# 記号実行を用いた改善余地のあるコードの検出に向けた実行時エラー分類手法

佐々木 裕介[1]

**概要**：品質の低いコードは, ソフトウェアの拡張性や保守性に関する問題を起こす. 品質の低いコードを検出するために静的解析ツールが使われているが, コードの品質向上にかけられる時間はプロジェクトにより異なる. 例えば, (1) プロジェクトが確保できる時間に余裕があり, さらに品質を向上したい場合と, (2) プロジェクトが確保できる時間に対して静的解析ツールが提示する警告が多すぎる場合がある.
そこで本稿では (1) 改善余地のあるコードを検出するため, 静的解析ではなく記号実行器で生成したテストケースで生じた実行時エラーを使えないか分析し, さらに有用な実行時エラーを見分ける手法を提案する. さらに (2) 発現の可能性が高い警告を見つけるため, 記号実行器を使い, 静的解析の警告する振る舞いをテストケースとして実行できるかについても確認する.
実験の結果, (1) 実業務で使われているウェブシステムを対象としたところ, 記号実行器により得られた実行時エラーの中に, 開発者にとって有用な気づきを与えるものがあることを確認した. また, 有用な気付きを与えるエラーを見分けるためのシンプルなメトリクスを提案し, メトリクスの有用性を計測するための定量的評価を実施した. (2) 実業務で使われているウェブシステムを対象とした実験を行い, 静的解析による 4,688 件の警告のうち 6 件を記号実行で生成したテストケース中で実行できた. また他の OSS プロジェクト 2 件についても同様の方法で実行できる警告があることを確認した.

# Runtime error evaluation towards detection for code to be improved using symbolic execution

## 1. はじめに

品質の低いコード, すなわち実行時エラーを起こしやすいコードや, 開発者にとって理解の難しいコードは, ソフトウェアの拡張性や保守性に関する問題を起こす [13] . そのため, 品質の低いコードを検出し継続的に改善したい. 品質の低いコードを検出するために静的解析ツールが使われている [18] . しかしコードの品質向上にかけられる時間はプロジェクトにより異なる：

- 状況 1 . プロジェクトが確保できる時間に余裕があり, さらに品質を向上したい. この場合, 品質向上のために提示する警告をさらに追加したい.
- 状況 2 . プロジェクトが確保できる時間に対して静的解析ツールが提示する警告が多すぎる. この場合, 信頼性の高い警告のみを残して警告件数を減らしたい.

このうち状況 1 に関連する研究として, Artemis[6] や Veritesting[5] は, 静的解析ではなく記号実行器の生成するテストケースで生じた実行時エラーを利用して, 潜在的なバグ（null 参照やゼロ割エラーなど）を見つけることを試みている. 一方で, 記号実行器でのテスト生成には問題がある. テスト対象メソッドにはしばしば, 隠れた事前条件が存在する. 例えば：

- 本来ならテスト対象メソッドへ引数で渡す Map オブジェクト “accounts” は, 特定のキー “users” を含んでいなければならない. しかし記号実行器で生成したテストケースでは本条件を満たせずに null 参照エラーを起こす（Listing 1 testee1）
- 本来ならテスト対象メソッド “invoke” を呼ぶ前に, 開発者はテスト対象内のフィールド “skeleton” を初期化する別のメソッド prepare を呼ばなければならない. しかし記号実行器で生成したテストケースでは本条件を満たせずに null 参照エラーを起こす（Listing 2）

[1] （株）富士通研究所
Fujitsu Laboratories Ltd.

Listing 1: 検出したい / 検出対象外としたいメソッド例

```
public class Target {
    private List<Map<String, String>> users;

    // precondition: accounts should have the value of "
        users" key
    public void testee1(Map<String, List<Map<String,
        String>>> accounts) {
        FW_Component component = FW_Component.createModel
            ();
        UserUtils utils = new UserUtils();
        List<Map<String, String>> users = accounts.get("
            users");
        utils.sort(users);
        // null pointer exception if users == null
        for (Map<String, String> user : users) {
            String name = user.get("name");
            if (component.getDefaultElements().containsKey
                (name)) {
                // ...
            }
        }
        this.users = users;
    }

    // precondition: none
    public void testee2() {
        FW_Component component = FW_Component.createModel
            ();
        // class cast exception if getDefaultObject does
            not return the string type value
        String cstring = (String) component.
            getDefaultObject();
    }

    // precondition: none
    public void testee3() {
        FW_Component component = FW_Component.createModel
            ();
        String source = component.getDefaultDateString();
        // illegal format exception if source is not
            yyyyMMdd format.
        Date date = DateFormat.getDateInstance().parse(
            source);
    }

    // precondition: utils should be set the yyyyMMdd date
        format value.
    public void testee4(UserUtils utils) {
        String source = utils.getDateString();
        // illegal format exception if source is not
            yyyyMMdd format.
        Date date = DateFormat.getDateInstance().parse(
            source);
    }

}
```

Listing 2: 呼ぶ前に気をつけるべき暗黙の了解があるメソッド実例

```
package org.springframework.remoting.caucho;
// ..
public class BurlapExporter extends RemoteExporter
    implements InitializingBean {
  private BurlapSkeleton skeleton;
  // ..
  public void prepare() {
    // ..
    this.skeleton = new BurlapSkeleton(getProxyForService
        (), getServiceInterface());
  }
  public void invoke(InputStream inputStream, OutputStream
      outputStream) throws Throwable {
    // ..
    try {
      this.skeleton.invoke(new BurlapInput(inputStream),
          new BurlapOutput(outputStream));
    } finally {
      // ..
    }
  }
}
```

