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

ミューテーション解析の容易化を目的とした段階的なミュータント抽象化の検討

石坂　颯大† 　柗本　真佑† 　楠本　真二†
安田　和矢†† 　伊藤　信治†† 　代　　吉楠††

† 大阪大学大学院情報科学研究科
†† 株式会社 日立製作所
E-mail: †{s-isizak,shinsuke}@ist.osaka-u.ac.jp

あらまし　ソフトウェアテストの品質評価手法としてミューテーション解析が活用されている．ミューテーション解析では，意図的にバグを混入させた複数のソースコード（ミュータント）を生成し，それらをテストで検出できるかを確認する．検出されず生存したミュータントは既存のテストの不十分さを示唆する．そのため開発者はそれらを分析してテストの改善を行う．しかしテスト不備とミュータントの間には意味的な乖離があり，開発者の類推によって補われている．これはテスト不備の原因理解や改善方針の検討を困難にし，分析コストの増加につながる．本研究ではテスト不備の解析の容易化を目的として，テストから生存した複数のミュータントを段階的に抽象化する手法を提案する．単一のテスト不備に起因する複数の生存ミュータントを段階的にまとめていくことで，単一不備に対する単一ミュータントの獲得を試みる．本研究会原稿では提案手法を擬似題材に対して適用することで，解析の容易化への有用性を検討する．

キーワード　ミューテーション解析，ミューテーションテスト，ミュータント，抽象化

1. はじめに

ソフトウェアテストの品質評価手法としてミューテーション解析 [**?**] が活用されている．ソースコードのふるまいをテストで保証するのと同様に，テストの動作も何らかの方法で保証されるべきである．ミューテーション解析のテスト不備の検出能力はテストの十分性を評価する指標として代表的なカバレッジよりも高い [1] [2].

ミューテーション解析では，意図的にバグを混入させた複数のソースコード（ミュータント）を生成し，それらをテストで検出できるかを確認する．テストが様々なケースを十分に網羅していればこのミュータントを正しく検出することができるはずである．逆に検出されずテストから生き残ったミュータントは既存のテストの不十分さを示唆する．このミュータントはバグが含まれており，既存のテストではこのバグを検知できていないためである．そのため開発者はそれらを分析してテストの改善を行う．テストから生き残ったミュータント（以下，生存ミュータント）のバグの内容を手掛かりに，どんなテストがあればこのミュータントを検知することができるのか，今はどんなテストが不足しているからこのミュータントを捕捉できないのかを類推する．

しかし，生存ミュータントはテスト不備を直接的に示すわけではない．あくまでも生存ミュータントはソースコードを書き換えた人工的なバグであり，テスト不備とは意味的な乖離がある．開発者はバグの箇所や内容から推測しながら分析を行う．単一の不備に起因する複数の生存ミュータントからテスト不備を導き出せるかはこの意味的な乖離を開発者の類推の能力で補えるかどうかに依存している．

そこで本研究ではミューテーション解析におけるテスト不備の分析の容易化を目的として，複数の生存ミュータントを段階的に抽象化・統合する手法を提案する．ミューテーション解析で生き残った複数のミュータントを，できるだけ少数でテスト不備を直接的に表現するミュータントを得ることを目指す．テスト不備を伝えるために多数のミュータントを用意するのではなく，より直接的な少数のミュータントであらわすべきである．一般的なミューテーションテストにおける生存ミュータントをより抽象度の高いミュータントに変形し，同形となったミュータントを統合することで，より本質的な少数のミュータントの獲得が期待できる．またこの操作を段階的に少しずつ行うことで，いきなりソースコードの広範囲に影響してしまうミュータントを作成しないようにしている．

本研究会原稿では提案手法を擬似題材に対してケーススタディを行うことで，解析の容易化できているかや今後どのように提案手法を改善していくかを議論した．結果として題材に対して目標としていたミュータントと同一のミュータントを得ることができた．一方で手法には時間的なコストの面で依然として改善が必要であるという結果となった．

