一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

自動テスト生成を用いたリファクタリング検証手法の改善

松尾　唯音†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{yu-matsuo,shinsuke}@ist.osaka-u.ac.jp

あらまし　リファクタリングとは，振る舞いを保持しつつソースコード内部の構造を改善するプロセスのことである．リファクタリングの実施の際に，リファクタリング失敗，すなわちソースコードの変更前後での振る舞いの差異が発生してしまう場合がある．リファクタリング失敗に対しては，テストを用いた検証が可能である．テストを用いた検証として，自動テスト生成を用いた等価性検証が存在する．しかし既存研究の報告によると，自動テスト生成によるリファクタリング検証手法の精度は約 50% と高くない．本研究ではリファクタリング失敗の自動検出を目的として，自動テスト生成ツールを用いたリファクタリング前後の等価性検証の改善に取り組む．テストの多様性を減らさずに等価性判定をより徹底的に行うことで，リファクタリング失敗の検出率を向上させる．本研究会原稿では，既存研究の手法を改善した検証手法を提案し，その有効性を評価する．

キーワード　リファクタリング，ソフトウェアテスト，自動テスト生成

1. はじめに

リファクタリングとは，外部の振る舞いを保持したまま，ソースコードの内部構造を改善するプロセスである [1] [2]．ソフトウェア開発において，ソースコードは要求追加や仕様変更に伴って継続的に修正される．そのため，保守性や可読性の低下を抑える観点から，リファクタリングは開発の中で欠かせないプロセスである．

リファクタリングの際には，ソースコード改変の前後で外的な振る舞いに差が生じてしまう場合がある．つまり，リファクタリングによって 2 つのソースコードの等価性を破壊してしまう．IDE が提供するリファクタリング支援は，変数名変更やメソッドのインライン化などのリファクタリングを容易に適用できる．しかし，リファクタリング支援ツールによるリファクタリングにはバグが含まれ得ることが報告されており，ツール支援が存在しても，等価性破壊のリスクをゼロにすることはできない [3] [4] [5]．さらに，IDE で自動変換できるようなリファクタリングは，実世界におけるリファクタリングの種類の一部にすぎない．開発現場では，機能追加やバグ修正の合間に小規模なリファクタリングを繰り返すだけでなく，API 変更などの構造改善を主目的とする大規模かつ複雑なリファクタリングも多く存在する [6]．このような大規模かつ複雑なリファクタリングでは，複数箇所にまたがる変更の整合や境界条件の扱いが難しく，結果として等価性破壊が発生し得る．したがって，リファクタリングを安全に適用するためには，リファクタリング後のソースコードがリファクタリング前のソースコードと等価であるかを検証する仕組みが重要となる．

リファクタリング前後のソースコードの等価性を検証する方法として，テストスイートが広く用いられている．これは，リファクタリング前後それぞれのソースコードにテストを適用することで，リファクタリング前後の等価性をテストを通じて判定するというアプローチである．2 つのソースコードの振る舞いが完全に等価であることを示すことは決定不能であるため，テストによる近似的な検証を行う．ソースコードの等価性判定には，静的解析や定理証明など複数の選択肢がある．さらに近年は，テスト駆動開発（Test-Driven Development：TDD）や CI の普及によりテストが存在している場合が多く，実務においてテストを用いた等価性判定の方が検証が容易である．

しかし Silva らによる既存研究によると，自動テスト生成ツールを用いたリファクタリング失敗の検証は，検出率が約 50% と精度が高くない [7]．Silva らは，自動テスト生成ツールを用いて，リファクタリング前のソースコードからテストを生成し，リファクタリング後にテストを適用することで検証している．自動テスト生成によるリファクタリング検証で見逃しが生じる要因として，EvoSuite が境界値や条件分岐のバグの検出能力が低いことが挙げられている．