仮にコード解析技術だけを使ってテストドライバやモックを自動生成する場合（Monpratarnchai[14] ら），生成するテストドライバやモックがこのような事前条件を満たすのは困難である．特にテスト対象メソッドに存在する隠れた事前条件が複雑である場合，記号実行器の生成するテストケースは，実行時エラーを引き起こす．一方，そのテスト対象メソッドに詳しくない開発者，例えば新しくアサインされた開発者にとっても，隠れた事前条件を考慮してメソッドを呼ぶのは難しい．本稿ではこの点に着目し，隠れた事前条件の存在するメソッドを探すために有用な実行時

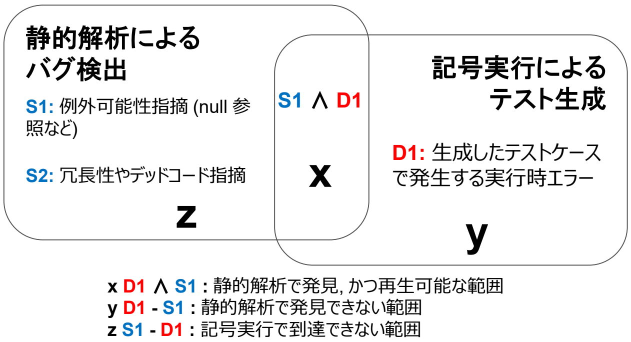


図 1: 静的解析ツールおよび記号実行による検出対象

エラーが，テストケースの実行時エラーの中に存在するか確認し，改善余地のあるコードを検出する．具体的には次の“開発者にとって有用な情報をもたらす”実行時エラーを検出する．

- 実行時エラーが発生することを防ぐために，“テスト対象コードに改善の余地がある”と開発者が気づくきっかけを与える
- 隠れた事前条件が存在する等の理由で“テスト対象を呼ぶ前に気をつけるべき暗黙の了解がある”と開発者が気づくきっかけを与える

検出したコードを改善するためのアクションとして，次の２つを開発者へ提示することを想定している．もし静的解析にできない指摘を開発者へ提示できるなら，より高い品質を目指すことができる．

- 実行時エラーを防ぐためのコード修正．例えば，エラーハンドリングを行うコードを追加する．
- Java プログラムを対象とする場合，検出されたコードを内包するメソッドの Javadoc コメント内へ，警告を追記する．具体的には，呼び出す前に特定の事前条件を満たす必要がある旨を，メソッドの利用者へ知らせる．例えば，“メソッドのマップ型引数 accounts に，メソッド内で参照される users キーを含めるべきである”ことを知らせる（Listing 1 testee1）．

状況２については，静的解析の誤検出率を減らすために，動的解析を用いて信頼性を向上しようとする研究が既に存在する [8]．この研究はバグを検出することを目的として組み立てられたものであるが，既存の静的解析ツールと記号実行器の出力を単純に組み合わせるだけでも，信頼度の高い警告を得られるのではないかと考えられる（図 1 x）．そこで本稿では，警告が指摘する振る舞いを，自動生成するテストケースにおいて再生できる場合のみ，その警告を開発者へ提示する．以降は“警告が指摘する振る舞いを，テストケースで再生する”ことを“ **警告を再生する** ”と表記する．テストケースとして再生できる警告は，誤検出でなく実際に発現する危険性が高い．さらにテストケース実行により発現時の具体的な実行経路や入力値を示すことも出来るため，開発者が理解しやすく対処も容易になる．

本稿における研究課題を次のように定式化する.

- RQ.1 図 1 y 部分について, 記号実行器の生成するテストケースが引き起こした実行時エラーは, 本当に開発者に有用な情報をもたらすのか. 実プロジェクトを対象とし, 記号実行器が生成するテストケース実行中に発生したエラーのうち, 開発者にとって有用な情報をもたらすと判断できる実行時エラーは何件あるか分析を行う.
- RQ.2 もし開発者に有用な情報をもたらすエラーがあるとすれば, それらを簡単な方法で効率的に見つけることはできるか. 本稿で検出しようとしている有用な実行時エラーを見つけるための, 定量的な方法は存在するか検討する.
- RQ.3 図 1 x 部分について, 記号実行器で静的解析の警告を実行時エラーとして検出可能であるのか. 実プロジェクトを対象とし, 静的解析ツールによる警告のうち, 記号実行器が生成したテストケースによって実行時エラーとして再生できる警告は何件あるか分析を行う.

本稿の主な貢献を以下に示す.

- RQ.1 について, 改善余地のあるコードを検出するために, 記号実行器で生成したテストケースの実行中に生じた実行時エラーを使えるか否か評価した. 記号実行器によるテストケースを実行する際に生じるエラーの一部は, 開発者にとって有用な気づきを与えることがある.
- RQ.2 について, テストケースを実行して発生したエラーが有用か否か判断するためのメトリクス算出方法を提案した. 有用な実行時エラーを見つける上で, 提案するメトリクスを有効利用できるか否か定量的評価を実施した. 本稿で提案するテストケース実行時エラーを評価するメトリクスは, 設定する閾値次第では有用である. また, 実行時エラーの種類を確認することでそのエラーが有用であると判定できる場合があることを確認した.
- RQ.3 について, 静的解析ツールによる警告をより信頼度の高いものとするため, 記号実行器を使い, 警告をテストケースとして再生できるか否か評価した. 静的解析ツールによる警告の一部は, 記号実行器の生成するテストケースによって再生可能である.