2. ミューテーションテストとその課題

2.1 ミューテーションテスト

ミューテーションテストとはテストの品質を評価する手法である．直感的には人工的かつ微細なバグをソースコードのあらゆる箇所に埋め込んで，その全てをテストで検出できるかを確かめる，というイメージである．このようにバグを埋め込まれたソースコードのことをミュータントと呼ぶ．もし全てのミュータントを検出できるならテストは完璧だが，ミュータントを見逃すならバグを見逃す，つまり何かしらのテスト不備があるということを示す．以降ではこの検出されたミュータントをキルされたミュータント，見逃されたミュータントを生存ミュータントと呼ぶ．

ミューテーションテストは伝統的なカバレッジよりもテスト不備の検出能力が高い [1] [2]．カバレッジはソースコードが実行されたかどうかしか評価せず，期待値の設定ミスや検証不足といったテストケース設計上の不備を見逃しやすいためである [2]．またミューテーションテストは研究用途にとどまらず実践的利用も進みつつある [3]．言語ごとにミューテーションツールが開発されており，Python の mutmut，Rust の cargo-mutants，Java の PIT [4] などが挙げられる．

ミューテーションテストの具体例を示す．題材は Python で記述された FizzBuzz プログラムである． 図 1(a) のようなソースコードと 図 1(b) のような 4 つのテストがある．ただし 3 つ目のテスト `test_buzz()` は意図的に不十分な状態としている．これに対して mutmut でミューテーションテストを行うと，22 個のミュータントが生成され，そのうち 17 個のミュータントが検出され，5 個が生存ミュータント であった．このときの生存ミュータント の例を 図 2 に示す．

この場合 図 2 に示す生存ミュータントから 5 の倍数のテストが不十分であることが読み取れる．理由は生存ミュータントを以下のように分析できるからである． 図 2(a) や 図 2(b) のように 5 の倍数であるかどうかを判定する処理に対するミュータントや， 図 2(c) や 図 2(d) のように 5 の倍数であった場合に返すはずの文字列`"Buzz"`に対するミュータントが生き残っている．これらのミュータントが見逃されているのは，n が 5 の倍数であった場合のテストが不十分であるためだと予想できる．

2.2 Motivating Example

本節ではミューテーションテストの結果の理解に対する課題について，前節で述べた具体例に基づいて紹介する．ミューテーションテストではテスト不備を発見するために，開発者は生存ミュータント の改変の内容を調べる．これは生存ミュータント を検出するようにテストを改善することで，テストの品質が向上するからである．

しかし生存ミュータントからテスト不備を類推するのは容易ではない．なぜならば複数の微細なミュータントは直接的にテスト不備を示してくれるわけではないためである．このミュータントとテスト不備の意味的な乖離を開発者の類推により補っている．

この乖離は 図 2 にも表れている． 図 2 のミュータントはすべて「5 の倍数のテスト不足」という一つの不備からなる．実際 図 1(b) の 5 の倍数のテストは現状機能していない．この不備により，演算子 `%` を `/` に書き換えたり（ 図 2(a)），数値リテラルを 1 大きく（ 図 2(b)）してもミュータントは生き残る．また返り値の文字リテラルに余計な文字列を付け加えたり（ 図 2(c)），全文字を大文字化したミュータントも生き残ってしまう（ 図 2(d)）．ミューテーションテストではこれらの事実から「5 の倍数のテストが不足している」と類推しなければならない．

```python
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizBuz'
    if n % 3 == 0:
        return 'Fizz'
    if n % 5 == 0:
        return 'Buzz'
    return str(n)
```

(a) ソースコード

```python
def test_fizz_buzz():
    assert fizzbuzz(15)
        == 'FizBuz'

def test_fizz():
    assert fizzbuzz(3)
        == 'Fizz'

def test_buzz():
    assert fizzbuzz(5)

def test_number():
    assert fizzbuzz(1)
        == '1'
```