本研究では，リファクタリング失敗の自動検出を目的として，自動テスト生成ツールを用いたリファクタリング前後のソースコードの等価性検証の改善に取り組む．既存の Silva らの手法を改善し，より高精度の等価性判定を行うことを目指す．提案手法は非最小化と相互適用によって構成される．非最小化によりテストの多様性を保ち，境界値近傍のバグの検出率の向上を狙う．また，相互適用によりリファクタリング後に追加された分岐や条件へのテストを増やし，等価性判定をより徹底的に行う．

本稿では，既存研究の Silva らの手法を改善した検証手法を提案し，それを擬似題材に対して適用することで，提案手法のリファクタリング検証における有効性を議論する．

2. 準　　備

2.1 自動テスト生成ツール：EvoSuite

2.1.1 EvoSuite の概要

EvoSuite は，遺伝子アルゴリズム（Genetic Algorithm: GA）による探索で，カバレッジが最大となるような JUnit テストスイートを自動生成するツールである．Java のソースコードを対象とし，対象クラス（Class Under Test: CUT）のそれぞれの public メソッドに対して，分岐網羅などのカバレッジ指標を高めるようにテストスイート全体を最適化する [8] [9]．EvoSuite の特徴は，テストケース単体ではなくテストスイート全体に対して，テストスイート内のテストを組み合わせてカバレッジが最大となるように探索を進める点にある [9].

また EvoSuite は，仕様に基づいてテストを生成するトップダウンな方法ではなく，ソースコードを基にテストを生成するボトムアップな方法である．つまり，テスト生成実行時点の外部の振る舞いを正しい振る舞いとするようなアサーションを付与する．そのため，生成されたテストを別のバージョンのソースコードに適用した際に，外部の振る舞いに変更があればテスト失敗となる．この性質は，後述するリファクタリング前後の等価性検証において重要となる．

2.1.2 EvoSuite の生成プロセス

EvoSuite によるテスト生成は大きく，(i) 引数の生成（呼び出し列と値の探索），(ii) アサーションの生成からなる．引数の生成では，CUT のメソッドを呼び出すために必要なオブジェクト生成や，メソッド実行に必要な引数を構成する．引数を生成するための探索は GA に基づいているため，同じ CUT でも生成結果が変化し得る．

アサーションの生成では，テストを生成した時点での外部の振る舞いをもとにアサーションを付与する．例えば，メソッド呼び出しの返り値がある場合や，生成したオブジェクトのフィールド値を観測できる場合は，その値を期待値とするようなアサーションが生成される．

このように EvoSuite による自動テスト生成は，仕様が十分に記述されていないソースコードに対しても外部の振る舞いの違いを検出できる利点を持つ．

2.1.3 最　小　化

EvoSuite は生成後の後処理として，テストスイートの最小化を行う．最小化は，2.1.2 節の生成プロセスで生成されたテストスイートをより短く実行コストの低いテストスイートに縮約する処理である [9]．具体的には，生成したテストスイート全体のカバレッジを維持しつつ，冗長なテストのステートメントやテスト，アサーションを削除する．そのため，カバレッジ最大化を目的とする通常のテスト生成用途では，最小化は計算コストの軽減に有益である．

2.2 既 存 研 究

2.2.1 既存研究の目的と位置付け

Silva らは，自動テスト生成ツールがリファクタリング検証において，どの程度有効的であるかを分析している [7]．リファクタリングは振る舞いが保存されることを意図して実施される．しかし，リファクタリング実施の際に，リファクタリング前後で外部の振る舞いが変化する可能性がある．したがって，外部の振る舞いが等価であることを判定する必要があるが，プログラム等価性判定は決定不能であり，テストによる近似的な検証が用いられる．そこで Silva らは，自動テスト生成により得たテストをリファクタリング後のソースコードへ適用し，リファクタリング失敗をどの程度検出できるのかを実証的に評価している．

2.2.2 既存研究のリファクタリング検証手法

既存研究のリファクタリング検証方法の前に，一般的なテストを用いたリファクタリング検証方法を紹介する．リファクタリング前のソースコードを R_{pre}，リファクタリング後のソースコードを R_{post} とする．リファクタリングが正しく行われた場合，両者は外部の振る舞いが等価であることが期待される．そこで，あるテスト T を両者に適用し，テスト結果に差があれば，R_{pre} と R_{post} は等価でないと判定する．つまり，リファクタリングに失敗したと判定する．この手法が，一般的なテストを用いたリファクタリング検証である．