## 2. 関連研究

### 2.1 静的解析を用いたバグ検出

改善余地のあるコードを検出する技術の実装として, 例えば Findbugs[9] や ESC/Java2[7] , OpenJML[3] , SonarQube[4] , Infer[1] 等の OSS ツールが存在する. 例えば Findbugs[9] は null 参照やゼロ割, 非互換の型キャスト, 配列サイズが負であるコード等を探し出し警告する. OpenJML[3] は適合率を上げるため, 動的解析も利用している. コードに出現する単語の naturalness に基づくバグ検出も研究されている [15] .

しかし静的解析ツールは, 多くの誤検出があるという問題を抱えている [8] . 誤検出をフィルタする研究として, 例えば BugMem[12] はバグ修正履歴を使い, 警告の適合率を改善している. DSD-Crasher[8] は静的解析と動的解析を組み合わせてバグの検出を行っている.

### 2.2 記号実行器を用いたテスト生成

Artemis[6] や Veritesting[5] は, 記号実行器の生成するテストケースで生じた実行時エラーを利用して, 潜在的なバグ（null 参照やゼロ割エラーなど）を見つけることを試みている. Artemis の目的はバグの検出であるが, 我々の興味は開発者にとって有用な情報をもたらすと判断できる実行時エラーがあるか否かである. TesMa and CATG[17] は設計書の情報を基に conclic testing を行うことで, 人間が想定する範囲での動作を確認するテストケースを生成する.

## 3. システム概要

Java で書かれたプログラムのクラスファイルを入力として, テストケースと実行時エラーの組を出力するシステムを実装した. 本システムの出力は図 1 に示す x, y に相当する. 図 2 にシステム概要を示す. 実装したシステムは次の処理を順に実施する.

- モックファイル生成処理. 入力プログラム中に定義されているメソッドを呼び出すドライバと, それらをコンパイルするために必要なモックファイル群を生成する.
- JPF[2] を用いた記号実行処理. テスト対象コードのカバレッジを上げるため, 先に生成したテストケースを使い, テストケースの数を増やす. 記号実行器の制限のため, スレッドプログラミングはスコープ外である.
- テストケース実行処理. 生成した全てのテストケースを Java VM 上で実行し, 結果を要約する.
- 実行時エラーの評価処理. テスト実行時に発生した各々の実行時エラーに対し, 有用度合を評価するためのメトリクスを算出する. 本処理により図 1 y に当たる部分が評価できる.
- Findbugs による静的解析. 本システムは Findbugs を使って, コードに対する警告と, 警告対象コードの行番号を取得する. 数ある静的解析ツールの中から Findbugs を選択する理由は, 前節の既存研究で比較の対象とされており, かつ導入が容易であるためである.
- マージ処理. Findbugs による警告と, テストの実行結果をマージする. Findbugs が警告している行番号と, 実行時エラーの発生した行番号が一致していれば, その行番号と, Findbugs の警告および発生した実行時エ

ラーの内容を出力する．本処理により図 1 x に当たる部分が検出できる．

### 3.1 モックファイル生成処理による参照先の置換

ベースとなるテストケースを生成する．各テスト対象クラスに対し，public もしくは protected メソッド 1 つにつき，1 件のドライバを生成する．同時に，コンパイルに必要なモックファイル群を生成する．

テストしたいクラスが依存するライブラリやフレームワークに，例えば設定ファイルや業務データベース，環境変数などを参照する処理があると，記号実行器がその部分をエミュレートできず，動作停止する問題がある．特にフレームワークは初期化の際に大量の設定を読み込むことが多いために，記号実行器によるエミュレーションが，テストしたいクラスのロジックへ到達する前に，動作停止する問題が頻繁に起こる．初期化で動作停止するとカバレッジは最悪 0%となる．そこで我々は，テスト対象クラスが参照する全てのクラスに対してソースファイルを自動生成して置き換え，コンパイルし直すツールを開発した．バグを検出することを目的として Islam[10] らも類似する方法を採用している．

### 3.2 実行時エラーの評価処理

テスト実行時に発生した各々の実行時エラーに対して，有用度合を評価するためのメトリクスを算出する．

記号実行器の生成するテストケースは，開発者に提示するべきでない余分な実行時エラーも引き起こす．例えば，あるフレームワークのコード（Listing 3）を参照している，本システムの利用者が開発したコード（Listing 4, Listing 1）を考える．Listing 1 がテスト対象である．これらは実際の例をかなり簡略化してある．

Listing 3: モック化したフレームワーク側プログラム例

```
public class FW_Component {
  public static FW_Component createModel() {
      return new FW_Component();
  }

  public Map getDefaultElements() {
      return new HashMap();
  }

  public Object getDefaultObject() {
      return "";
  }

  public String getDefaultDateString() {
    return "";
  }
}
```

Listing 4: モック化した本システムの利用者が開発対象としているプログラム例

```
public class UserUtils {
  public void sort(List<Map<String, String>> users) {

  }
  public String getDateString() {
      return "";
  }
}
```

