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

# テストカバレッジに基づくテストケース間の包含関係の提案

岡本　琉生†　　柗本　真佑†　　楠本　真二†

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

**あらまし**　プロダクトの品質保証や開発中でのバグの早期発見のため，ソフトウェアテストは重要である．テスト手法の 1 つである自動テストは，開発者がプロダクトの予期する動作をプログラム（テストケース）として複数記述し，それらの実行によって検証する方法である．テストケースには単一機能を確認する単体テストと複数の機能の組み合わせを確認する結合テストがある．結合テストは複数の単体テストをある種包含しているといえる．この包含関係は自然な性質であるにも関わらず，定義および検出法，その活用はわれわれが調べた限り存在しない．本研究では，包含関係の定量的な定義を与え，それを利用した単体・結合の自動計測方法を与える．また，包含関係のデバッグ支援への活用例を 2 つ提案する．「失敗テストケースが複数存在する場合の解決順序の提示」と「結合寄りテストケースでしか通過されない箇所の報告」である．

**キーワード**　テストケース，単体テスト，結合テスト，包含関係，包含グラフ，包含レベル

## 1. はじめに

ソフトウェア開発においてテストは重要である．テストの実施によりシステムの信頼性を確保できるだけでなく，バグの早期発見やそれに伴う開発コストの削減につながるとされている．本研究では様々なテスト活動のうち，JUnit[1]等のテストフレームワークを用いたプログラムベースのテスト方法である自動テストに着目する．

テストにおける広く知られた概念として，単体テストと結合テストが存在する．一般的に，単体テストは単一の機能を検証するテスト，結合テストは複数の機能の組み合わせを検証するテストと定義される [1] [2]．自動テストにおいては，複数の単体テストケースによって個々の独立した機能の正しさを徹底的に検証する．この際，検証の十分性の計測にはカバレッジが用いられることが多い．プロダクト全体のうちのどの程度がテストによって実行されたかを表す指標である．各機能の組み合わせについては，複数の機能を呼び出すプロダクトを対象とした結合テストケースによって検証される[2]．

単体・結合テストの考えは定性的，あるいは概念的な定義にとどまっており，その具体的な識別の基準は我々の知る限り存在しない．この定義の曖昧さは，「単一の機能」の定義の曖昧さに起因する．つまり，ソフトウェアが備える機能のうち，何をもって単一の機能と見なすかは様々な解釈が存在する．例えば，1 つのユースケースは利用者への直接的な価値を与える単一の機能であると見なせるが，ユースケースは複数の内部機能，例えばメソッドによって実現されることが多い．また，メソッドはソフトウェア内部における最小の機能単位であるとも解釈できるが，ユースケースと同様に他の複数のメソッド，すなわち機能によって実現されることも多い．

もし単体・結合を自動で識別し，さらに単体・結合のような二値ではなく，結合度のような連続値として計測できれば，それらをテスト活動の支援に役立てられる可能性がある．1 つの活用例は複数のテストケースが失敗した際の解決順序の提示である．複数のテストケースが失敗した場合，より単体寄りなテストケースを優先して解決するべきだと考えられる．結合寄りテストケースは複数の機能にまたがって広く検証するため，欠陥箇所の特定が難しいためである．

本研究ではテスト活動の支援を目的として，テストケースの単体・結合の自動計測方法について検討する．キーアイデアは，テストケース間の単体・結合関係の抽象化である包含関係の利用である．具体的には，テストケースの実行したプロダクトコードの行の集合が別のテストケースのそれを内包するとき，その間に包含関係があると見なす．さらに，包含関係の測定結果に基づいた以下の 2 つの活用方法を提案する．

（1）失敗テストケースが複数存在する際の解決順序の提示
（2）結合寄りテストケースでしか通過されない箇所の報告

以降，1 つ目を ORD（ORDer），2 つ目を COV（COVer）と呼ぶ．

提案手法の有効性を確認するために 3 つの評価実験を行った．実験内容は実プロジェクトにおける包含関係の存在確認，および ORD と COV のデバッグ支援としての有効性評価である．結果，実際のプロジェクトにおいて包含関係が検出され，そのいくつかは従来の単体・結合の関係であると確認できた．ORD では，素朴な方法であるカバレッジ行数昇順の解決順序に対して約 10% だけデバッグ時の労力を削減できた．COV では，結合寄りテストケースでしか通過されない箇所が実際のプ