一方，Silva らは次のような手法でリファクタリング検証を行っている．以降，この手法を Silva らの手法と呼ぶ．Silva らの手法は，テスト T を R_{pre} から自動テスト生成で生成する．Silva らの手法によるリファクタリング検証の流れは次の通りである．

（1） R_{pre} を対象に自動テスト生成を行い，テストスイート T を得る．

（2） T を R_{post} へ適用し，テスト結果を観測する．

（3） テストが 1 つでも失敗した場合は，リファクタリング失敗を検出したとみなす．

R_{pre} からテストを生成するため，2.1 節で述べたようにテスト T は R_{pre} の振る舞いを正とするように生成される．そのため，テスト T を R_{pre} に適用した場合，必ずテストは成功する．これより Silva らの手法は，テストを R_{post} のみに適用するだけでリファクタリング検証ができる点が特徴である．

2.2.3 見逃しが生じ得る要因

Silva らは，自動テスト生成を用いたリファクタリング検証では，リファクタリング失敗の見逃しが約 50% 発生することを報告している [7]．自動テスト生成によるリファクタリング検証で見逃しが生じ得る要因として，EvoSuite が境界値や条件分岐のバグの検出能力が低いことが挙げられている．これは，EvoSuite が分岐網羅を目的としてテスト生成を行うためである。分岐網羅とは，ソースコード中の各分岐について，真偽両方の経路を少なくとも 1 回通過するようにテストを設計することである．各分岐を通るだけで良いため，境界値近傍を検証することなく分岐網羅を満たすことができる．そのため，境界値や条件分岐近傍でバグが含まれている場合の検出率が下がってしまう．

3. 自動テスト生成を用いたリファクタリング検証

3.1 提案手法の概要

本研究の目的は，自動テスト生成を用いたリファクタリング検証における，リファクタリング失敗の検出率の向上である．Silva らは，2.2.3 節で述べたように，約 50% のリファクタリング失敗を見逃すことを報告している．そこで本稿では，リファクタリング失敗の見逃しを低減し，2 つのソースコードの等価性検証をより徹底するための改善手法を提案する．

提案手法の基本方針は次の 2 点である．第一に，EvoSuite により自動生成されるテストの網羅性を保ち，境界値や条件分岐に現れるバグの検出率を向上させる．第二に，リファクタリング後に分岐や条件が新しく追加される場合に生じるリファクタリング失敗の見逃しを低減する．

上記方針を実現するために，提案手法は非最小化と相互適用から構成される．非最小化は，テスト生成後の最小化処理を適用せず，生成されたテストを削除しないことでテストの多様性を保つ．特に，境界値近傍を検証するテストが削除されることを避け，リファクタリング失敗の検出率向上を狙う．相互適用は，リファクタリング前後それぞれから生成したテストを交差させて適用する．すなわち，リファクタリング前から生成したテストをリファクタリング後へ適用するだけでなく，リファクタリング後から生成したテストもリファクタリング前へ適用することで等価性判定をより徹底的に検証する．

提案手法は，非最小化によりテストの多様性保ち，さらに相互適用により等価性判定を強固にする．一方 Silva らの手法は，リファクタリング前から生成したテストをリファクタリング後へ適用する片方向の検証のみであり，最小化処理を適用している．本稿では，リファクタリング前のプログラムを R_{pre}，リファクタリング後のプログラムを R_{post} とする．また，R_{pre} から生成したテストスイートを T_{pre}，R_{post} から生成したテストスイートを T_{post} と表す．