自動生成されたテストケースにおいて，エラーが発生する原因はおおよそ以下の 3 つに分類できる．

- 引数或いはモッククラスが本来の挙動を忠実に再生していない．次の 2 通り．
  – モック化の対象が，本システムの利用者が手を入れることのないフレームワークやライブラリ．現実にはありえないテストデータをモックから入力している．例えば testee3 (Listing 1) から Listing 3 に示すモック getDefaultDateString を呼ぶと，本来フレームワークが返す日付形式で文字データが受け取れず testee3 内 parse 処理でエラーとなる．testee2 (Listing 1) の場合は，Listing 3 に示す getDefaultObject の返す型が本来フレームワークの返す型と異なるためエラーが起きる．本ケースは実際にはありえないため，コードの改善には使えない．エラーの有用度は低いと判断したい．
  – モック化の対象が，本システムの利用者が開発対象としているプログラム．例えば既存の開発プロジェクトへ新しくアサインされた開発者が，テスト対象のメソッドが前提としている暗黙の了解（例えば事前条件）を知らずに，モックと同様の単純な呼び出しコードを書く可能性がある．Listing 1 に示す testee1 はモックである Listing 4 を呼ぶが，sort 呼び出し結果が null になるとエラーが発生する．Javadoc の改善等により Listing 4 の使い方や Listing 1 のコードについて開発者へ注意を促すことが期待できる．エラーの有用度は高いと判断したい．
- テスト対象のコードに欠陥がある．例えば Listing 1 へ示すコードに異常があるケースはエラーの有用度は高いと判断したい．

そこで，エラーの発生原因となるデータを生成しているのがフレームワークやライブラリの動作を模したモックなのか（Listing 3），それ以外か（Listing 4）を区別するため，各エラーを評価するメトリクス値を算出する．本稿では，実装の簡易さを優先して，テストケース実行時のメソッド呼び出し履歴だけを使って，メトリクスを算出する方法を提案した．計算方法を以下に示す．

- 例外発生直前で呼ばれたモックがフレームワークを模したものか否かで重みをつける．フレームワークか否かはパッケージ名で判別する．
- モックが呼ばれた位置から，例外発生までの距離に応じて，重みをつける．
- 呼出しごとに上記の重みの積を取って加算する．エラー発生直前で呼ばれるモックにフレームワークを模したものが多いと，算出する値は小さくなる（図 3）．

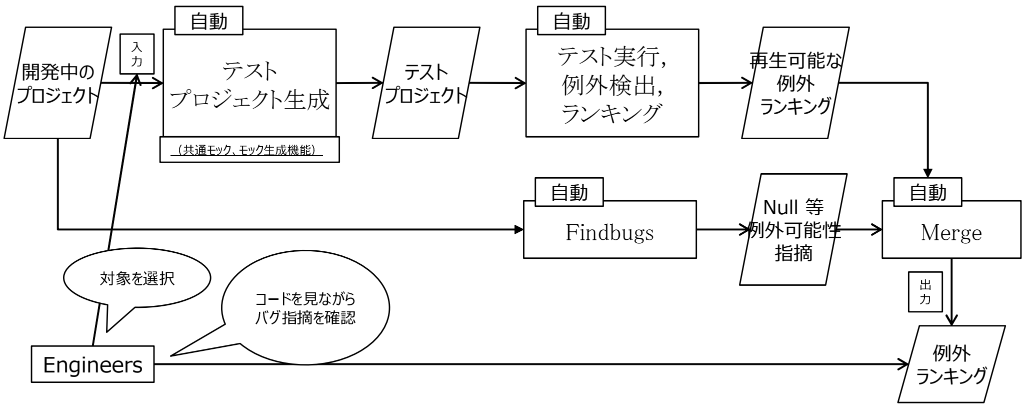


図 2: システム概要

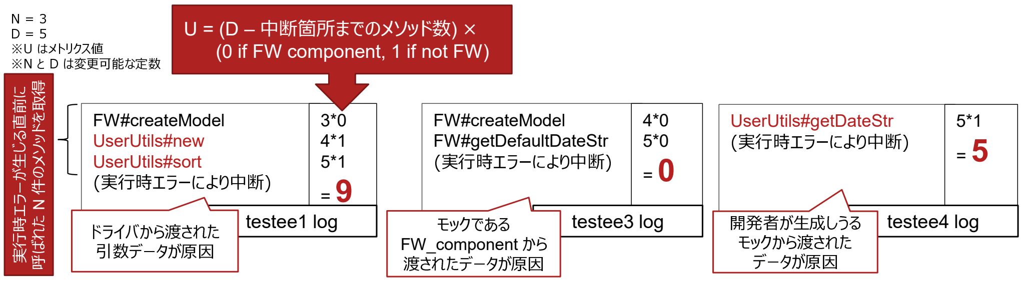


図 3: 実行時エラーを評価するメトリクスの算出方法

## 4. 評価実験

### 4.1 調査対象

- A. ある企業内で利用しているウェブシステム. 全てのクラスを対象とする.
- B. Maven 公式リポジトリ内の OSS. クラスファイルが提供されており JDK7 でソースファイルがビルド可能である OSS. 特に Findbugs によって任意の警告が出るクラスのみを対象とする.
- C. defects4j[11] に収録されている OSS. 特にバグを含むクラスのみを対象とする.

対象 B, C については RQ.3 のみについて評価を実施した.