(注1)：https://junit.org/junit5/
(注2)：自動テストは，そのフレームワークである XUnit の名前にもあるように，単体テスト専用の方法と見なされることがある．しかし実際には，Facade クラスや Factory クラス，エントリポイント等を対象とした結合テストに該当するテストケースも多数存在する．

ロジェクトに存在することを確認し，簡単なテストケースの追加によってそのような箇所を削減できることを確認できた．

## 2. 単体・結合の識別に対する課題

単体・結合の定義は様々であり，その多くが定性的である．従来の定義では，プロダクトコード上に「単体」という単位を定義し，1 つの「単体」のみを検証するものが単体テスト，そうでないものが結合テストであるとされる [1] [2]．「単体」の定義は，1 つのメソッドやクラス，パッケージなど様々な解釈がありうる．Titus らはテストの持つ「規模」と「範囲」という 2 つの要素に着目した分類を試みている [3]．規模はテスト実行時に必要なメモリや時間などのリソースを，範囲はテスト実行によって検証されることが意図されているプロダクトコードの部分を表す．このうち範囲に着目して単体・結合と呼び分けるが，具体的な数値指標は明示されず，小・中・大 の定性的な尺度で分類されるにとどまっている．

Trautsch と Grabowski は Python で書かれた OSS において，従来の定義に従うような単体テストが実際に使われているかを調査している [4]．開発者が意図した単体テストの多くが従来の単体テストの定義に一致しなかったこと，また定義の違いによって一致した数が変化したことを，結果として報告している．このことから，従来の単体・結合の定義は概念的かつ多義的であり，実際の開発者にとってそれらを区別することは容易でないと考えられる．加えて，単体・結合を自動で検出する方法およびツールは，われわれの知る限りでは存在しない．

他方，Kanstén はテストレベルという数値指標を定義し，これをもってテストケースの結合度を表現している [5]．実行時に呼び出す関数の数によってテストケースを数直線上に配置し，0 から一定間隔で数直線を仕切ってグループ分けする．そのときに何番目のグループに属するかによって，テストケースのテストレベルを決定する．定量的であるため自動計測が可能であるという利点はあるが，区切り間隔は可変パラメータとして扱われていて一意的でない．また，一次元の数直線上に各テストケースを射影するため，どのテストケースがどのテストケースに包含されているのかという，単体・結合が持つ自然な包含関係の情報を捨象してしまう．

## 3. テストケース間の包含関係

本研究の目的はテストケースの単体・結合の自動計測である．そのために，本節では包含関係，包含グラフ，包含レベルの 3 つの概念を定義する．2 つのテストケース間の包含関係を各テストケースの実行部分，すなわちカバレッジの包含関係として定義する．包含グラフは全てのテストケース間の包含関係を管理するためのデータ構造である．包含レベルはテストケースの結合度を表現する数値指標である．各概念の計測方法の流れを 図 4 の青の領域に示す．適宜参照されたい．

### 3.1 包含関係の定義

テストケース $T_A, T_B$ の包含関係次のように定義する．

$$T_A \subset T_B \stackrel{\text{def}}{=} \text{Stmt}(T_A) \subsetneq \text{Stmt}(T_B)$$

```
1  class UserRecord:
2      ...
3    Void registerUser(id, name):
4      if (validateID(id) && validateName(name))
5        userList.append((id, name))
6
7  // id が「英小文字＋数字3桁」かをチェック
8  Bool validateID(id):
9    return match(id, "\l\d{3}")
10 // name が英小文字だけかをチェック
11 Bool validateName(name):
12   return match(name, "\l[\l]*")
```

図 1　プロダクトコードの例

```
1  Void testRegisterUser():
2      record = new UserRecord()
3      record.registerUser("a001", "okamoto")
4      assert(record.findByID("a001"))
5  Void testValidateID():
6      assert(validateID("x099")).isTrue()
7      assert(validateID("0y99")).isFalse()
8  Void testValidtateName():
9      assert(validateName("alice")).isTrue()
10     assert(validateName("Alice")).isFalse()
```