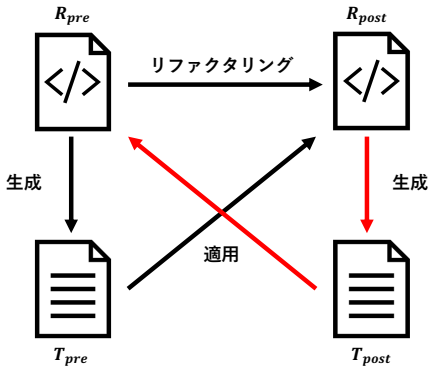

図 1: 相互適用の概要

3.2 非最小化

2.1.3 節で述べたように，EvoSuite はテスト生成後に最小化を行い，カバレッジを維持しつつテストを削除する [9]．しかしリファクタリング検証では，カバレッジという観点では捉えにくい境界値近傍などのバグを検出するために，引数やテスト数の多様性が重要となる．しかし最小化は，冗長なアサーションや，カバレッジ上は不要な追加の呼び出しを削除し，バグの検出率を低下させる可能性がある．

そこで本稿では，EvoSuite 実行時にテスト生成後の削除を行わないようにする．つまり，最小化処理を適用しない．非最小化では，EvoSuite のテスト生成過程で通常削除される，カバレッジ向上に寄与しない冗長なテストのステートメントやテストを残すことで，リファクタリング失敗の検出率を向上させる．

3.3 相互適用

Silva らの手法である片方向での検証のみでは，R_{pre} に存在する分岐や条件の探索で生成されるテストのみを扱う．そのため，R_{post} で新規に追加した分岐や条件に由来するバグを拾えない可能性がある．そこで相互適用では，リファクタリング前後両方からテストを生成し，交差して適用させることで，等価判定をより徹底的に行う．

相互適用は，図 1 に示す通り，次の 2 通りの実行を行うことで実現させる．

（1） R_{pre} から生成した T_{pre} を R_{post} に適用する．

（2） R_{post} から生成した T_{post} を R_{pre} に適用する．

(2) は図 1 における赤線の流れを表している．この赤線の流れが Silva らの手法にはない逆方向の検証である．この逆方向の検証を追加することにより，2 つのソースコードの等価性検証をより徹底的に行う．

相互適用を行うにあたって，前後で公開 API が変化するケースは，除外する．相互適用を行うには，リファクタリング前後両者で生成したテストが適用できなければならない．しかし，API が変更されると片方で生成されたテストが他方でコンパイルできなくなってしまう．この API 変更を伴うリファクタリングに対する対応については，後の 5.2 節で説明する．

提案手法では，片方のみテストが失敗した場合と両方でテストが失敗した場合をリファクタリング失敗を検出したと判定する．一方で，両方でテストが成功した場合はリファクタリング失敗を見逃したと判定する．

4. ケーススタディ

本節では，2 つの事例を用いて提案手法のケーススタディを示す．以降では，既存の Silva らの手法 [7] を比較対象として，提案手法の効果を説明する．自動テスト生成には，既存の Silva らの手法と同様に EvoSuite を用いる．

4.1 シナリオ 1：料金の割引率の計算

本節では，料金の割引率の計算プログラムを題材に，提案手法がリファクタリング失敗の検出に寄与することを示す．対象プログラムの仕様は次の通りである．(1) 18 歳未満はチケット価格が半額とする．(2) 18 歳未満の中で価格が 2000 円以上の場合，チケット価格は 4 割引とする．

図 2 は，料金の割引率計算プログラムのリファクタリング前のソースコード R_{pre} である．`discount` メソッドはチケッ

```
int discount(User user, Ticket ticket) {
    double rate = 1;
    if (user.age < 18) {
        if (ticket.price >= 2000) {
            rate = 0.6;
        } else {
            rate = 0.5;
        }
    }
    return (int) (ticket.price * rate);
}
```

図 2: シナリオ 1 の R_{pre}

トの割引後の価格を返すメソッドである．user がチケット購入者の年齢，price がチケットの元の価格を表す．

また図 3 は，料金の割引率計算プログラムのリファクタリング後のソースコード R_{post} である．チケット料金の割引率を取得する処理を getDiscountRate メソッドとして抽出し，discount メソッドから呼び出す形に変更している．このリファクタリングの際に，境界値のバグが埋め込まれてしまっている．具体的には，年齢条件が age < 18 から age <= 18 に誤って変更されてしまっている．