### 4.2 RQ.1 記号実行器の生成するテストケースが引き起こした実行時エラーの分析

ある企業内で利用しているウェブシステムに対し, 記号実行器が生成したテストケース実行中に発生したエラーは 6,022 件である. さらに, 各々の実行時エラーに対する評価メトリクス（図 3）を算出した. 生成したテストケースで発生した実行時エラーのうち, 評価メトリクスの各値についてエラー数ができるだけ均等になるように 206 件をランダムサンプリングした上で, 有用か否かを開発者の手で判断した（図 4, 5）. 206 件の実行時エラーを 1 人で調査するために約 28 時間かかった. 判断結果が妥当か否かについて 2 人で精査した. 少なくとも Java での開発経験が 5 年以上あり, 数万行のプログラムからなるツールを構築するスキルのある開発者が調査を担当した.

図 4 の横軸は実行時エラーに対して算出したメトリクスの値で, 縦軸はそのメトリクス値をもつ実行時エラーが何件あったかを示す. 図 4 では改善の余地があるコードに気づくために有用なエラー件数と有用でないエラー件数を色分けで示している. 図 5 も同様で, こちらは呼ぶ前に気をつけるべき暗黙の了解があるメソッドに気づくために有用なエラー件数と有用でないエラー件数を色分けで示す.

- “テスト対象コードに改善の余地がある” と気づくために有用な実行時エラーの数は, 206 件の内 47 件であった. いずれも Findbugs が警告していないものである.
- “テスト対象メソッドを呼ぶ前に気をつけるべき暗黙の了解がある” と気づくために有用な実行時エラーの数は, 206 件の内 69 件であった. いずれも Findbugs が警告していないものである.

### 4.3 RQ.2 開発者に有用な情報をもたらすエラーを効率的に見つけることはできるか

実際に開発者がメトリクス値を使う際には, メトリクスの性質に応じて閾値を設定する. 例えば “呼ぶ前に気をつけるべき暗黙の了解があるメソッド” を開発者が見つけたいとする. 閾値を仮に 14 とすると, 開発者はメトリクス値が 14 以上となる全ての実行時エラーについて確認を行う. すなわち, 図 5 においてメトリクス値 15 となる 18 件と, メトリクス値 14 となる 14 件のエラーを目視で確認す

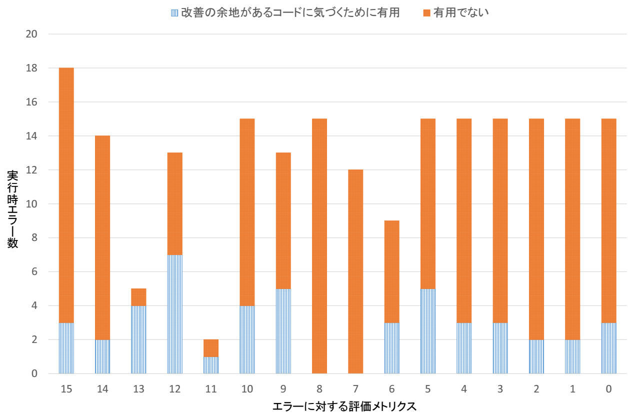


図 4: 評価メトリクス値ごとに見た, 改善の余地があるコードに気づくために有用なエラー件数

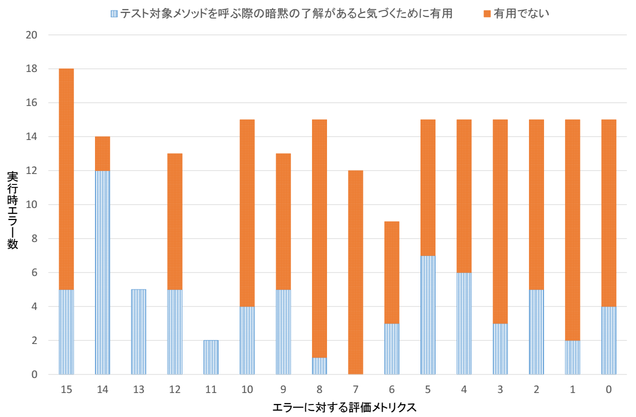


図 5: 評価メトリクス値ごとに見た, 呼ぶ前に気をつけるべき暗黙の了解があるメソッドに気づくために有用なエラー件数

る．開発者が確認する範囲に含まれる有用なエラーの数が少ないと, 開発者は有用でない実行時エラーを確認するために多くの時間を費やす．そこでカイ二乗検定を使い, メトリクスが閾値以上のエラーを確認する場合と, 全てのエラーの中から同数をランダムに確認する場合とで, 見つかる有用なエラーおよび有用でないエラー数に, 統計的に有意な差が出るのか確認した．帰無仮説は, 見つかる有用なエラーおよび有用でないエラー数に, メトリクスの使用有無で差が存在しないとみなせることである．

RQ.2 で調査したエラーの内, 呼ぶ前に気をつけるべき暗黙の了解があるメソッドに気づくために, 有用であると判断したエラーの占める割合は約 33.5% である（表１）．メトリクス値の高いエラーから順に確認する場合, 閾値を 13 としたときに開発者の確認するエラー数は 37 個, うち 22 個が有用なエラーで 59.4% 程度となる．全てのエラーの中から 37 個をランダムに確認する場合, 有用なエラー数の期待値は約 12 個となる．このとき p 値は 0.000819086 であり, 想定する母集団の大きさを十分大きくした場合に有意水準 95% かつ許容誤差 ±16% で統計的有意差があると判断できる．つまり閾値を 13 とすると, メトリクスを利用することで開発者が確認するエラーの内, 有用なエラーの割合を増やし有用でないエラーの割合を減らすことが出来る．一方で閾値 15 とすると, 開発者が確認するエラー数は 18 個, 有用なエラーの占める割合は約 27.8% で, ランダムに確認する場合の約 33.5% と比べて低くなる．つまり閾値 15 でメトリクスを利用すると, 有用なエラーの発見効率がランダムに確認する場合に比べて下がる．メトリクス値 15 だが有用でないエラーには testee2 (Listing 1) に類似するケースがあった．この例では, モック化した FW_Component の返すオブジェクト型によって ClassCastException が生じる．モック化されていない getDefaultObject は String 型を常に返す仕様であり, 本来なら ClassCastException は起こりえない．