図 2　テストケースの例

ここで，テストケース $T$ に対し $\text{Stmt}(T)$ は $T$ が通過するプロダクトコードの行の集合，すなわちカバレッジを表す．

具体例を示す．図 1 のプロダクトコード，および図 2 のテストコードが与えられたとする．自作レコードに指定の `id` と `name` でユーザ登録をするプロダクトである．登録は与えられた `id` と `name` の形式をチェックしてから行われる．テストケースは 3 つであり，登録処理を検証する `testRegisterUser`，`id` と `name` の形式をチェックする機能を検証する `testValidateID` と `testValidateName` からなる．このとき，各テストケースのカバレッジはそれぞれ次のように計測できる．

$$\text{Stmt}(\texttt{testRegisterUser}) = \{\, \ell4, \ell5, \ell9, \ell12, \dots \,\}$$
$$\text{Stmt}(\texttt{testValidateID}) = \{\, \ell9 \,\}$$
$$\text{Stmt}(\texttt{testValidateName}) = \{\, \ell12 \,\}$$

ただし，$\ell x$ は図 1 中における $x$ 行目を表す．カバレッジの包含関係により，これらのテストケースは次の包含関係を持つ．

$$\texttt{testValidateID} \subset \texttt{testRegisterUser}$$
$$\texttt{testValidateName} \subset \texttt{testRegisterUser}$$

この包含関係の考えに基づくと，他のテストケースの実行部分を一切含まないものが単体テストケースであり，他のテストケースをより多く含むものがより結合度の高いテストケースだと見做せる．従来の単体・結合の定義では，「単体」が指す対象の曖昧さが避けられなかった．テストケース間の包含関係を利用することにより，単体テストを絶対的ではなく相対的に定義し，この曖昧さを回避するのである．

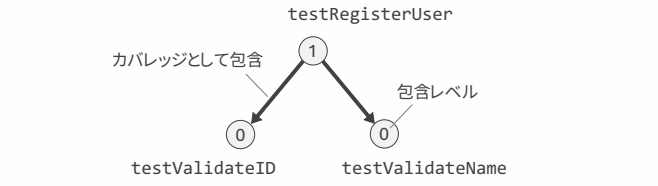


図 3　包含グラフの例

### 3.2　包含グラフと包含レベルの定義

テストケース $T_1, T_2, \cdots, T_n$ $(n \geq 2)$ を頂点とし，$T_i \subset T_j$ $(1 \leq i, j \leq n)$ のときに辺 $(T_j, T_i)$ を持つような有向グラフを，$T_1, T_2, \cdots, T_n$ の包含グラフと呼ぶ．

包含関係と有向辺の定義より，包含グラフは DAG（Directed Acyclic Graph）となる．DAG 上では各頂点の高さを一意に定義することができる．そこで，各テストケース $T$ に対応する頂点の高さを $T$ の包含レベルとして定義し，level($T$) と表す．具体的には次のように求められる．包含グラフ $G = (V, E)$ に対して各有向辺の向きを反転させた有向グラフ $G^R = (V, E^R)$ を得て，$T \in V$ の包含レベル level($T$) を次のように計算する．

$$\text{level}(T) = \begin{cases} 0 & \text{if } \text{Pred}(T) = \emptyset \\ \max\limits_{T' \in \text{Pred}(T)} \text{level}(T') + 1 & \text{if } \text{Pred}(T) \neq \emptyset \end{cases}$$

ここで，$\text{Pred}(T) = \{\, T' \in V \mid (T', T) \in E^R \,\}$ である．

先の具体例の包含グラフを図 3 に示す．それぞれの包含レベルを計算すると，対応するノード内の数値となる．

あるテストケースの包含レベルの大きさは，そのテストケースが他のテストケースをどの程度包含するかを表す．つまり，結合度を数値として表現しているといえる．最小値は 0 であり，他のテストケースを 1 つも包含していないため単体テストに相当する．最大値はプロジェクトに依存するが，数値が大きいほど結合度が高いことを意味する．

## 4.　包含関係を利用したテスト支援

定義した包含関係のソフトウェアテスト支援への活用を 2 つ提案する．「失敗テストケースが複数存在する場合の解決順序の提示（ORD）」と「結合寄りテストケースでしか通過されない箇所の報告（COV）」である．図 4 の緑と赤の領域に，ORD と COV の流れを示す．どちらも入力はプロダクトコードとテストケースである．図 4 の青の領域は共通の前準備を示す．包含関係の検出と包含グラフの構築，包含レベルの算出を行う．

### 4.1　ORD

失敗テストケースが複数存在する場合，どれから解決すれば失敗の原因となった欠陥箇所の特定労力が少なく済むかを開発者に提示する．出力は失敗テストケースの解決順序である．