この R_{pre} と R_{post} に対し，リファクタリング失敗を検出できるか検証する．既存の Silva らの手法では，リファクタリング前のソースコードからテストを生成し，リファクタリング後に適用する片方向の検証を行う．その際に使用する EvoSuite は，設定を変更せずにデフォルトのままである．結果として，Silva らの手法ではリファクタリング失敗を検出できなかった．

一方，提案手法を用いて同様の検証を行った場合，リファクタリング失敗を検出することができた．具体的には，年齢の境界値 age = 18 を含むテストを生成できた．図 4 に提案手法によるテスト生成結果を示す．図 4 のテストは age = 18 のとき割引が適用されず，料金 18 が返るという振る舞いを期待している．しかし，リファクタリング前では割引が適用されないはずの age = 18 に対して，リファクタリング後では割引が適用され料金が 7 となる．したがって，age = 18 に対する返り値が不一致となり，提案手法を用いたテストではリファクタリング失敗を検出できる．

EvoSuite は生成後にテストスイート最小化を行い，カバレッジを維持したまま冗長なテストを削除する [9]．分岐網羅を満たすためには age = 0 や age = 17 などで代替可能な場合であり，境界値 age = 18 は，EvoSuite の最小化によりカバレッジ的に冗長と判断され削除されうる．本シナリオでは，リファクタリング時のバグが境界値にのみ現れるため，このような境界値テストが削除されると境界値付近でのバグが見つけられず，リファクタリング失敗の見逃しにつながる．

これに対して提案手法の非最小化では，探索過程で得られたテストケースが削除されないため，境界値入力を含むテストが残る可能性が高まる．その結果，提案手法を適用していない場合と比較して，境界値付近でのバグが混入するリファクタリング失敗の検出率が向上する．以上より，非最小化は境界値近傍にバグが含まれるという状況で，リファクタリング失敗の見逃し低減への寄与を確認できた．

```
int discount(User user, Ticket ticket) {
    double rate = getDiscountRate(user, ticket);
    return (int) (ticket.price * rate);
}

double getDiscountRate(User user, Ticket ticket) {
    if (user.age <= 18) {.          // BUG
        if (ticket.price >= 2000) {
            return 0.6;
        }
        return 0.5;
    }
    return 1;
}
```

図 3: シナリオ 1 の R_{post}

```
void test01()  throws Throwable  {
    User user0 = new User(18);
    Ticket ticket0 = new Ticket(18);
    int int0 = example0.discount(user0, ticket0);
    assertEquals(18, int0);
}
```

図 4: 非最小化により生成されたテスト

```
int Delta(int a, int b) {
    return Math.abs(a - b);
}
```

図 5: シナリオ 2 の R_{pre}

```
int Delta(int a, int b) {
    if ((a - b) == Integer.MIN_VALUE) {   // BUG
        return 0;
    }
    return Math.abs(a - b);
}
```

図 6: シナリオ 2 の R_{post}

4.2 シナリオ 2：差分の絶対値の計算

本節では，差分の絶対値の計算プログラムを題材に，提案手法がリファクタリング失敗の検出に寄与することを示す．対象プログラムは，2 つの整数引数の差分の絶対値を返すという仕様である．

図 5 は，差分の絶対値計算プログラムのリファクタリング前のソースコード R_{pre} である．Delta メソッドは，2 つの整数引数 a，b の差分の絶対値を返すメソッドである．

また図 6 は，差分の絶対値計算プログラムのリファクタリング後のソースコード R_{post} である．条件分岐を早期リター

ンで整理するガード節化を行っており，このガード節化は，典型的なリファクタリングとして知られている [1]．具体的には，`Integer.MIN_VALUE` という最小値が入ってしまった場合に，2^{31} を表すことができない問題に対処するための分岐を追加している．このような `Math.abs(Integer.MIN_VALUE)` が負の値のままとなる問題は実プロジェクトでも実際に起こっており，最小値を考慮した修正が行われている [10]．このリファクタリングの際に，最小値の処理が不適切となり，期待している振る舞いと異なってしまっている．この図 5 と図 6 のソースコードに対し，リファクタリング失敗の検出を検証する．

