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

Peacefulテストの解析による 不足したミューテーション演算子検出の試み

後藤　有希†　　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-gotou,shinsuke}@ist.osaka-u.ac.jp

あらまし　ミューテーションテストはプログラムに人工的な改変を組み込むことでテストスイートの十分性を評価する手法である．しかし，既存のミューテーション演算子は十分であるとは限らない．例えば Python においては動的型付け言語特有の欠陥を十分に表現できていないことが指摘されている．本研究では，ミューテーション演算子の不足を発見する体系的な方法の獲得を目的として，ミュータントの検出に貢献していないテストに着目した解析手法を提案する．本研究ではこのようなテストを Peaceful テストと定義する．一般的には，生成されたミュータントの集合が欠陥を十分に代表しているとみなし，テストスイートの評価や改善が行われている．これに対し，テストスイートが十分であるという前提に立てば，Peaceful テストの存在はその検証範囲にミュータントが生成されていないこと，すなわちミューテーション演算子の不足を示すと解釈できる．この点に注目し，Peaceful テストを収集・分析することで不足しているミューテーション演算子を探す．

キーワード　ソフトウェアテスト，ミューテーションテスト，ミューテーション演算子，Peaceful テスト

1. はじめに

テストスイートの十分性を評価するための手法として，ミューテーションテスト [1] が注目を集めている．ミューテーションテストでは，人工的なバグが埋め込まれたミュータントと呼ばれるプログラムをテストスイートによって検出できるかを確かめる．ミュータントがテストスイートによって検出できない場合は，テストスイートがバグ検出の観点で不十分だとみなされる．生成されたミュータントの数に対する検出されたミュータントの割合はミューテーションスコアと呼ばれる [2]．このミューテーションスコアはテストスイートの欠陥検出能力を示す指標となり，実際のバグ検出率との相関は従来の代表的な指標である命令網羅率よりも高いことが報告されている [3].

ミューテーションテストの有効性はどのようなバグを埋め込むのか，すなわちミュータントの生成方法に強く依存する．このミュータントを生成する際の規則をミューテーション演算子と呼ぶ．広く用いられているミューテーション演算子の例としては，加算演算子 + から減算演算子 - への変更や，関係演算子 >= から > への置換のような単純な改変である．質の高いミューテーションテストを行うためには，このミューテーション演算子の適切な設計が不可欠である．これまでに Java [4] [5] [6] や C/C++ [7] [8] を中心とした様々なプログラミング言語に対してミューテーションツールが提案されており，それぞれの言語特性や想定する欠陥に応じたミューテーション演算子が組み込まれている．

しかし，既存のミューテーションツールに組み込まれているミューテーション演算子は十分であるとは限らない．実際に，特定の欠陥や言語特性を対象とした新たなミューテーション演算子の提案は継続的に行われている [9] [10] [11]．これは既存のミューテーション演算子のみでは現実の欠陥を十分に表現できていない可能性を示唆している．例えば Python においては，動的型付け言語特有の欠陥を十分に表現できていないことが指摘されている [12]．加えて，不足しているミューテーション演算子を体系的に発見することは容易ではない．そのためのアプローチとして実際の欠陥や典型的な誤りの分析に基づく手法が広く採用されている [10] [12] [13] が，現実には複雑な欠陥も多く，手作業による分析には多大なコストを伴う．したがって，より効率的に不足しているミューテーション演算子を探すためには，欠陥データの分析に依存しない新たな観点からのアプローチが必要となる．

本研究では，ミューテーション演算子の不足を発見する体系的な方法の獲得を目的として，ミュータントの検出に貢献していないテストケース(注1)に着目した解析手法を提案する．本研究ではこのようなテストを Peaceful テストと定義する．一般的にミューテーションテストを活用する際には，生成されたミュータントの集合が検証対象の欠陥を十分に代表しているとみなし，検出されなかったミュータントの存在はテストスイートの不十分さを示していると解釈される．これに対し，テストスイートが十分であるという前提に立てば，この解釈は逆転する．すなわち，Peaceful テストの存在はそのテストの検証範囲にミュータントが生成されていないことを意