失敗テストケースが複数存在する場合，より単体寄りなテストケースを優先して解決するべきだと考えられる．結合度の高いテストケースは複数の機能にまたがって広く検証するため，欠陥箇所の特定が難しいためである．よって，単体寄り，すなわち結合度の低いようなテストケースを解決対象として優先するために，包含レベルをキーとして昇順にソートする．

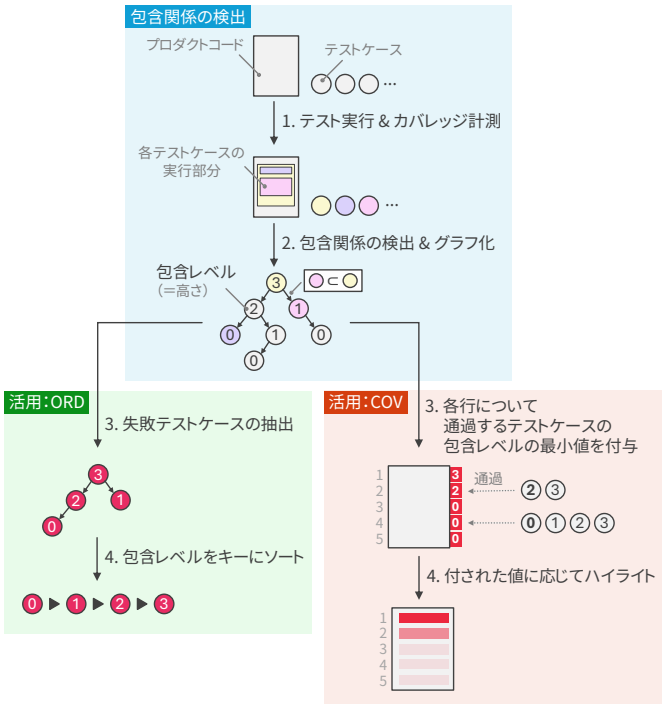


図 4　2 つの活用（ORD と COV）の流れ

レベルが同一なテストケースは複数存在しうる．その場合には，単に実行する行数が小さいものから解決すべきという素朴な考えに基づき，カバレッジ行数をキーとして昇順にソートする．つまり，結合度と通過した量の 2 つの観点で小さいものから順に解決すべきだと提案する．

手法の流れを図 4 における緑の領域に示す．はじめに，失敗テストケースからなる部分グラフを抽出する．次に，部分グラフにおいてテストケースの包含レベルを第 1 キーに，カバレッジ行数を第 2 キーにして昇順ソートする．ここで，テストケース $T$ のカバレッジ行数とは $|\text{Stmt}(T)|$，すなわち $T$ が通過するプロダクトコードの行数である．

### 4.2　COV

プロダクトコードのうち結合寄りテストケースでしか通過されない箇所を報告し，開発者に単体寄りテストケースへの細分化を促す．出力は結合度の高いテストケースでしか通過されない箇所の情報である．その表現として，プロダクトコードの各行にハイライトを施す．より結合度の高いテストケースでしか通過されない行はより濃い赤色でハイライトする．

結合テストはいくつかの機能の組み合わせを検証するものであり，各機能の動作を検証する責務はない．結合テストを行なった場合にのみ検証されるような機能は，その機能に対する単体テストで検証されておくべきである．なぜなら，欠陥箇所の特定が困難になりうるためである．ある 1 つの機能の中の，結合テストでしか通過されない箇所にバグがある状況を考える．このとき，単体テストは成功して結合テストだけが失敗することになる．実際はある 1 つの機能に問題があるのにも関わらず，結合テストの検証対象である機能の組み合わせ自体に問題があると開発者は勘違いしうる．結果，すぐには欠陥箇所を特定できなくなる可能性がある．

手法の流れを図 4 における赤の領域に示す．プロダクトコードの各行について，そこを通過するテストケースの包含レベルの最小値を記録する（以降，行の包含レベルと呼ぶ）．そして，その値の大きさに応じて赤色のグラデーションでハイライトを行う．行の包含レベルが大きいほど濃くハイライトする．

## 5. 評価実験

提案手法の有効性を確認するために OSS プロジェクトを対象とした 3 つの実験を行う．1 つ目の実験では，定義した包含関係が実在するかを確認する．実在すれば，包含関係を検出することによって一般的に定義される単体・結合の関係を検出できるかを確認する．2 つ目は ORD についての実験である．提案手法による失敗テストケースの解決順序がデバッグ時の労力を削減できるかを確認する．比較対象には，実行時に通過する行数が小さいものから解決すべきということを素朴に実現するカバレッジ行数昇順を選ぶ．3 つ目は COV についての実験である．実際のプロジェクトにおいて，結合度の高いテストケースでしか通過されない行がどの程度存在するかを確認する．また，提案手法によって結合度の高いテストケースでしか通過されないと報告された箇所について，そこを通過するような単体寄りテストケースを作るべきであるかを考察する．