しかし，本シナリオは差分が最小値の際の処理を追加する変更を含むため，厳密には振る舞い保存を満たしていない．つまり，厳密にはリファクタリングではない．一方実務では，バグ修正・機能追加の途中にリファクタリングが混在することが多いことが報告されている [11]．そのため，本節ではこのような修正混在により振る舞い不一致が生じた状況を，リファクタリング失敗の一例として扱う．

まず，R_{pre} と R_{post} に対し，既存の Silva らの手法で検証したところ，リファクタリング失敗を見逃した．Silva らの手法は，R_{pre} に対し EvoSuite でテストを生成し，そのテストを R_{post} へ適用するという片方向の検証である（pre→post）．リファクタリング失敗を見逃したのは，R_{pre} には `Integer.MIN_VALUE` に対応する分岐が存在しないためである．そのため，探索の際に `(a-b)==Integer.MIN_VALUE` となるような引数を持つテストの生成が誘導されにくい．このように，片方向適用ではリファクタリング前の実装構造に依存して探索が偏るため，リファクタリング失敗の見逃しが生じうる．

次に，提案手法で検証したところ，リファクタリング失敗を検出した．提案手法は，Silva らの手法に加えて，R_{post} に対し EvoSuite でテストを生成し，そのテストを R_{pre} へ適用するという逆方向適用も実施している（post→pre）．提案手法によりリファクタリング失敗を検出できたのは，R_{post} には `(a-b)==Integer.MIN_VALUE` の分岐が明示的に存在するためである．そのため，EvoSuite は，`(a-b)==Integer.MIN_VALUE` の分岐網羅を目標としたテスト生成を行いやすい．結果として，リファクタリング失敗を検出できるような引数（例：`a=0`，`b=Integer.MIN_VALUE`）を含むテストが生成され得る．生成されたテストを R_{pre} へ適用すると，R_{post} 側で期待されている返り値 (0) と R_{pre} の返り値 (-2^{31}) が不一致となり，リファクタリング失敗を検出することができる．

以上より，本シナリオでは pre→post はリファクタリング失敗を見逃すが，post→pre ではリファクタリング失敗を検出できるという結果が得られた．これは，バグがリファクタリング後に追加された分岐に存在し，片方向適用では探索がその分岐へ到達できなかったためと解釈できる．したがって，相互適用は片方向適用の探索バイアスを緩和し，リファクタリング前後の等価性検証をより徹底する上で有効な手段となることが確認できた．

5. 今後の課題

本章では，提案手法を研究としてより強固にするために必要な課題を述べる．特に，実データにを用いた定量的な評価の実験方法と API 破壊を含むリファクタリングへの対応の 2 点を中心に議論する．

5.1 実データを用いた定量的な評価実験

本稿ではケーススタディにより提案手法の直観的有効性を示した．一方で，研究としては，より大規模な実データに基づく定量的評価が不可欠である．特に，自動テスト生成を用いたリファクタリング検証の有効性は，対象プロジェクト，リファクタリング種別，およびバグの作り方に強く依存するため，評価データの設計自体が主要な検討課題となる．

5.1.1 評価データの基本方針

リファクタリング検証の評価には，リファクタリングに失敗した事例を用意する必要がある．しかし，実プロジェクトではリファクタリングとそれ以外の編集を含んだコミットが多いため，リファクタリング単独の失敗事例として切り出しにくい．そのため，実プロジェクトからリファクタリング失敗事例を直接収集することは困難である [12] [13]．そこで，次の 2 つのステップで評価データを構成する．

i）リファクタリング成功データの収集とフィルタリング

まず，リファクタリング成功データとして，RefactoringMiner 2.0 を用いる [14]．ここで公開されているデータは，多数のリファクタリングコミットを提供するが，コミット単位ではリファクタリング以外の変更が混入し得る．そこで，リファクタリング成功データの信頼度を高めるため，以下のフィルタを実行する．