(b) テスト

図 1: ミューテーションテストの題材

```
--- src/fizzbuzz.py
+++ src/fizzbuzz.py
@@ -3,6 +3,6 @@
         return 'FizzBuzz'
     if n % 3 == 0:
         return 'Fizz'
-    if n % 5 == 0:
+    if n / 5 == 0:
         return 'Buzz'
     return str(n)
```

(a)

```
--- src/fizzbuzz.py
+++ src/fizzbuzz.py
@@ -3,6 +3,6 @@
         return 'FizzBuzz'
     if n % 3 == 0:
         return 'Fizz'
-    if n % 5 == 0:
+    if n % 6 == 0:
         return 'Buzz'
     return str(n)
```

(b)

```
--- src/fizzbuzz.py
+++ src/fizzbuzz.py
@@ -4,5 +4,5 @@
     if n % 3 == 0:
         return 'Fizz'
     if n % 5 == 0:
-        return 'Buzz'
+        return 'XXBuzzXX'
     return str(n)
```

(c)

```
--- src/fizzbuzz.py
+++ src/fizzbuzz.py
@@ -4,5 +4,5 @@
     if n % 3 == 0:
         return 'Fizz'
     if n % 5 == 0:
-        return 'Buzz'
+        return 'BUZZ'
     return str(n)
```

(d)

図 2: 生存ミュータント

このようにミューテーションテストの課題はミュータントとテスト不備の意味的な乖離により解析が困難になってしまうことである．もし単一不備に起因する複数のミュータントをより少数の抽象化されたミュータントに変換できれば，ミュータントの分析に要するコストすなわちミューテーション解析のコストの削減に繋がる可能性がある．

3. 段階的ミューテーションテスト

3.1 手法の概要

本研究の目的はミューテーション解析の容易化である．そのために段階的ミューテーションテストという新たな手法を提案する．提案手法のアイデアは，ミューテーション解析で生き残った複数のミュータントから，テスト不備を直接的に表現できる少数の（できれば単一の）ミュータントを得る，という点にある．このアイデアはピタゴラスの格言「多くの言葉で少しを語るのではなく少しの言葉で多くを語りなさい」に通ずる．あるテスト不備を伝えるためには，複数のミュータントで間接的に表現するのではなく，より直接的な少数のミュータントで表現すべきである．

例えば先の 図 2 に対しては最終的に 図 3 のようなミュータントを作成することを目指す． 図 3 は元のソースコード 図 1(a) から 5 の倍数を判定して"Buzz"を返す処理を削除している．これは 図 2 よりダイレクトに 5 の倍数のテストに不備があることを示唆している．これこそがピタゴラスの格言における「少しの言葉」である．このように不備を直接的に伝えてくれるミュータントの獲得を目指す．

提案する段階的ミューテーションテストは抽象化と統合の 2 つの操作で構成される．一般的なミューテーションテストで生き残ったミュータントを対象に，より抽象度の高いミュータントへの変形を試みる．変形されたミュータントに再度テストを適用し，生き残るのであれば変形（抽象化）に成功したと見なす．この操作を複数の生存ミュータントに適用していき，ソースコードが同形となれば統合しさらなる抽象化を加えていく．この操作を繰り返すことで少数の直接的なミュータントを得る．

```
def fizzbuzz(n):
    if n % 15 == 0:
        return 'FizzBuzz'
    if n % 3 == 0:
        return 'Fizz'
-   if n % 5 == 0:
-       return 'Buzz'
    return str(n)
```

図 3: 理想のミュータント

3.2 単一ミュータントの抽象化

まず抽象化のアイデア・コンセプトについて説明する．抽象化とは，ミューテーションによりソースコードが書き換えられたとき，この書き換えをより一般化したソースコードを得ることである．抽象化操作は元のミュータントよりも少し影響の大きいバグに進化させるイメージである．従来のミューテーションテストでは微小なバグを広範囲に埋め込むことで網羅的な分析を行っていた．これはソースコードを破壊しすぎないようにしつつ，ソースコード全体を分析するためであった．提案手法では段階的にミュータントをできるだけ大きくしていくことで，なるべくわかりやすいミュータントを生成する．