(注1)：以降，本稿ではテストケースを単にテストと表記する．

味し，ミュータントを生成するミューテーション演算子が不足していると解釈できる．したがって，本研究では Peaceful テストの収集・分析によりこのような事例を抽出し，不足しているミューテーション演算子の特定を試みる．

本稿では，提案手法の有効性を評価するために，Python 向けミューテーションツールである mutmut を対象として Peaceful テストの解析を行い，不足しているミューテーション演算子を調査する．10 プロジェクトを対象とした調査を行った結果，ミューテーション演算子の不足が原因である Peaceful テストは 207 件発見され，これらの分析により不足しているミューテーション演算子を 17 個発見することに成功した．

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はテストスイートの十分性を評価する手法として広く研究されている [2] [14]．ミューテーションテストでは，人工的なバグが埋め込まれたミュータントと呼ばれるプログラムを大量に生成する．ミュータントの生成には，ミューテーション演算子と呼ばれる事前定義されたソースコードの変更規則を用いる．例えば，加算演算子 + から減算演算子 - への置換，定数<C>から<C>+1 への変更，さらに代入文の右辺や戻り値を `0` や `null` に置換する操作などが広く採用されている．ミュータントが生成された後はそのミュータントに対してテストスイートを実行し，ミュータントを検出できるかを判定する．あるミュータントに対して少なくとも一つのテストが失敗，すなわちミュータントを発見した場合，テストスイートはミュータントを検出しキル（kill）したとみなされる．反対に全てのテストが成功，すなわちミュータントを見逃した場合，そのミュータントは生存（survive）しているとみなされる．Survived ミュータントは元のプログラムとは異なる振る舞いをするにも関わらずテストで検出されなかったミュータントであり，その振る舞いの違いを観測するアサーションや入力の不足を意味する．したがって，開発者は Survived ミュータントを検出可能なテストを追加することでテストスイートの不足を補い，その十分性を高める．

2.2. ミューテーション演算子の不足

ミューテーションテストの課題の一つとして，既存のミューテーションツールに組み込まれているミューテーション演算子は十分であるとは限らないことが挙げられる．この課題に対処するため，言語特性を対象とした新たなミューテーション演算子の提案が継続的に行われてきた [9] [10].

Python 向けのミューテーションツールにおいても，言語特有の動的な振る舞いに起因する欠陥を既存のツールが表現しきれていないという問題が存在する [12]．静的型付け言語ではコンパイラが検出するためテストスイートによる検証の対象とならなかった欠陥も，動的型付け言語ではテストスイートによって検証する必要がある．しかし，Python 向けの代表的なミューテーションツールである mutmut(注2) や Cosmic Ray(注3) は，算術演算子の置換や定数の変更といった手続き的な処理の誤りを想定した汎用的なミューテーション演算子を主に採用している．これらはプログラムの静的な構造の書き換えに留まっているため，動的型付け言語特有の欠陥を表現するミュータントを生成できていない．

Python 特有の欠陥を表現するために，Alimadadi らは静的解析と動的解析を組み合わせた障害駆動のミューテーションツール PyTation を提案している [12]．PyTation には Python 特有の欠陥に合わせた 7 個の新たなミューテーション演算子が組み込まれており，Python の柔軟な引数渡しに起因する欠陥や動的型付けに起因する型変換の省略，オブジェクトの属性やメソッドの誤用など，言語特性に起因する欠陥パターンを対象としている．評価実験の結果，PyTation が生成するミュータントは既存ツールである Cosmic Ray が生成するミュータントとは異なる振る舞いを引き起こし，既存ツールでは表現できない欠陥パターンを扱えることが示されている．