題材には Java プロジェクトである jsoup[3] と Gson[4] の 2 つを用いる．選定理由は GitHub 上の Star 数が jsoup が 10,485，Gson が 22,757（2024 年 1 月 29 日時点）と比較的多く，多くの人に利用されているためである．また，手法では JUnit と JaCoCo を利用するため，利用の簡単のためにビルドツールとして Maven[5] が使用されているものを選んだ．

### 5.1 包含関係の検出

**評価方法：** 対象プロジェクトに含まれる全てのテストケースの包含レベルを計算し，各レベルごとのテストケースの数を確認する．包含関係が存在すれば具体的にいくつかをサンプリングし，一般的な単体・結合の関係に合致するかを考察する．考察の際にはテストケースの名前，テストコードの記述，プロダクトコードの記述を参考にする．

**結果：** 図 5 に Gson の一部のテストケースからなる包含グラフの様子を示す．全てのテストケースでは数が膨大で視認性が低いため，2 つのテストクラス `MapTest` と `JsonWriterTest` 内で定義されたテストケースに限定している．表 1 に jsoup と Gson における，全てのテストケースの包含レベルの分布を示す．これらの結果から，定義した包含関係は実在することが確認できた．また，レベルが 2 以上のテストケースの存在から，多段階の包含が存在することも確認できた．

ここで，図 5 で色付けしたノードに対応するテストケースとそれらの包含関係をピックアップする．

A. `MapTest#testSerializeMapOfMaps`

B. ⊃ `JsonWriterTest#testDeepNestingObjects`

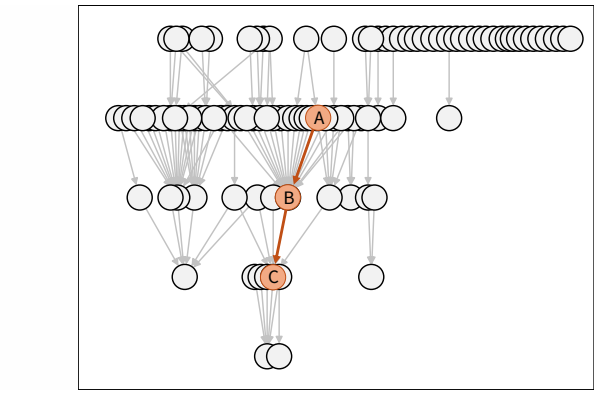


図 5 Gson の包含グラフの一部抜粋

C. ⊃ `JsonWriterTest#testEmptyObject`

B と C は JSON オブジェクトを文字列でダンプするクラスである `JsonWriter` を検証するテストケース，A は `java.util.Map` と JSON オブジェクトとを相互変換する機能を検証するテストケースである．それぞれの名前から，包含されるにつれてより特殊な機能のみを検証していることが推測できる．実際にテストコードを確認すると，各テストの名前が表す事柄を検証していることが確認できた．A に対しての B，B に対しての C がそれぞれの役割において先のテストケースの一部になっていて，これらの間にはテストとしての意味的な包含関係があるといえる．また，A は `functional` というテストディレクトリ下に配置されていた．このことから，A は一般的な結合テストのうちの機能テストに相当するものであると考えられる．一方で，B と C は `JsonWriter` に対する単体テストであると考えられる．したがって，定義した包含関係は一般的な単体・結合の関係を内包しうるものであり，包含関係の検出によって単体・結合の関係の一部を検出できるといえる．

サンプリングした包含関係の多くが上記のような関係を示していた．特に，B と C のような，同一クラスに対するテストケース間の包含関係が多く検出されていた．

### 5.2 ORD

**評価方法：** ミューテーション解析により実験を行う．まず，プロダクトコードの特定のパッケージ下に Mutanerator [6] を使ってランダムに 2 または 3 個のミューテーション（バグ）を仕込んでミュータントを生成し，テストケースを 2 個以上失敗させる．次に，提案手法（proposal）とカバレッジ行数昇順（naive）を適用して解決順序を決定し，後に定義する評価指標を計算する．上記を複数回繰り返し，比較する．

