配列の未アクセス要素可視化による潜在バグ発見支援ツールの提案†

金谷　葵*

A tool for discovering latent bugs by visualizing unaccessed array elements

Aoi Kanaya

1　はじめに

ソフトウェアテストは，プログラムの品質と信頼性を保証する上で不可欠な工程である．しかし，テストコードに変更がないにも関わらず，実行するたびに成功と失敗を繰り返すFlaky Test（フレイキーテスト）は，テストの信頼性を著しく損なう深刻な問題[1]として広く認識されている．Eckらの調査では，開発者の59%が週次または日次でFlaky Testに対処していると報告されており[2]，これが単なる特殊なケースではなく，開発現場で頻繁に発生する課題であることがうかがえる．

Flaky Testが引き起こす問題の中でも，本来失敗して欠陥を検出するべきテストが偶然成功してしまう「見過ごされたアラーム(Missed Alarm)」は特に危険である[3]．この現象は，プログラムに存在する欠陥を長期間にわたって隠蔽してしまう可能性があり，実際，テストの実行順序に依存するバグが原因で，Apache CLIライブラリ[4]の欠陥が3年以上も発見されなかった事例も報告されている[3]．

このような見過ごされたアラームは，テストの実行順序だけでなく，単一テスト内で使用される入力データの内容の依存によっても発生する[3]．

本来であれば網羅的なテストデータを用意することで回避できるが，あらゆる組み合わせをテストすることはコストが高く，現実的ではない[5-7]．

そのため，特定の条件下でしか異常な出力とならないデータ依存のバグは，テスト設計時に見落とされやすく[7]，結果として「テストが成功したにも関わらず，欠陥がプログラム内に残存してしまう」という事態を招く．本研究では，このようなバグを「潜在バグ」と定義する．

本研究は，潜在バグの中でも特に入力データ依存に起因する「配列の未アクセスバグ」を対象とする．この種のバグは，特定の入力データ（例：配列の末尾が偶然最大値である場合など）においてはプログラムの出力が正常に見えるため，その発見は極めて困難である．Huoらも指摘するように，変数が使用されているにも関わらずその一部（特定の要素）のみがアクセスされないという微細な挙動の検出は難しい[6]．加えて，従来のデバッガやFault Localization技術の多くは，例外の発生やテストの「失敗」をトリガーとして分析を行うため，テストが成功しエラーログも出力されない状況下では，開発者に調査のきっかけ（起点）を提供することができない．

そこで本研究では，テスト成功時においても開発者にデバッグの起点を提供することを目的とし，入力データの内容に関わらずバグの根本原因となりうる「配列の未アクセス」という内部的な振る舞いを可視化するツール「Lacuna（ラクーナ）」を提案する．本稿では，Lacunaが開発者の潜在バグ発見と原因特定を支援できるかを定量的に評価した結果について述べる．

2　関連研究

テスト依存性の問題に対し，その検出を目的とした代表的な研究にZhangら[3]のものがある．彼らは，テストの実行順序を逆順やランダムに入れ替えることで，結果が変化するテスト（Order-Dependent Test）を特定するツールDTDetectorを提案した．このアプローチは，テスト依存性の存在を外部から特定する点で有効であるが，検出後の原因究明は開発者に委ねられる．

これに対し，依存性の検出だけでなく，デバッグそのものを支援するアプローチとして，実行トレースの可視化に関する研究[8]がある．Shimariらが提案したNOD4Jは，特に継続的インテグレーション（CI）環境など，インタラクティブなデバッガの利用が困難な状況を対象とする．NOD4Jは，実行トレースが肥大化する問題を，各命令ごとに記録する値の数を制限することで解決し，テスト失敗時のプログラムの振る舞いを効率的に調査する手段を提供する．