しかし，このような不足しているミューテーション演算子を体系的に発見することは容易ではない．ミューテーションテストは本来，実際の欠陥の疑似的な再現を目的として提案された背景があるため，不足しているミューテーション演算子を探す際にも実際の欠陥や典型的な誤りを分析する手法が広く採用されている [10] [12] [13]．しかし，実際の欠陥を分析する際には複雑な構造の理解や分析対象のプロジェクト固有の知識を要する場合があり，膨大な欠陥データの中からミューテーションテストに適用可能な規則を手作業で抽出・分類するには多大なコストがかかる．したがって，スケーラブルかつ体系的に不足しているミューテーション演算子を発見するためには，欠陥データの分析に依存しない新たな観点からのアプローチが必要となる．

3. Peaceful テストの解析

本研究の目的は，不足しているミューテーション演算子を発見する体系的な方法の獲得である．そのために，本研究ではミュータントの検出に貢献していないテストを Peaceful テストと定義し，その解析により不足したミューテーション演算子を特定する手法を提案する．ミューテーションツールの開発者はこの手法により不足を把握し，特定したミューテーション演算子を実装することでツールを改善できる．一般的にミューテーションテストを活用する際には，生成されたミュータントの集合が検証対象の欠陥を十分に代表しているとみなし，テストスイートの評価や改善が行われてきた．これに対し，提案手法はテストスイートが十分であると仮定してミューテーション演算子の不足を探すという新しい視点からのアプローチである．以降，Peaceful テストの詳細と具体的な解析手法について説明する．

3.1. Peaceful テスト

本研究では，ミュータントの検出に貢献していないテスト，すなわち全てのミュータントに対して成功するテストを

(注2)：https://github.com/boxed/mutmut

(注3)：https://cosmic-ray.readthedocs.io/en/latest/index.html

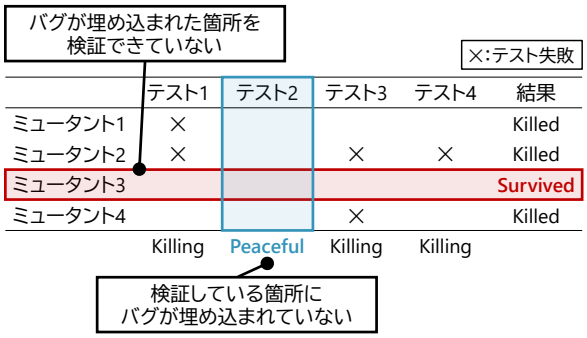

図 1: ミューテーションテストの結果の例

Peaceful テストと定義する．通常，ミューテーションテストの結果の分析では 図 1 の赤枠のようにどのテストにも発見されていないミュータントに着目するが，提案手法では 図 1 の青枠のようにどのミュータントも発見できていないテストに着目する．例えば 図 1 ではテスト 2 が Peaceful テストとなる．反対に，テスト 1 など少なくとも一つのミュータントを発見できているテストを本稿では Killing テストと呼ぶ．

Peaceful テストの存在は，適用したミューテーションツールに組み込まれているミューテーション演算子の不足を意味している場合がある．なぜなら，テストスイートが十分であると仮定すると，既存のミューテーション演算子では Peaceful テストの検証箇所にミュータントを生成できないと解釈できるためである．ただし，テスト対象プログラムの振る舞いを全て検証するテストスイートは現実には存在しないため，この仮定を成立させるためには多数のプロジェクトを用いることが必須となる．これは 図 1 における列（テスト）の母集団の疑似的な拡張に相当する．様々な開発者が作成した無数のテストを用意し，どのようにバグを混入させても成否が変わらないテストが存在するのであれば，バグ混入方法であるミューテーション演算子に不足があるとみなせる．