次に抽象化アルゴリズムを説明する．

- 抽象構文木（Abstract Syntax Tree, AST）を探索
- AST の葉ノードに変数や定数があればそれを定数に置換
- 兄弟ノードが抽象化済みの場合，その兄弟をまとめて置換
- 文の本文（代入文の場合は右辺，if 文/while 文の場合は本文）が削除済みなら文ごと削除

基本的な流れは上のとおりである．一般的なミューテーションツールにより改変が行われたノードの兄弟ノードが葉ノードであり，変数や定数ならばそのノードを定数に変換する．ここで「改変」とは例えば 図 2(a) で % を / に書き換えた操作を指す．ただしこの定数はプログラム中で意味をなさない定数とする．この走査を葉からスタートして段階的に抽象化を行っていく．走査が進むにつれて，式が丸ごと置換されたり，文が削除される．

このアルゴリズムを適用した簡単な例を 図 4 に示す． 図 4

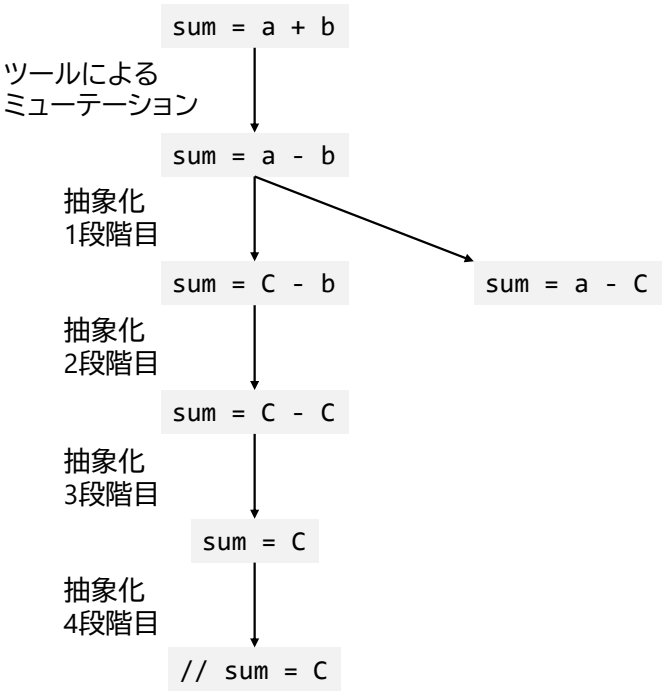

図 4: 抽象化の適用例

はソースコード中の sum = a + b に対して"+"を"-"に書き換えるミューテーションが行われた場合の例である．抽象化 1 段階目ではツールによりミューテーションが行われた"-"の兄弟ノードである変数 a を定数 C で置換している．C は前に述べたプログラムに無関係な定数である． 図 4 の右側のように変数 b を定数 C に置換する抽象化も考えられるが，これについては後述する．抽象化 2 段階目では sum = C - b と置換さ

れた式に対して，まだ置換されていない変数 b を置換する．3 段階目ではこれ以上置換できる葉ノードがないので兄弟をまとめて置換し，4 段階目では本文が C となっているため文ごと削除を行う．よって最終的には sum = a - b が削除されたミュータントが得られる．

実際には過度な抽象化を抑制するために抽象化を終了する条件を設定している．この条件がないと抽象化が終了することはなく，ソースコードの大部分を削除してしまうためである．この文脈での過度な抽象化とは，抽象化で新しく生成されたミュータントがテストで検出されてしまうことだ．なぜならば元々生存していた，つまり何かしらのテスト不備を暗示していたミュータントが，テスト不備を示さない状態になったといえるからである．これはすなわちソースコードへの影響力が大きすぎるバグを作成してしまったという意味でもある．

段階的に抽象化を行っている理由は過度な抽象化の抑制と関連している．なるべく大きい影響力を持つミュータントを作成したいが，生成したミュータントが大きすぎてテストで検知されては意味がない．そこで抽象化したらテストにより過度な抽象化ではないかを確認して，次の抽象化に進むという段階を経ることで，テストに検出されない最大のミュータントを得ることができる．