同様に, RQ.2 で調査したエラーの内, 改善の余地があるコードに気づくために, 有用であると判断したエラーの占める割合は約 22.8% であった．メトリクス値の高いエラーから順に確認する場合, 閾値を 11 としたときに, 開発者の確認するエラー数は 52 個, うち 17 個が有用なエラーで約 32.7% となった．一方, 全てのエラーの中から 52 個をランダムに確認する場合, 有用なエラー数の期待値は約 12 個となった．このとき p 値は 0.089656284 となり, 想定する母集団の大きさを十分大きくした場合に有意水準 90% かつ許容誤差 ±11% で統計的有意差があると判断できる．一方で閾値 15 のとき, 開発者が確認するエラー数は 18 個, 有用なエラーの占める割合は約 16.7% で , ランダムに確認する場合の 22.8% と比べても低かった．本節の結果から, 提案したエラー評価メトリクスは, 設定する閾値次第では有効利用できるといえる．

#### 4.3.1　有用と判断できる実行時エラークラスの種類

A. ある企業内で利用しているウェブシステムについて, 調査により得られた実行時エラークラスの種類を次に示す．

- java.lang.NullPointerException
- java.lang.NumberFormatException
- java.lang.StringIndexOutOfBoundsException
- java.lang.IndexOutOfBoundsException
- java.lang.ClassCastException
- java.lang.Exception
- java.lang.IllegalArgumentException
- java.lang.ArrayIndexOutOfBoundsException
- java.io.FileNotFoundException
- 他, 独自定義の業務エラー３種

上記の内, 前者４つ (NullPointerException, NumberFormatException, StringIndexOutOfBoundsException, IndexOutOfBoundsException) の実行時エラークラスについては, ほぼ全てが “テスト対象メソッドを呼ぶ際の暗黙の了解がある” と気づくために有用であった．また多くのケースにおいて, コードへ任意の修正を加えることで, これらの実行時エラーの発生を防ぐことが可能であった．具

表 1: メトリクスが閾値以上のエラーを確認する場合に見つかる, 呼ぶ前に気をつけるべき暗黙の了解があるメソッドに気づくために, 有用なエラーおよび有用でないエラー数

| 閾値 | 確認するエラー総数 | メトリクス値が閾値以上有用なエラー | メトリクス値が閾値以上有用でないエラー | メトリクス値が閾値以上有用なエラー(割合) | ランダム取得時有用なエラー(期待値) | ランダム取得時有用でないエラー(期待値) | p 値 |
|---|---|---|---|---|---|---|---|
| 全て確認 | 206 | 69 | 137 | 0.334951456 | - | - | - |
| 15 | 18 | 5 | 13 | 0.277777778 | - | - | - |
| 14 | 32 | 17 | 15 | 0.53125 | 10.7184466 | 21.2815534 | 0.018635518 |
| 13 | 37 | 22 | 15 | 0.594594595 | 12.39320388 | 24.60679612 | 0.000819086 |
| 12 | 50 | 27 | 23 | 0.54 | 16.74757282 | 33.25242718 | 0.002126153 |
| 11 | 52 | 29 | 23 | 0.557692308 | 17.41747573 | 34.58252427 | 0.000666083 |
| 10 | 67 | 33 | 34 | 0.492537313 | 22.44174757 | 44.55825243 | 0.006276345 |
| 9 | 80 | 38 | 42 | 0.475 | 26.7961165 | 53.2038835 | 0.007953653 |
| 8 | 95 | 39 | 56 | 0.410526316 | 31.82038835 | 63.17961165 | 0.11859276 |
| | | | | | | | 以降 p 値 0.1 以上 |

体的には testee1 (Listing 1) と類似するケースがあり, 引数 accounts に対して, キー users が格納されているか事前に確認することで null 参照エラーを防ぐことが出来る. 或いは, 実際にエラーを検出した Listing 2 についても, this.skeleton.invoke を呼ぶ前に this.skeleton が null でないことを確認することで null 参照エラーを防ぐことが出来る. また Listing 2 については, 事前条件 "先に prepare を呼ぶ" を満たせばエラーが発生しないことも確認できる. 本節の結果から, 4 種類の実行時エラーについては, 実行時エラーの有用性を測るための分類尺度として利用できる.

### 4.4 RQ.3 Findbugs による警告を再生できるか

調査対象 A, B について, 記号実行器の生成するテストケースが Findbugs による警告を再生した. 調査対象 C については, 警告 4 件についてテストケースの実行によって警告箇所へ到達可能であることは確認できたが, 警告の再生には至らなかった.