なお，Peaceful テストはミューテーション演算子の不足だけでなく，テストの内容やツールの仕様によっても生じうる．テストの内容に関する要因としては，アサーションの不足や入力の不備，あるいはライブラリなどミューテーションの対象外となる外部機能の検証によるケースが挙げられる．ツールの仕様に関する要因としては，例えば mutmut では特定の名前の関数やデコレータ付き関数がミューテーションの対象外とされており，特定のコード領域に対してミュータントが生成されない．また，ツールの仕組み上検出できない場合も考えられる．このような要因によって生じる Peaceful テストの中には，ミューテーション演算子の不足を発見するうえでは単なるノイズとなるものが含まれる．例えば，外部ライブラリの振る舞いを確認するテストは検証箇所がミューテーションの対象外であるため，どのようなミューテーション演算子を追加しても Killing テストにはなりえない．また，ツールの仕様により Peaceful となったテストも，その制限を取り除いた場合に Killing テストになるのであれば，既存のミューテーション演算子で検出可能であったことを意味するため，不足したミューテーション演算子の発見にはつながらない．

3.2. 解析手法

本節では，Peaceful テストを収集・分析する具体的な手法について説明する．まず，対象とするミューテーションツールとプログラム，およびそのテストスイートを用意してミューテーションテストを実行する．この際，Peaceful テスト検出のために各ミュータントに対して全てのテストを実行し，その結果を収集する．次に，得られた結果からどのミュータントも発見していないテストを Peaceful テストとして抽出する．

続いて，抽出された Peaceful テストの要因を目視で解析して分類する．3.1 節で述べた内容を整理すると，要因は主に以下のカテゴリに分類される．

テスト側の要因

- アサーションの不足
- 入力の不備
- 外部機能の検証

ミューテーションツール側の要因

- ツールの仕様によりミューテーション適用外
- ツールの仕組み上検出不可能
- ミューテーション演算子の不足

なお，一つのテストが複数の要因に該当するケースも存在する．例えば，例外の型は検証しているがメッセージは検証していない場合，既存のミューテーション演算子では Killing テストにならないが，例外の型を変えるような操作を考えれば Killing テストになりうる．この場合はアサーションの不足とミューテーション演算子の不足の両方に該当する．

分類作業においてミューテーション演算子の不足に該当するテストを発見した際には，どのようなミューテーションが行われれば Killing テストになりうるのかを手動で分析し，新しいミューテーション演算子を設計するための知見を得る．ただし，適切なミューテーション演算子を設計するためには対象のテストを Killing テストにできるだけでは不十分であり，次の 2 点が重要だと考えられる．一つ目は，等価ミュータント [2] を生成しにくいことである．等価ミュータントはミューテーション後も振る舞いが変わらないミュータントを指す．これはいかなるテストでも検出できずテストスイート評価のノイズになるうえに実行コストもかかるため，可能な限り生成しない方が望ましい．二つ目は，生成されるミュータントが容易には検出されないことである．この性質を持つミューテーション演算子を用いれば，より厳格にテストスイートの十分性を評価できる．例えば振る舞いを過度に変える破壊的なミューテーション演算子は，等価ミュータントを生成しにくい一方で容易に検出されてしまうため，必ずしも望ましいとは言えない．

本手法は Peaceful テストを特定し整理する枠組みを提供することで，分析対象を限定できる点に利点がある．本論文では要因分類などを手動で行っているが，提案手法には部分的に自動化の余地がある．この自動化の可能性および具体的なアプローチについては，5 節にて詳細を議論する．

表 1: 実験対象のプロジェクト

No.	プロジェクト名	コミット	LOC	テスト数	命令網羅率
P1	blinker	669f3a0	268	25	94.0%
P2	pyjwt	7144e45	2,005	369	66.1%
P3	structlog	c0ef9e0	3,148	896	95.0%
P4	django-rest-framework	22e231c	9,523	1,552	95.0%
P5	praw	9e74044	7,382	1,029	87.1%
P6	packaging	3b77a26	4,121	27,489	94.1%
P7	python-certifi	8571a4b	44	3	47.4%
P8	typing_extensions	9d1637e	2,473	547	23.6%
P9	requests	111d2b7	3,129	618	87.5%
P10	charset_normalizer	0f07891	4,708	162	92.7%