3.3 統　　合

抽象化を進めていくと，同一のミュータントが生成されるため，これらの統合を行う必要がある．例えば 図 4 右側の sum = a - C の抽象化を進めていくとこの問題が生じる．sum = a - C をもう一段階抽象化すると sum = C - C となり，sum = C - b を抽象化したミュータントと一致してしまう．

ミュータントを統合していくことで，ミュータント数を削減することができる．この削減は同一のテスト不備からなるミュータント同士で頻繁に発生する．なぜならば同一のテスト不備からなるミュータントの位置は，AST の同一階層付近である可能性が高いためである．例えば 図 2 のミュータントはすべて局所的な範囲に集中しているといえる．よって抽象化と統合を繰り返し行うことで徐々にミュータント数は減少する．最終的に不備の意味合いに近い少数のミュータントが得られる．

同じミュータントが生成されたことを確認した場合はその後は片方だけを考えていくものとする．将来的には効率的な統合手法を考える必要がある．

4. ケーススタディ

4.1 概　　要

提案手法のケーススタディを行った． 図 1 に対して mutmut によるミューテーションテストと提案手法を適用し，以下の観点から検証する．

・従来のミューテーションテストからどのようなミュータントが得られるか

・提案手法でどのようなミュータントが得られるか

・それをどう解析すればテスト不備の発見につながるか

題材は 図 1 でも挙げた FizzBuzz プログラムである． 図 1 は FizzBuzz のソースコードとそのテストから構成される．ただしテストは 5 の倍数のテストだけアサーションが機能しておらず，不十分な状態としている．mutmut により 図 2 のように改変された 4 つのミュータントを含む 5 つの生存ミュータントに対して抽象化・統合を行った．ただしケーススタディにおいては提案手法によるミュータント生成・統合は手動で行った．

4.2 結果と過程

最終的には，目標としていた 図 3 と同一のミュータントを得ることができた． 図 3 を目標としていたのは， 図 3 が 1 つのミュータントでテスト不備の内容を直接的に示しているためである． 図 3 は FizzBuzz プログラムから 5 の倍数の判定処理と 5 の倍数のときに"Buzz"を返す処理を削除したミュータントである．すなわち 図 3 からは 5 の倍数にかかわる処理がすべて削除されている状態である．このミュータントが生存したということは，5 の倍数のときのテストが正常に機能していないことを直接的に示すため，目標のミュータントとしていた．