- 対象は Java プロジェクトに限定し，リファクタリング前後のファイルがビルド可能であることを確認する．
- 既存テストが存在する場合は，前後のファイルで既存テストが通過することを確認し，等価とみなせるデータを優先する．

ii）リファクタリング失敗データの作成

リファクタリング失敗例を実データから直接収集する代わりに，本研究では，収集したリファクタリング成功例に対して，リファクタリング失敗を模したバグを意図的に注入することで，リファクタリング失敗データを構成する方針を採る．ミューテーションテストは人工欠陥によりテストの検出能力を測る枠組みとして成熟しており，欠陥注入の設計・分類・妥当性に関して多く研究されている [15]．そこで，収集したリファクタリング成功例に対し，ミューテーションを加えることでリファクタリング失敗例を生成する．

5.1.2 対象プロジェクトとリファクタリング種別の選定

題材の選定では，まず，RefactoringMiner 2.0 に含まれるプロジェクトから，(1) ビルドが自動化されている，(2) 外部環境依存が小さい，といった条件で候補を抽出する [14]．対象とするリファクタリング種別は，5.2 節で説明する対策をした上で 40 種すべてを扱う．これにより，どの種別に対して非最小化が効くか，どの種別に対して相互適用が効くかを分解して評価

できる．

5.1.3 評価指標と実験設計

比較対象として，既存研究の手法を基本の検証方法とし，次の 4 条件を比較する．

（1） 既存手法（Silva らの手法）
（2） 提案手法（非最小化）
（3） 提案手法（相互適用）
（4） 提案手法（非最小化と相互適用）

自動テスト生成は確率的要素を持つため [9]，シードを変えた複数回実行を行い，評価する．指標としては，リファクタリング失敗の検出率を用いる．以上の設計により，提案手法による改善効果を実データで定量的に評価できると考える．

5.2 API 破壊を含むリファクタリングへの対策

提案手法の相互適用は，リファクタリング前後のコードが同一のテストをコンパイル・実行できることを前提としている．しかし実際のソフトウェア開発では，公開 API の変更により，リファクタリング後のソースコードがコンパイルできなくなる破壊的変更が発生しうる．API の変更の多くがリファクタリングで行われることや，コンパイルに影響を及ぼす変更が生じうることは，既存研究でも指摘されている [16] [17]．

API 破壊が存在すると，リファクタリング前で生成したテストをリファクタリング後に適用できないため，等価性検証としての比較が不可能になる．この問題に対する現実的な方向性として，本研究ではリファクタリング前後の間に互換層を挟む方針が考えられる．すなわち，旧 API と新 API の間を変換するラッパーを用意し，テスト生成時にラッパーを仲介することで，両バージョンでテストの実行ができるようにする．

互換層の構築は手作業では高コストであるが，API マイグレーションを支援する研究が多数存在する．例えば，Diff-CatchUp や SemDiff，ReBA は API 変更前後のソースコードの互換性を保つことを支援するツールである [18] [19] [20]．これらのツールが存在することを踏まえると，リファクタリング前後の間に互換層を挟む方針は実現可能性が高いと考えられる．

以上より，API 破壊を含む実世界のリファクタリングを対象にするには，テスト生成以前に，同一のテストをリファクタリング前後のソースコードへ適用可能にする仕組みが必要である．互換層の導入はその有力候補であり，既存の API の変更に関する研究を参考にしつつ，提案手法の適用範囲を拡張することが今後の課題である．

6. おわりに

本稿では，自動テスト生成を用いたリファクタリング検証の精度向上を目的として，Silva らの手法をもとに，非最小化と相互適用を導入した改善手法を提案した．その有効性を評価するためにケーススタディを行い，Silva らの手法と比較してどのようなバグの検出が可能になったのかを調査した．結果としてケーススタディにおいて，Silva らの手法よりもリファクタリング失敗の検出能力の改善に成功した．

