S Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, Vol. 9, No. 3, pp. 256–278, 1974.

[38] Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, Vol. 2, No. 4, pp. 299–306, 1998.

[39] Joao Marques-Sila and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pp. 171–182. Springer, 2011.

[40] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering (ASE)*, pp. 378–381. ACM, 2012.

[41] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pp. 159–172. ACM, 2011.

[42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pp. 214–224. IBM Corp., 2010.

[43] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, p. 8. IBM Press, 2000.