また提案手法の途中経過を 図 5 と 表 1 に示す．まず 図 5 に

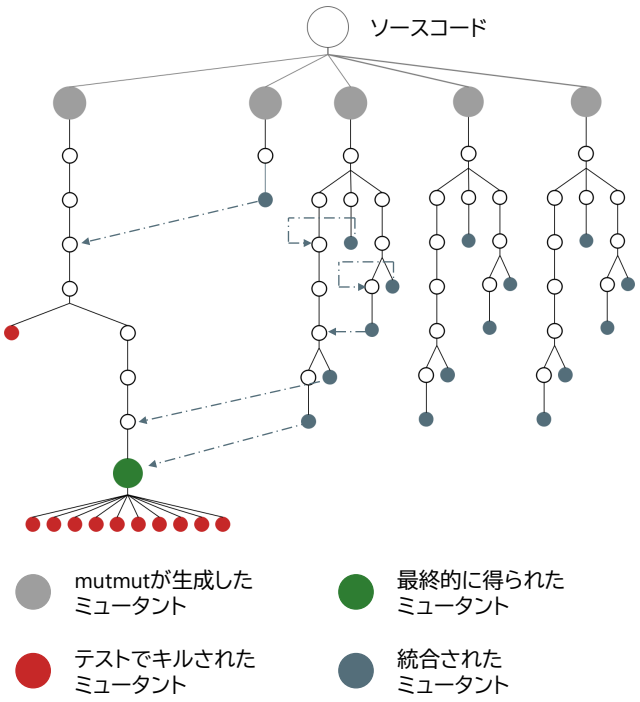

図 5: FizzBuzz に対する提案手法の実行過程

ついて説明する． 図 5 では丸がミュータントを表しており，どのミュータントからどのように抽象化・統合が進行しているかを木構造で示している．またこの木の深さは抽象化の段階を示している．例えば一番左のミュータントは 4 段階目までは 1 つのミュータントから 1 つのみの抽象化ミュータントが生成され，5 段階目では 2 つの抽象化に分岐している．分岐の左側の赤色のミュータントはテストでキルされたミュータントである．テストできるされたため過度な抽象化と判定されている．これは抽象度を上げすぎてしまったことを示している．

ブルーグレーの丸は統合されたミュータントを表しており，どのミュータントがどのミュータントに統合されたのかを点線の矢印であらわしている．ただし 図 5 の右側のミュータントには矢印を付与していない．これは中央のミュータントとほぼ同じであり省略しているためである．これについて詳しく説明する．5 個の mutmut により生成されたミュータントは if の条件文を改変したミュータントと return の返り値である文字リテラルを改変したミュータントに分類できる．同じ分類のミュータント同士は早いうちに統合されるはずである．例えば条件文を改変したミュータントは 図 6 のように抽象化・統合される．このように改変箇所が AST の同一階層付近であれば，

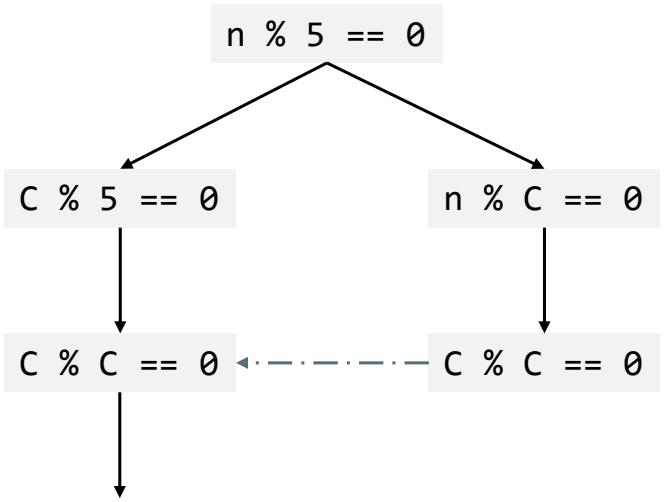

図 6: ミュータントの統合過程

早い段階で統合される．mutmut に生成された 5 つのミュータントのうち， 図 5 の右 3 つのミュータントのはすべて return の返り値である文字リテラルを改変したミュータントであったため，それらを抽象化したミュータントは全く同じ抽象化・統合の過程をたどった．

過度な抽象化によりテストでミュータントが検出されたり，すでに同一のミュータントが生成されている場合に統合されて終了するため，赤または灰色の丸が抽象化の終端となっている．ケーススタディでは左下の緑のミュータント以外はすべてテストで検出もしくは統合されたため，最終的に開発者に提示されるのはこのミュータントだけとなる．これが 図 3 と一致していた．

表 1: FizzBuzz に対する提案手法の実行結果

指標	値
抽象化前の生存ミュータント数	5
最終的に得られたミュータント数	1
抽象化で生成されたミュータント数	69
最大段階数	9
統合されたミュータント数	16
テストでキルされたミュータント数	11
mutmut によるミュータント生成・テスト実行時間 (s)	1.7
提案手法における処理終了までの実行時間 (s)	3.7

表 1 のようにミューテーションツール mutmut により生成された 5 つの生存ミュータントを 1 つの本質的なミュータントに抽象化・統合することができた．この結果にたどり着くまでに 69 個のミュータントが，最大 9 段階のフェーズで生成された．また 16 個のミュータントがすでに生成されているミュータントと同一であったために統合され，11 個のミュータントが過度な抽象化と判定された．また従来のミューテーションテストの実行時間は 1.7 秒なのに対し，提案手法で処理を終了させるまでに 3.7 秒を要した．従来のミューテーションテストの後に提案手法を実行するので，3.7 秒には mutmut の実行時間も含まれている．ただし本稿では提案手法のミュータント生成・統合を手動で行ったため，本来はそれらの時間の分が加算される．