表1に本研究と関連研究のアプローチ比較を示す．本研究は，これらの先行研究が持つ課題を補完するものである．

---

†本研究の一部は以下において発表した
・日本ソフトウェア科学会第42回大会講演論文集 3c-1-R

*電子情報メディア工学専攻　2248004　橋浦研究室

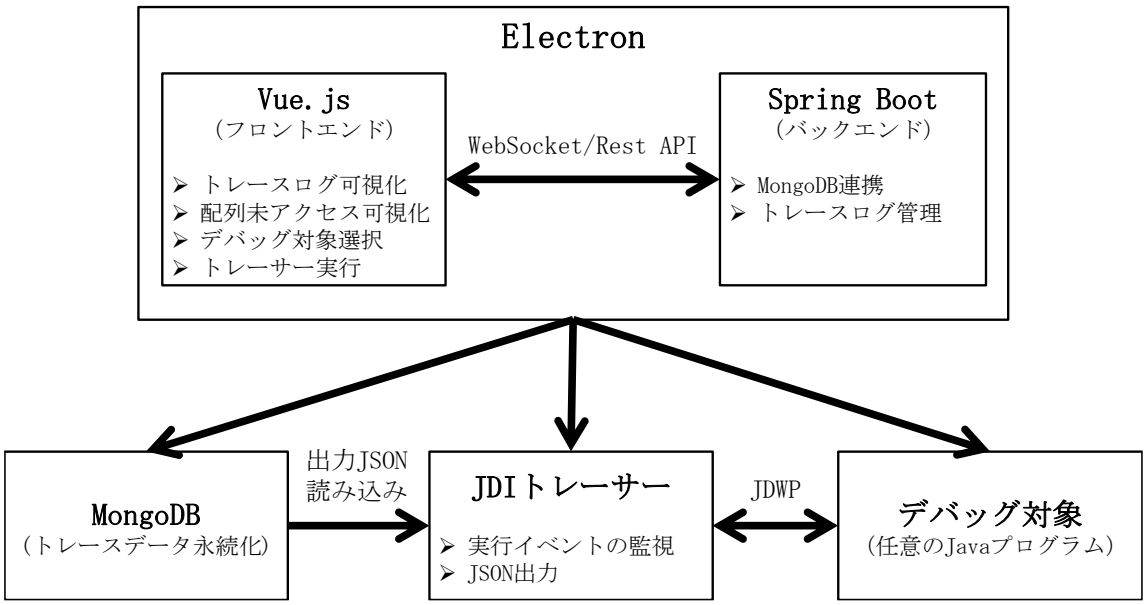

図 1：提案システムの全体構成

Zhang らのアプローチは依存性の有無を特定するが,出力が正常な「見過ごされたアラーム」のケースで，なぜそれが問題なのかという開発者の認知的な判断を支援するものではない．また，Shimari らの NOD4J は問題が顕在化した後の詳細な原因分析には強力だが，本研究が対象とする「テストは成功しているが，内部に欠陥が潜んでいる」という状況の問題発見のきっかけを積極的に提供することに特化しているわけではない．

表 1：関連研究とのアプローチの比較

#	研究	目的	対象状況	検出対象
1	本研究	潜在バグ発見支援（きっかけの提供）	テスト成功時	配列の未アクセス要素
2	Zhang ら	テスト依存性有無の検出	成功⇔失敗の変化時	テスト実行順序の依存性
3	Shimari ら	テスト失敗時の振る舞い分析支援	テスト失敗時	失敗時の詳細なプログラム挙動

本研究は，これらの中間に位置し，入力データ依存に起因する潜在バグ発見の最初のステップを支援する．すなわち，バグの兆候である「配列の未アクセス」という見えない情報を可視化することで,開発者に調査の起点を提供し，その後の原因特定プロセスへと繋げる点に独自性がある．

3　提案手法

本研究では，潜在バグの発見を支援するため，プログラム実行時の配列アクセス状況を追跡・可視化するツール「Lacuna（ラクーナ)」を提案する．

3.1　システム構成

提案システムの全体構成を図 1 に示す．本システムは，トレース取得を行う Electron アプリケーションと，データ蓄積・可視化を行う Web アプリケーション（Spring Boot + Vue.js）から構成される．

Lacuna のトレーサー部は，JDI（Java Debug Interface）による動的解析に加え，ソースコードの静的解析を併用するハイブリッド解析手法を実装している．実行行のソースコードを解析して arr[i+1] 等のインデックス式を抽出し，JDI から取得した実行時の変数値を用いて動的に評価することで，アクセスされた正確なインデックス値を算出・記録する．

3.2　デバッグ手順と可視化

デバッグの流れを 4 段階に分けて示す．具体例として，リスト 1 に示す最大値探索プログラムを用いる．このコードは，ループ条件の誤り「length - 1」により配列の末尾要素が参照されないバグを含んでいるが，配列の最大値 99 が偶然先頭にあるため，表面上の出力結果は正常となる「潜在バグ」の事例である．

リスト 1：バグを含む Java コードの例

```
public static void main(String[] args) {
  int[] numbers = {99, 15, 45, 67, 32};
  int max = numbers[0];
  for (int i = 0; i < numbers.length - 1; i++)
{
      if (numbers[i] > max) {
          max = numbers[i];
      }
  }
}
```

1.　プログラムの実行とトレース取得

まず，Lacuna のホーム画面（図 2）から「デバッグ」を選択し，トレーサー実行画面（図 3）へ遷移する．ここで対象の Java プログラムを選択して実行する．