評価指標として，与えられた失敗テストケースの解決順序 $F = (F_1, F_2, \cdots, F_n)$ に対するデバッグ労力値 effort($F$) を定義する．$F_1$ から $F_k$ $(k \leq n)$ までの各テストケースが通過する部

表 1 各 OSS における全てのテストケースの包含レベル分布

| | 包含レベル | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| jsoup | 80 | 106 | 739 | 129 | 68 | 9 | 0 | 0 |
| Gson | 148 | 83 | 253 | 462 | 163 | 50 | 13 | 2 |

（注3）：https://github.com/jhy/jsoup
（注4）：https://github.com/google/gson
（注5）：https://maven.apache.org

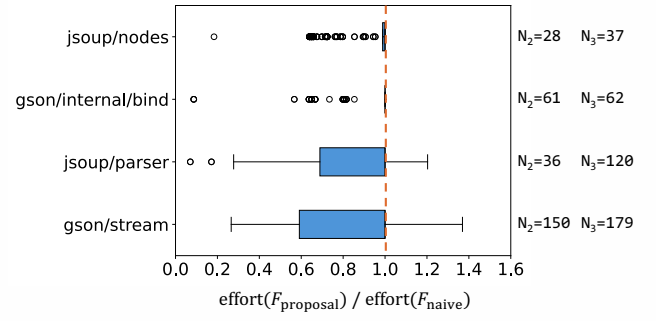


図 6　ORD：naive のデバッグ労力値に対する proposal のそれの比

分を確認してはじめて，全てのミューテーション行を発見できたとする．このとき effort($F$) を次のように計算する．

$$\text{effort}(F) = \sum_{i=1}^{k} \left(3.20 \times |\text{Stmt}(F_i)|^{1.05}\right)$$

ここで，式 $3.05 \times |\text{Stmt}(F_i)|^{1.05}$ はソフトウェア見積もりの工数計算における労力値の定義を参考にしている [7]．カバレッジに重み付けを行なった理由は，欠陥箇所の特定労力とプロダクトコードの行数の関係性は線形ではないためである．単純な分量増加だけでなく，分量増加に伴うコードの構造の複雑化によっても，労力は大きくなると考えられる．

**結果：** ミューテーション対象パッケージには `jsoup/nodes`, `gson/internal/bind`, `jsoup/parser`, `gson/stream` の 4 つをランダムに選んだ．4 つに限定した理由は実験時間の都合である．生成した 1 つのミュータントごとに 1 テストケースずつビルドと実行を行うので，膨大な時間を要するためである．

結果を図 6 に示す．縦軸は naive のデバッグ労力値 effort($F_{\text{naive}}$) に対する proposal のそれ effort($F_{\text{proposal}}$) の比を表す．ミューテーション対象パッケージに対して，バグを 2 個含むミュータントを $N_2$ 個，3 個含むものを $N_3$ 個生成し，各ミュータントに対してデバッグ労力値の比を計算して 1 つの箱ひげ図にした．結果としては，バグ数が 2 個の場合は平均して約 8%，3 個の場合は約 13%，総合で約 10%，proposal の方が naive よりもデバッグ時の労力が小さいと確認できた．

しかし，生成したミュータントの約 20% に対しては，proposal の方が naive よりもデバッグ時の労力が大きかった．これは，仕込んだミューテーション行を全て通過し，かつカバレッジ行数が比較的小さい失敗テストケース $T_{\text{all}}$ が存在したためだと考えられる．$T_{\text{all}}$ に比べ，1 つのミューテーション行しか通過しないがカバレッジ行数は大きく，包含レベルは小さいようなテストケース $T_{\text{one}}$ が存在する場合を考える．このとき，proposal はカバレッジ行数よりも包含レベルの小ささを優先するため $T_{\text{one}}$ を先に解決すべきと提案する．一方で naive はカバレッジ行数の小さい $T_{\text{all}}$ を先に解決すべきとする．結果，naive の方が少ないカバレッジ行数で全てのミューテーションを発見でき，デバッグ労力値が小さくなったと考えられる．