5. ケーススタディに対する議論と研究の展望

5.1 計算コストの削減

想定していた理想の形と同形のミュータントを得ることができた一方で実行コストは改善の必要がある．FizzBuzz の短いプログラムに対しても提案手法の実行時間が mutmut の実行時間の 2 倍以上になっていた．ソフトウェアの規模が大きくなるほど個々のテスト実行に要する時間も増加する．加えてテスト自体の数も増加するため，全体的には指数的な増加となる．一般的なミューテーション解析はしばしば計算コストが課題とされており [5] が提案手法はこの問題をさらに悪化させている．なぜなら通常のミューテーションテストでミュータントを作成したのち，抽象化によりさらにミュータントを生成していくためである．

この問題に対するアイデアとして抽象化の省略が考えられる．今回は単一のミュータントを抽象化して，その後で同形となったミュータントをまとめるという戦略を取った．しかし複数ミュータントから抽象化した 1 つのミュータントを生むような操作をすればよい．この操作が過度な抽象化であったすなわちテストで検出されるミュータントを作成した場合は，逆に具象化することにより巻き戻しを行えばよい．多少省略を行っても問題ないと予想している．なぜならば現状は過度にならないようにかなり丁寧に抽象化しているためである．これは 図 5 からも読み取ることができる．ミュータントが抽象化でいきなりテストで検出されることはなく，白丸が多いことから安定な抽象化が行われていることがわかる．

また別の省略の手法としてミュータントの抽象化・統合の傾向に基づいて省略した抽象化も考えられる．頻繁に省略が行われるパターンを見つけて，このパターンならこの段階まで省略できる，と突き止めることができれば処理の高速化につながる．例えば 図 4 で挙げていた `sum = a - b` の抽象化では `sum = C - b` という段階を踏んでから `sum = C - C` そして `sum = C`，最終的に`// sum = C` としていた．しかし `sum = C - b` が生存したということはおそらく `sum` は正しくテストされていないことが予想されるので， 図 7 のようにその次の抽象化を `sum = C`，あるいは`// sum = C` としても問題ないケースが多いのではないだろうか．このように「四則演算の片方の

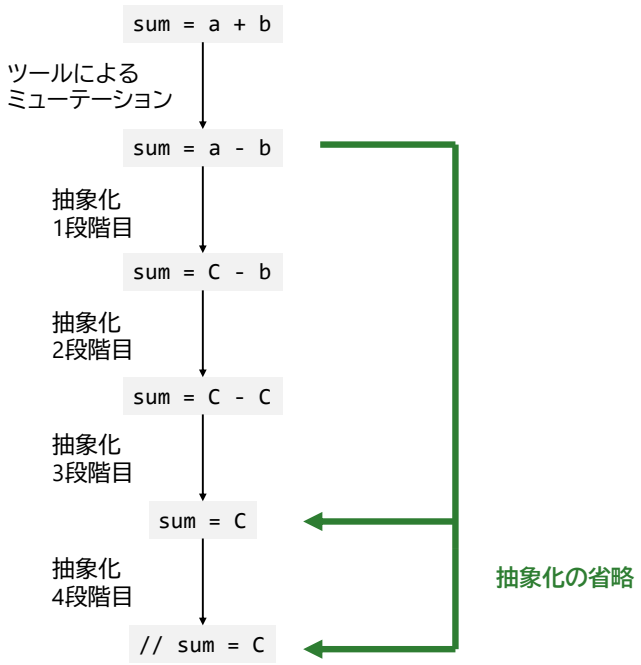

図 7: 抽象化の省略の例

被演算子を抽象化してもミュータントが生存するならば，演算全体を抽象化してよい」などのパターンが発見できると可能性がある．

処理の高速化に対する別のアプローチとして，抽象度の高いミュータントからキルされるミュータントを生成しすぎないようにすることも考える必要がある．これはある段階を超えるととたんに生成されるミュータント数が増加してしまうことが原因である．ケーススタディではこの事象が発生してしまった． 図 5 の左下のように，1 つのミュータントから 10 個のミュータントが生成され，すべてキルされている．