今後の課題として次の 2 つを挙げる．1 つ目は実データを用いた定量的な評価実験を行い，提案手法の評価を検証することである．RefactoringMiner 2.0 等の実データセットからリファクタリング成功例を収集し，ミューテーションを加えたリファクタリング失敗例を作成する必要がある．2 つ目は API 破壊を含むリファクタリングに対する対策である．リファクタリング前後の間に互換層を導入することにより，API 破壊を含むリファクタリングに対しても提案手法を適用することができる．そのため，提案手法の適用範囲を拡張することが可能となる．

謝辞 本研究の一部は，JSPS 科研費（JP25K15056, JP25K03102, JP24H00692）による助成を受けた．

文　献

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] T. Mens and T. Tourwé, “A survey of software refactoring,” IEEE Trans. Softw. Eng., vol.30, no.2, pp.126–139, 2004.
[3] G. Soares, R. Gheyi, and T. Massoni, “Saferefactor: A tool for checking refactoring safety,” Brazilian Symp. Softw. Eng., pp.49–54, 2009.
[4] G. Soares, R. Gheyi, and T. Massoni, “Automated behavioral testing of refactoring engines,” IEEE Trans. Softw. Eng., vol.39, no.2, pp.147–162, 2013.
[5] B. Daniel, D. Dig, K. Garcia, and D. Marinov, “Automated testing of refactoring engines,” Proc. Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Int’l Symp. Foundations of Softw. Eng., pp.185–194, 2007.
[6] E. Murphy-Hill and A.P. Black, “Refactoring tools: Fitness for purpose,” IEEE Softw., vol.25, no.5, pp.38–44, 2008.
[7] I.P.S.C. Silva, E.L.G. Alves, and W.d.L. Andrade, “Analyzing automatic test generation tools for refactoring validation,” Int. Workshop on Automation of Software Testing (AST), pp.38–44, 2017.
[8] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” Proc. Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Int’l Symp. Foundations of Softw. Eng., pp.416–419, 2011.
[9] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” IEEE Trans. Softw. Eng., vol.39, no.2, pp.276–291, 2013.
[10] Apache Software Foundation, “STORM-1481: avoid Math.abs(Integer) get a negative value,” JIRA issue. Accessed 2026-02-03. https://issues.apache.org/jira/browse/STORM-1481
[11] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” IEEE Trans. Softw. Eng., vol.38, no.1, pp.5–18, 2012.
[12] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” Proc. Int’l Conf. on Softw. Eng. (ICSE), pp.287–297, 2009.
[13] S. Herbold, et al., “A fine-grained data set and analysis of tangling in bug fixing commits,” Empirical Softw. Eng., pp.125–125, 2022.
[14] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” IEEE Trans. Softw. Eng., vol.48, no.3, pp.930–950, 2020.
[15] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” IEEE Trans. Softw. Eng., vol.37, no.5, pp.649–678, 2010.
[16] D. Dig and R. Johnson, “The role of refactorings in API evolution,” IEEE Int’l Conf. Softw. Maintenance, pp.389–398, 2005.
[17] R.G. Kula, A. Panichella, and A. Zaidman, “An empirical study on the impact of refactoring activities on evolving client-used APIs,” Information and Software Technology, vol.93, pp.186–199, 2018.
[18] Z. Xing and E. Stroulia, “API-evolution support with diff-catchup,” IEEE Trans. Softw. Eng., vol.33, no.12, pp.818–836, 2007.
[19] B. Dagenais and M.P. Robillard, “SemDiff: Analysis and recommendation support for API evolution,” Proc. Int’l Conf. Softw. Eng. (ICSE), pp.599–602, 2009.
[20] D. Dig, S. Negara, and V. Mohindra, “ReBA: Refactoring-aware binary adaptation of evolving libraries,” Proc. Int’l Conf. Softw. Eng. (ICSE), pp.441–450, 2008.