### 5.3　COV

**評価方法：** プロダクトコードの全行に対する行の包含レベルの内訳を確認する．単体テストケースによって通過される

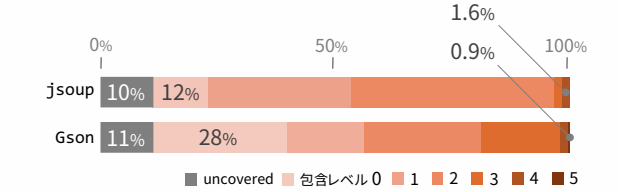


図 7　COV：プロダクトコードの全行に対する包含レベルの内訳

```
291.  */
292. public static URL resolve(URL base, String relUrl) thr
293.     relUrl = stripControlChars(relUrl);
294.     // workaround: java resolves '//path/file + ?foo'
295.     if (relUrl.startsWith("?"))
296.         relUrl = base.getPath() + relUrl;
297.     // workaround: //example.com + ./foo = //example.
298.     URL url = new URL(base, relUrl);
299.     String fixedFile = extraDotSegmentsPattern.matcher
```

図 8　COV：jsoup の一部のプロダクトコードに対するハイライト結果

行や結合度の高いテストケースでしか通過されない行がどの程度存在するかを観察する．また，高い包含レベルが付与された行を具体的に取り出し，そこにバグを加える．その結果，どのようなテストケースが失敗するかを確認する．

**結果：** 図 7 にプロダクトコード全行に対する行のレベルの内訳を示す．従来のカバレッジ計測では，図 7 における uncovered か否かの情報しか提示しない．提案手法ではそれに加え，通過された行がどの結合度のテストケースによって通過されたかの情報も提示している．レベル 0 が付された行は jsoup が 12%，Gson が 28% であり，Gson の方が比較的多くの行が単体テストケースで通過されていることが確認できた．一方で，jsoup ではレベル 4 の行が 1.6%，Gson ではレベル 5 の行が 0.36% 存在し，結合度の高いテストケースでしか通過されない行が存在することも確認できた．

図 8 に提案手法によってハイライトされたプロダクトコードの一部抜粋を示す．jsoup で定義されたあるメソッドの一部[6]である．全てのテストケースの実行によりメソッド内のカバレッジは 100% となる．しかし，ハイライトの様子から 296 行目や 301 行目は他の行と比べ，結合度の高いテストケースでしか通過されないことがわかる．そこで，図 9 に示すテストケースを加えて再び提案手法を適用したところ，図 10 のような結果が得られた．296 行目のハイライトがテストケース追加前よりも薄く，他の行と同じ濃さでハイライトされることが確認できた．適切にテストケースを加えれば，結合度の高いテストだけで通過されるような箇所を削減できるといえる．

図 8 において，296 行目に次のような変更（バグ）を加える．

```
relUrl = base.getPath() /* + relUrl */ ;
```

この状態でテストケースを追加する前と後，それぞれでテストを実行し，どのようなテストケースが失敗するかを確認した．結果，追加前は `NodeTest#absHandlesRelativeQuery`（D.）というテストケースが，追加後はこれと追加したものが失敗した．テストケースを追加せずともバグの存在は検知できるが（つまり少なくともテストケース D が失敗するが），D の名前

(注6)：https://github.com/jhy/jsoup/blob/master/src/main/java/org/jsoup/internal/StringUtil.java#L294-L306（2024 年 2 月 6 日 時点．実験の時点からコミットが行われて行番号が少しずれていることに注意．）

```java
@Test
public void added_resolvesRelativeUrls() {
    URL baseUrl = new URL("http://a.com");
    String relUrl = "?b";
    String expected = "http://a.com/?b";
    String actual = resolve(baseUrl, relUrl)
                    .toString();
    assertEquals(expected, actual);
}
```

図 9　COV：追加したテストケース

```
291.  */
292.  public static URL resolve(URL base, String relUrl) thr
293.      relUrl = stripControlChars(relUrl);
294.      // workaround: java resolves '//path/file + ?foo'
295.      if (relUrl.startsWith("?"))
296.          relUrl = base.getPath() + relUrl;
297.      // workaround: //example.com + ./foo = //example.c
298.      URL url = new URL(base, relUrl);
299.      String fixedFile = extraDotSegmentsPattern.matcher
```

図 10　COV：テストケースを追加した後のハイライト結果

からはバグが `resolve` メソッドに潜んでいるかを即座に判断することは容易ではない．単純にそれぞれのカバレッジ行数を比較しても，D は 1373，追加したメソッドは 13 と大きな差が存在した．以上のことから，提案手法の報告結果としてより濃くハイライトされる行について，そこを通過するような単体寄りテストケースを用意すべきといえる．