これは兄弟のノードを抽象化するときに，兄弟ノードが抽象化されていないなら兄弟ノードの葉ノードから抽象化しているためである． 図 8 を用いて具体的に説明する． 図 8 は抽象化がかなり進行したミュータントの構文木である．この場合の次の抽象化の対象は 図 8 のすべて葉ノードである．なぜなら削除したノードと兄弟であるノードの葉ノードが削除後の抽象化の対象となるからである．

しかしこの抽象化がないと `n % 15 == 0` に対して，`n % 15 == 1` のような生存ミュータントがあるときに次の抽象化が段階的な `C % 15 == 1` ではなくいきなり `C == 1` となってしまう．このように抽象度の高いミュータントに対しては，兄弟ノードの葉ノードに対する抽象化は改善する必要がある．

5.2 被験者実験

また提案手法によりミューテーション解析が容易になったかを確かめるために被験者実験が必要である．本稿ではケーススタディにより，主に提案手法の動作の過程を検証した．実際に提案手法により解析が容易になったか，という感覚的な指標を評価するために被験者実験を行う必要がある．提案手法がある場合とない場合で被験者に評価を行ってもらう．

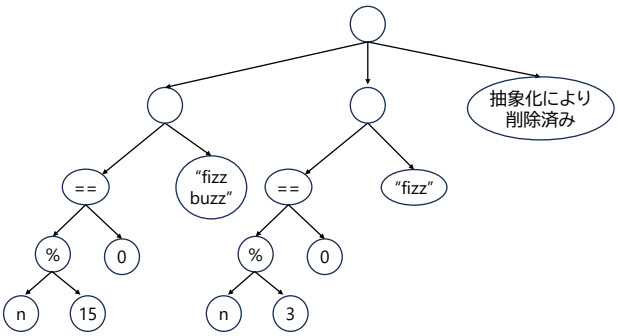

図 8: 抽象度の高いミュータントの構文木

6. おわりに

本稿ではミューテーションテストにおけるミュータントとテスト不備の意味的な乖離によりテスト不備の原因理解が困難になるという問題に対し，テスト不備を直接的に表現できる少数のミュータントを得る手法を提案した．その提案の第一歩としてケーススタディを行い，どのようなミュータントが得られ，それが本当に解析の容易化に貢献しているかを検証した．結果としてケーススタディにおいては望んでいたテスト不備を直接的に示す単一のミュータントを得ることに成功した．

今後取り組むべき課題は抽象化手順の改善である．なぜなら現在の手法では計算コストが非常に大きくなることが予想できるためである．本稿での手法のように丁寧に抽象化していくだけでなく，複数ミュータントから一気に抽象化を試みる．生成されるミュータントの数を減らすことでテストの実行時間を削減できる．そして実装を行った後に被験者実験により提案手法により解析が容易になったかを評価する．

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] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Does mutation testing improve testing practices?,” In Proceedings of the 43rd International Conference on Software Engineering, pp.910–921, 2021.

[2] R. Baker and I. Habli, “An empirical evaluation of mutation testing for improving the test quality of safety-critical software,” IEEE Trans. Softw. Eng., vol.39, no.6, pp.787–805, 2013.

[3] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y.L. Traon, and M. Harman, Chapter Six - Mutation Testing Advances: An Analysis and Survey, Elsevier, 2019.

[4] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “PIT: a practical mutation testing tool for Java (demo),” In Proceedings of the 25th International Symposium on Software Testing and Analysis, pp.449–452, 2016.

[5] M.B. Bashir and A. Nadeem, “Improved genetic algorithm to reduce mutation testing cost,” IEEE Access, vol.5, pp.3657–3674, 2017.