2.　未アクセス要素の確認（制御パネル）

次に，ホーム画面（図 2）から,「ドキュメント」を選択し，制御パネルを確認する（図 4）．自動検出された未

図 4：本ツールの制御パネル

アクセス要素をここで把握する．

3. 分析範囲の絞り込み（抽出パネル）

問題の変数を特定した後，抽出パネルでその変数が使われている箇所に範囲抽出設定を行う（図 5）．

4. 原因特定（分析パネル）

範囲抽出設定後，メモリマップを確認する．ここでは，配列要素のアクセス状況を確認でき，アクセスされた要素は白枠で表示される一方，未アクセス要素は橙色でハイライト表示される（図 6）．このフィードバックにより，「末尾の要素だけが処理されなかった」ことに気づくことできる．

図 2：本ツールのホーム画面

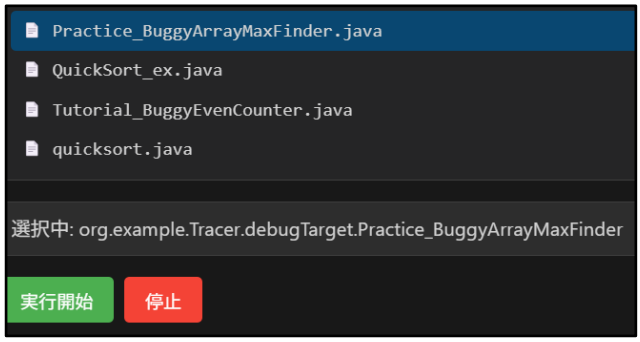

図 3：本ツールのトレーサー実行画面

```
5     public static void main(String[] args) {
6         int[] numbers = {99, 15, 45, 67, 32};
7         int max = numbers[0];
8         for (int i = 0; i < numbers.length- 1 ; i++) {
9             if (numbers[i] > max) {
10                max = numbers[i];
11            }
12        }
13        System.out.println("Maximum value: " + max);
14    }
15 }
16
```

図 5：本ツールの抽出パネル

図 6:本ツールの分析パネル

4　実験評価

本実験では，本ツールが従来のデバッガより潜在バグの発見に有効かを検証するため，RQ「出力異常がなくてもツール利用でバグに気づけるか？」を設定した．被験者として Java 学習経験のある学生 20 名を，実験群と統制群（各 10 名）に分け，4 種類の潜在バグを含む課題プログラムのデバッグを課した．課題には，マージソート，クイックソート，統計値計算，累積和計算の 4 種類の Java プログラムを用いた．これらには全て，配列の末尾要素が処理対象から漏れるという，潜在バグが意図的に埋め込まれている．

これらのバグを隠蔽し，「テストは成功するが内部に欠陥が潜んでいる」という状況を再現するため，Java エージェントを利用してプログラムの動作を操作し，提供した入力データでは出力が正常に見えるようにした．

実験手順として，被験者にはバグの有無を知らせずにプログラムの調査を依頼し，その過程と発見事項を事前に用意した報告書に記述し提出するように指示した．タスクの達成度は，被験者が提出した報告書を，表 2 の項目と表 3 に基づき採点した．そのうえで「RQ 達成」の判断は，バグ修正できた被験者は表 2 の項番「2, 4, 5, 6」が B 評価以上であること，バグ修正できなかった被験者は項番「1, 2, 3, 4」が B 評価以上であることを基準とした．

表 2：デバッグタスクの評価項目

#	フェーズ	No.	評価項目
1	現象の把握と再現	1. 1	問題の検知
2		1. 2	具体的な現象の記述
3		1. 3	現象再現の報告
4	原因範囲の特定	2. 1	関連関数の特定
5		2. 2	原因コード行の特定
6	原因メカニズム理解	3. 1	誤りの直接原因報告

表 3：回答の明確性に関する評価基準

#	評価	定義
1	A	評価者が再現できるほど明確に回答
2	B	評価者が再現する際に迷いが生じる回答
3	C	言及がない・問いに回答できていない

5　考察

実験結果は，本ツールが従来手法に比べ，潜在バグ発見を

統計的に有意に支援（$p < 0.01$）することを示した（表 4）．

表 4：バグ検出率（RQ 達成数）と群間比較

#	課題	実験群（ツールあり）	統制群（ツールなし）	p 値
1	マージソート	40%（4/10）	10%（1/10）	0.303
2	統計値計算	30%（3/10）	0%（0/10）	0.311
3	クイックソート	30%（3/10）	10%（1/10）	0.582
4	累積和計算	70%（7/10）	20%（2/10）	0.07
5	全体	42.5%（17/40）	10.0%（4/40）	0.002