表 2: ミューテーションテストの実行結果

No.	テスト数	調査したテスト数	Peaceful テスト数	Peaceful 率	Mut 数	MS*
P1	25	25	0	0.0%	193	72.5%
P2	369	365	21	5.8%	1,585	71.7%
P3	896	624	21	3.4%	2,533	57.2%
P4	1,552	1,544	107	6.9%	13,708	100.0%
P5	1,029	514	54	10.5%	5,982	36%
P6	27,489	27,489	35	0.1%	3,516	89.5%
P7	3	3	0	0.0%	16	75.0%
P8	547	522	501	96.0%	171	17.5%
P9	618	594	11	1.9%	3,778	60.2%
P10	162	161	27	16.8%	2,454	38.4%

* 拡張した mutmut が出力する XML から検出された割合を算出

4. 評 価 実 験

提案手法が不足したミューテーション演算子の発見に有用かを評価するために，Python の実プロジェクトを用いて以下の Research Questions（RQ）に回答する．

RQ1 Peaceful テストはどの程度存在するか

RQ2 Peaceful テストが発生する原因は何か

RQ3 ミューテーション演算子の不足を発見できるか

RQ1 では，実プロジェクトにおける Peaceful テストの存在割合を定量的に調査し，提案手法が分析対象とする Peaceful テストが現実にどの程度生じるのかを把握する．RQ2 では，3.2 節で紹介した解析手法を用いて各 Peaceful テストの原因を分類し，どのような原因がどの程度存在しているのかを明らかにする．RQ3 では，提案手法により不足しているミューテーション演算子の発見が可能かを調査する．不足しているミューテーション演算子は一意に定まるものではないため，RQ3 の評価ではまず PyTation [12] のミューテーション演算子に相当する操作を発見できるかを確認する．また，PyTation のミューテーション演算子でも Killing テストにならなかった場合には，さらにどのような操作が必要かを分析し，PyTation のミューテーション演算子以外の新たなミューテーション演算子の発見につながるかを確認する．

本実験で対象とするミューテーションツールは，Python 向けの代表的なツールである mutmut（v3.5.0）である．ただし，3.2 節の解析手法を実現するため，mutmut に変更を加えている．まず，Peaceful テストの判定には全てのミュータントに対して全てのテストを実行する必要がある．しかし，標準の mutmut はコスト低減のためにカバレッジ情報を用いてミューテーション箇所をカバーしているテストのみを実行し，さらにいずれかのテストがミュータントを発見した時点でそのミュータントに対するテスト実行を打ち切る．そのため，本実験ではカバレッジに基づくテスト選択およびテスト実行の打ち切りを無効化した．加えて，目視分類のために各ミュータントを発見できた/発見できていないテストの集合を記録し，XML 形式で出力する機能を追加している．

本実験で対象とする Python プロジェクトを 表 1 に示す．いずれも GitHub から取得しており， 表 1 に記載のコミットを対象としている．本実験では，mutmut を正常に適用可能であり設定ファイルとして pyproject.toml を使用しているプロジェクトを対象とした．P1 から P5 は PyTation の提案論文 [12] において評価に用いられた題材の 13 件のうち条件を満たす 5 件である．他方，P6 から P10 は PyPI の人気ランキング（2026 年 4 月 7 日時点）の上位から順に条件を満たすプロジェクトを 5 件抽出している．

4.1. RQ1：Peaceful テストはどの程度存在するか

4.1.1. 評 価 手 法

Peaceful テスト解析の前提として，Peaceful テストが実プロジェクトにどの程度存在するかを定量的に調査する．具体的には，3.2 節で述べた解析手法のうち Peaceful テストの抽出までを各プロジェクトに対して実施し，検出された Peaceful テストの数やその割合を計測する．

4.1.2. 結　　果

RQ1 の調査結果を 表 2 に示す．まず，mutmut を実行するにあたり，開発者によってスキップされているテストおよび mutmut 適用時に失敗するテストを調査対象から除外した．これは，mutmut が全テストの成功を前提として動作し，失敗するテストが含まれると正常に適用できないためである．除外後に調査できたテストは全体の 9 割以上であり，プロジェクト単位で見てもその多くで 9 割近くを調査できている．なお，失敗するテストを mutmut 実行時に除外したことや，mutmut の技術的な要因によりミュータントに対してテストが実行されなかったことが原因で Peaceful テストになった場合などは，Peaceful テストにはカウントしていない．