#### 4.4.1 A. ある企業内で利用しているウェブシステム

Findbugs は 4,688 件の警告を生成した. 生成したテストケース実行中に発生したエラーは 6,022 件である. うち例外発生原因および例外発生箇所が重複するエラーを除くと 3,357 件となった. 実行時エラーの内, その発生箇所と, Findbugs が警告する行番号が一致したのは 61 件である. うち 9 件の警告について調べたところ, テストケースが再生した警告の数は 6 件であった. 再生した具体例を Listing 2 に示す. また, エラー発生箇所と Findbugs の警告する行番号は一致したが, エラー発生原因と警告理由が異なる例を Listing 5 に示す.

Listing 5: テストケースの実行時エラー発生原因と Findbugs の警告理由が異なるプログラム

```
package org.springframework.beans.propertyeditors;
```

```
// ..
public class CustomDateEditor {
  // ..
  public void setAsText(String text) throws
       IllegalArgumentException {
    if (this.allowEmpty && !StringUtils.hasText(text)) {
      // ..
    } else {
      try {
        setValue(this.dateFormat.parse(text));
      } catch (ParseException ex) {
        throw new IllegalArgumentException("Could not
             parse date: " + ex.getMessage(), ex);
      }
    }
  }
}
```

#### 4.4.2 B. Maven 公式リポジトリ内の OSS

表 2 内の再生数に示すとおり, 調査対象 B. に関しては計 2 件の警告をテストケースとして再生できた. 以降の表で C0, C1 は各々 C0, C1 テストカバレッジを指す. 警告を再生できるか否かと, テストカバレッジとの間に強い相関は見られない.

#### 4.4.3 C. defects4j に収録されている OSS

今回は収録されている 5 プロジェクト Lang(Apache commons-lang), Math(Apache commons-math), Mockito, Time(Joda-Time), Chart(JFreeChart) で修正されたバグ 262 件のうち 58 件を対象とした.

表 3 に記すバグについて, Findbugs により警告された行番号が, 実際に修正された行番号に一致した. Mockito-9, Chart-10 については, テストカバレッジが C0 および C1 ともに 100% を達成したにもかかわらず, 実行時エラーが 1 つも起きなかった. Mockito-9, Chart-10 はテスト対象コード内で実行時エラーを起こすバグではなく, 予め用意されたテストケース内の期待値によって確認できる類のバグであったためである.

表 4 に記すバグについて, 生成したテストケース実行中に実行時エラーが発生した行番号が, 実際に修正された行番号に一致した. しかし発生した実行時エラーは修正対象のバグが原因で生じたものではなかった. つまり, テスト

表 2: Maven 公式リポジトリ内の OSS に対し生成したテストケース実行結果

| Project ID | Findbugs 警告あり クラス数 | テスト 例外あり クラス数 | 再生数 | 行数 | C0 |
|---|---|---|---|---|---|
| assertj-core-1.7.1 | 34 | 9 | 0 | 132 | 73% |
| commons-cli-1.3.1 | 7 | 0 | 0 | 69 | 35% |
| hamcrest-core-1.3 | 1 | 6 | 0 | 9 | 44% |
| httpclient-4.3.6 | 122 | 8 | 0 | 197 | 85% |
| httpcore-4.3.3 | 25 | 58 | 0 | 426 | 58% |
| jackson-core-2.4.4 | 76 | 0 | 0 | 40 | 85% |
| logback-core-1.1.3 | 154 | 0 | 0 | 28 | 35% |
| maven-project-2.2.1 | 34 | 376 | 1 | 810 | 6% |
| plexus-utils-3.0.17 | 141 | 165 | 0 | 2268 | 54% |
| spring-beans-3.1.4 | 113 | 58 | 0 | 482 | 53% |
| spring-core-3.1.4 | 99 | 117 | 0 | 1268 | 46% |
| spring-test-4.0.3 | 59 | 57 | 0 | 541 | 56% |
| spring-web-4.1.4 | 71 | 18 | 1 | 228 | 67% |
| testng-6.8.13-1 | 128 | 479 | 0 | 1099 | 36% |
| tools.nrepl-0.2.6 | 5 | 0 | 0 | 4 | 75% |

表 3: Findbugs による警告が存在する defects4j Bug ID とテストカバレッジ

| Bug ID | C0 | C1 |
|---|---|---|
| Chart-8 | 63% | 33% |
| Chart-10 | 100% | 100% |
| Chart-20 | 54% | 50% |
| Chart-22 | 54% | 27% |
| Chart-24 | 68% | 30% |
| Lang-5 | 62% | 32% |
| Lang-19 | 14% | 27% |
| Lang-44 | 38% | 18% |
| Lang-45 | 89% | 79% |
| Lang-46 | 42% | 37% |
| Lang-4 | 56% | 35% |
| Lang-51 | 55% | 59% |
| Math-56 | 10% | 0% |
| Math-87 | 6% | 2% |
| Math-96 | 71% | 49% |
| Math-102 | 26% | 17% |
| Mockito-2 | 88% | 33% |
| Mockito-9 | 100% | 100% |

ケースの実行で実際に修正されたコードへ到達可能であることのみを確認できた．また, 今回の実験では “修正されたバグがテストケースとして再生可能か否か” と “生成したテストケースのカバレッジ” の間に強い相関は見られない．他, Findbugs に警告されず, かつ, 自動生成したテストケースで到達可能であることを示せないバグが 36 件存在した．