実験群のバグ検出率（42.5%）は統制群（10.0%）を 4 倍以上上回り，この差は統計的にも有意であった（$p < 0.01$）．これは，出力が正常に見えるという手がかりのない状況下で，本ツールが「未アクセス要素の可視化」によって，これまで発見が困難だった問題への調査の起点を提供したことを意味する．

しかし，同時に実験群の検出率が半数に満たなかったという事実は，本ツールの有効性を最大限に引き出す上での課題を浮き彫りにしている．この原因を明らかにするため，バグを発見できなかった被験者の行動を分析した（表 5）．

表 5：バグ検出を妨げた要因と観測人数（実験群 N=10）

#	要因の分類	具体的な行動・事象	観測数
1	誤った焦点	ツールが警告を出しているメソッドや変数ではなく，関係のない変数に固執して分析を続けた	6 名
2	不注意	時系列に並んでいる配列情報を最後まで確認せず，末尾の未アクセス要素を見落とした	5 名
3	確証バイアス	プログラムの出力が正常であることを過信し，ツールの警告を誤検知や仕様であると解釈した	1 名

その結果，最も頻発したつまずきは「誤った焦点」（6 名）や「不注意」（5 名）といった被験者に起因する問題であることがわかった．具体的には，ツールが未アクセス箇所を提示しているにも関わらず，関係のない変数に注目し続けたり，ログ情報を最後まで確認しないことで，決定的な証拠を見逃していた．

また，一部の被験者は，出力が正常であることから，ツールの未アクセス要素の指摘をバグとして認識しない事例も確認できた．

加えて，UI の視認性（横スクロールの見逃し）や，バグではない未アクセス箇所も表示されることによる情報の過多といったツール側の課題も，被験者の混乱を招き，原因特定を妨げる一因となっていた．

それに対しバグを発見できた被験者は，ツールの提示する未アクセス要素に注目し，ログ情報を丁寧に確認し，関連する変数を意識的に追跡していた．具体的には main 関数から処理を追い，文脈に合わせて注目変数を変更することで，1 つ 1 つ分析を積み重ね，バグ発見につながっていた．

これらの結果は，ツールがバグの″きっかけ″を提示しても，利用者がその情報を正しく解釈し，活用する能動的なデバッグ戦略を取らなければ，原因特定には至らないことを示唆している．以上を踏まえ本ツールの価値は，「不可視な問題の可視化」によって，初学者・熟練者問わず，調査の起点をすべての利用者に提供する点にある．

今後の課題は，配列の未アクセスを可視化することによる「現象の提示」から，その可視化が本質的に何を示しているのかという「原因理解」への認知的なギャップを埋めることである．そのためには，ツールの UI 改善や，ログ情報のフィルタリング・要約による情報提示の最適化が求められる．さらに，ツール内に効果的な利用手順をガイドする機能を組み込むなど，利用者の思考プロセスを積極的に導く仕組みを整備することが，本ツールの潜在能力を最大限に引き出し，検出率を向上させる上で不可欠であると認識している．

参 考 文 献

1) O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," ACM Trans. Softw. Eng. Methodol., vol. 31, no. 1, pp. 17:1–17:74, Oct 2021.
2) M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," in Proc. 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 830–840, Aug 2019.
3) S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically Revisiting the Test Independence Assumption," in Proc. 2014 International Symposium on Software Testing and Analysis (ISSTA), pp. 385–396, Jul 2014.
4) Apache Software Foundation, "Apache Commons CLI," [Online]. Available: https://commons.apache.org/proper/commons-cli/, Jan 2026. (Accessed: 2026-01-05)
5) M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly Detecting Relevant Program Invariants," in Proc. 22nd Int. Conf. Softw. Eng. (ICSE), pp. 449–458, Jun 2000.
6) C. Huo and J. Clause, "Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests," in Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE), pp. 621–631, Nov 2014.
7) P. McMinn, "Search-Based Software Test Data Generation: A Survey," Softw. Test. Verif. Reliab., vol. 14, no. 2, pp. 105–156, Jun 2004.
8) K. Shimari, T. Ishio, T. Kanda, N. Ishida, and K. Inoue, "NOD4J: Near-Omniscient Debugging Tool for Java Using Size-Limited Execution Trace," Sci. Comput. Program., vol. 206, pp. 1–13, Jun 2021.

指導教授	審査委員（主査）	准教授	橋浦　弘明
	審査委員（副査）	教授	粂野　文洋
	審査委員（副査）	准教授	加藤　利康