## 6. 議　　論

### 6.1 包含関係の利点

テストカバレッジ，つまりは実行したプロダクト行の集合という数学的対象としてテストケースを解釈することで，集合の包含関係という自然な概念をテストケース間の包含関係として利用した．テストカバレッジは実務においても広く利用されるメトリクスであり，多くの場面で容易に収集することが可能である．本研究で定義した包含関係およびその活用は特定の言語や場面に縛られず，計測および利用することができる．

また，テストケースの実体をそのカバレッジとして捉えることは，テストケースの同値性の考えと親和性がある．テストケースが固有するパラメータの具体的な違いを無視して，そのカバレッジだけを見てテストの同等性を定義しているといえる．ここで，テストケースが固有するパラメータとは，テストケース内でテスト対象のメソッド等に与える数値や文字列などの引数のことを指す．したがって，提案した包含関係は，具体的な計測方法を提示しながらも，テストケースの持つ具体的なパラメータに左右されにくい包含関係を表現する．

### 6.2 包含関係の妥当性

一方で，提案した包含関係が既存の単体・結合の関係に完全に合致するかには議論の余地がある．包含レベルの大小が結合度を表すことは直観的には認められるが，反例がいくつか考えられる．例えば挿入ソートに対するテストケースとして，整列済みのデータ（sorted）を入力として与える場合とランダムに並べられたデータ（randomized）を入力として与える場合を考える．このとき，文献 [1] [2] でのような一般的な単体・結合の考え方によれば，どちらのテストケースもソートという 1 つのメソッド，機能に対するテストであり，プロダクト全体で見た上では単体テストに相当する．しかし，挿入ソートの多くの実装では randomized の実行部分が sorted の実行部分を真に含んでしまうため，2 つの間には包含関係が検出されてしまう．結果，期待されている以上に細かく単体・結合を区別したり，あるいはその逆が発生したりする可能性がある．

また，提案手法はテストケースが十分に用意されていることを前提とする．単体テストに相当するようなテストケースが存在せず，結合テストのような検証範囲が広いテストしか用意されていないプロジェクトを考える．このとき，従来の定義で結合テストと分類されるようなテストケースでも，他のテストケースを包含しないために包含レベルとして 0 が割り当てられてしまう．結合度を相対的に計測するがゆえに，全てのテストケースがある程度存在することを前提とするのである．

## 7. おわりに

定義した包含関係は一般的な単体・結合の関係の一部を内包することを確認した．自動計測が可能であり，テストケースの結合度を定量化する．また，定義した包含関係の活用先として ORD と COV の 2 つを提案した．評価実験により，これらのデバッグ支援としての有効性を確認した．

今後の課題は大きく 2 つある．1 つ目は実験の拡充である．別のプロジェクトを対象とする，あるいは観察時のサンプリング数を増やすなどして，実験の妥当性を確保する必要がある．2 つ目は定義した包含関係の妥当性検証である．本研究では，包含関係の検出によって一般的な単体・結合の関係を検出を試みているが，これらは必要十分の関係にはない．そのため，包含関係と一般的な単体・結合関係との関係性調査や，包含関係の定義の調整を行う必要があると考える．

**謝辞**　本研究の一部は，JSPS 科研費（JP21H04877, JP20H04166, JP21K18302, JP21K11829）による助成を受けた．

## 文　　献

[1] V. Khorikov, Unit Testing Principles, Practices, and Patterns: Effective Testing Styles, Patterns, and Reliable Automation for Unit Testing, Mocking, and Integration Testing with Examples in C#, Manning Publications, 2021.

[2] IEEE, “ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary,” ISO/IEC/IEEE 24765:2010(E), pp.1–418, 2010.

[3] T. Winter, T. Manshreck, and H. Wright, Software Engineering at Google: Lessons Learned from Programming over Time, Oreilly & Associates Inc, 2020.

[4] F. Trautsch and J. Grabowski, “Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects,” Proc. IEEE International Conference on Software Testing, Verification and Validation, pp.207–218, 2017.

[5] T. Kanstrén, “Towards A Deeper Understanding of Test Coverage,” Journal of Software Maintenance and Evolution: Research and Practice, vol.20, no.1, pp.59–76, 2008.

[6] 佐々木唯，肥後芳樹，柗本真佑，楠本真二，“プログラムに対する欠陥限局の適合性計測，” 情報処理学会論文誌，vol.62，no.4，pp.1029–1038，2021.

[7] B.W. Boehm, “Software Engineering Economics,” IEEE Trans. Software Engineering, vol.SE-10, no.1, pp.4–21, 1984.