表 4: テストケース実行で修正箇所へ到達可能な defects4j Bug ID とテストカバレッジ

| Project | C0 | C1 |
|---|---|---|
| Lang-7 | 42% | 22% |
| Lang-12 | 74% | 85% |
| Lang-55 | 59% | 46% |
| Math-21 | 65% | 57% |

## 5. 妥当性に対する脅威

### 5.1 テストカバレッジ

テストカバレッジは Findbugs の出す警告の再生（図 1 x）, 或いは実行時エラーの検出（図 1 y）に影響する．テストカバレッジの良し悪しは自動生成したモックファイルに依存する．今後, より高いテストカバレッジを実現でき, かつ汎用的なモックファイルの生成が出来るようになれば, RQ.2 で得られた有用な実行時エラーの件数は増える可能性があり, RQ.3 の結果も改善できる可能性がある．

### 5.2 調査範囲とツールの制限

“ テスト対象クラスがサポートしていない JDK の API を使用している” , “ 開発したシステムに組み込まれている JPF に未修正のバグがある” といった理由で, ときどき記号実行器が強制終了する．もし強制終了を避けることが出来れば調査対象を増やし, 定量的な評価を行う上で十分なデータ量を取得できた可能性がある．

#### 5.2.1 分析対象とする実行時エラーのサンプリング方法

本稿では, 評価メトリクスの各値のエラー数ができるだけ均等になるように実行時エラーの選出を行った．しかし図 4, 5 に示すように, メトリクス値が 13, 11 となるエラーについては選出元の全数が極端に少なかったため偏りが生じている．今後, 調査対象を増やすことでエラーの全数を増やしたい．

また, メトリクスの値によってエラーの有用性の有無には偏りがあると考えられる．よって本稿の分析では有用性のあるエラーが一定数存在することは確認できたが, 全体に占める割合としてどの程度存在するかまでは確認できていない．例えば評価メトリクスの値が 0 となったエラーは全体の約 8 割を占めているため, 今後この範囲についてより詳しく分析を行いたい．

またメトリクスの有用性評価について, 分析対象とするエラーのサンプル件数を増やすことで, より小さな許容誤差で有用性を評価できた可能性がある．例えば本稿の評価方法で許容誤差を 5 % 程度に抑えるためには, メトリクスによる評価値が上位にくる累積エラー件数が少なくとも 350 件程度は必要になる．調査対象を増やすことでエラーの全数を増やしたい．

### 5.3 Findbugs の設定

RQ.3 の OSS 群 B, C に対する調査では Findbugs の設定は全てデフォルト値である. 各々のプロジェクトの特徴を考慮した設定にしていない. また依存するライブラリを全て含めることが出来ないプロジェクトもあった. 4 節 RQ.3 の目的は Findbugs との純粋な性能比較ではなく, Findbugs による警告がテストケースとして再生可能か否かを調べることにあったため, 致命的な問題ではないと判断した. A のウェブシステムに対しては RQ.1 の評価に影響するため, Findbugs ができるだけ多くの警告を出すよう設定した.

### 5.4 バグリポジトリの選定

Bug ID Mockito-9（表 3 ）のように実行時エラーを起こさないケースを取り除くため, 先にバグを分別してから実験した方が効率的である. 例えば, 実行時エラーが発生する類のバグを収録している bugs.jar[16] の利用が考えられる.

## 6. Future work

今後, 評価実験を追加することで, 本稿で示す手法が有効であることを, より定量的に説明したい. 今回は出来る範囲での調査を行ったが, ある程度まで実験対象を増やして結果が変わらないかを確認したい. さらに, Findbugs による警告をテストケースとして提示できるか評価したが, 他の静的解析ツールを用いた実験も行いたい. また, メトリクスの算出方法について, データフローを厳密に追跡してデータ起源を特定する等の方法で改善する可能性がある.

### 参考文献

[1] Infer. "http://fbinfer.com/".
[2] JPF. "https://github.com/javapathfinder/jpf-core".
[3] OpenJML. "https://github.com/OpenJML/OpenJML".
[4] SonarQube. "https://github.com/SonarSource/sonarqube".
[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.
[6] W. H. K. Bester, C. P. Inggs, and W. C. Visser. Test-case Generation and Bug-finding Through Symbolic Execution. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 1–9, New York, NY, USA, 2012. ACM.
[7] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 108–128, Berlin, Heidelberg, 2005. Springer-Verlag.
[8] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.
[9] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.
[10] M. Islam and C. Csallner. Generating Test Cases for Programs That Are Coded Against Interfaces and Annotations. *ACM Trans. Softw. Eng. Methodol.*, 23(3):21:1–21:38, June 2014.
[11] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.
[12] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of Bug Fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006. ACM.
[13] Z. Li, P. Avgeriou, and P. Liang. A Systematic Mapping Study on Technical Debt and Its Management. *J. Syst. Softw.*, 101(C):193–220, Mar. 2015.
[14] S. Monpratarnchai, S. Fujiwara, A. Katayama, and T. Uehara. An Automated Testing Tool for Java Application Using Symbolic Execution Based Test Case Generation. In *Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC) - Volume 02*, APSEC '13, pages 93–98, Washington, DC, USA, 2013. IEEE Computer Society.
[15] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.
[16] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 648–659, Piscataway, NJ, USA, 2017. IEEE Press.
[17] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 717–720, Piscataway, NJ, USA, 2015. IEEE Press.
[18] A. Vetro'. Using Automatic Static Analysis to Identify Technical Debt. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1613–1615, Piscataway, NJ, USA, 2012. IEEE Press.