ミューテーションテストを実行した結果，10 件中 8 件のプロジェクトで Peaceful テストが抽出されており，Peaceful テストが実プロジェクトに存在することが確認できた．一方でその割合はプロジェクトによって 0.0% から 96.0% まで大きく異なっており，プロジェクトの特性に強く依存する．

提案手法はテストスイートが十分だと仮定すればミューテーション演算子の不足を発見できるという考えに基づいていたが，テストスイートの十分性の代替指標とみなせるミューテーションスコア（MS）が低いプロジェクトにおいても Peaceful テストは多く検出されている．例えば，P5 や P8，P10 は他のプロジェクトと比較してミューテーションスコアが低いにも関わらず，Peaceful テストは高い割合で検出され

表 3: Peaceful テストの分類項目

分類項目	説明
asrt	アサーションが不十分である
data	入力データが与えられていない
ext	外部機能を検証している
deco	デコレータ付きの関数であり対象外
nest	非トップレベルの関数であり対象外
name	関数名により対象外
fstr	f 文字列であり対象外
lim	mutmut の仕組み上検出不可能
mop	ミューテーション演算子が不足している

ている．これは検出された Peaceful テストに 3.1 節で述べたようなテストの内容や mutmut の仕様に起因するもの，すなわち不足しているミューテーション演算子の発見には寄与しないノイズも含まれている可能性があると考えられる．これらの要因の内訳については RQ2 で詳細に分析する．

RQ1 への回答：Peaceful テストは実プロジェクトに存在し，提案手法により 777 件を抽出できた．また，その割合はプロジェクトによって 0.0% から 96.0% まで大きく異なる．

4.2. RQ2：Peaceful テストが発生する原因は何か

4.2.1. 評 価 手 法

RQ1 で抽出した Peaceful テスト 777 件を対象に，著者 1 名により 3.2 節で述べた目視分類を行う．分類の基準は，その要因が解消された場合にそのテストが Killing テストになりうるか否かに置いている．また，本分類は 3.2 節で述べたように，複数の要因に該当する場合がある．そのため，分類後の各項目の合計が Peaceful テスト数を超える可能性がある．

4.2.2. 結　　果

RQ2 の分類項目を 表 3 に示す．分類項目はテスト側と mutmut 側の 2 種類に大別でき，asrt と data，ext がテスト側の要因，その他が mutmut 側の要因である．deco と nest，name，fstr に分類される Peaceful テストは，mutmut の仕様上ミューテーションの対象外となった箇所を検証しているテストである．ここで，非トップレベルの関数とは if 節の内部などで定義された関数を指し，関数名による除外とは `len` などミューテーション対象外となる特定の名前を持つ場合を指す．lim は mutmut の仕組み上，ミューテーションが反映されず検出不可能となる場合である．mutmut では先に import を行った後にミュータントごとに子プロセスを fork するため，関数外の文や自作デコレータのような import 時にしか評価されない処理に対するテストは Peaceful テストになってしまう．mop はミューテーション演算子の不足が要因である場合であり，本研究において最も重要なケースである．

表 3 の分類に基づいて RQ1 で抽出した Peaceful テスト 777 件を分類した結果を 表 4 に示す．Peaceful テストの要因の分布は，対象プロジェクトの特性によって大きく異なる．例えば P8 は，外部機能を多く利用しているという特徴に加え，実装を単一のファイルに記述しており関数外の文が多いという特徴を持つ．そのため，ext や lim が多くを占めている．また，mop に分類される Peaceful テストも確かに検出されており，提案手法が意図した事例を抽出できることが示された．

表 4: Peaceful テストの分類結果

No.	Peaceful	asrt	data	ext	deco	nest	name	fstr	lim	mop
P2	21	4	0	0	10	10	3	0	0	9
P3	21	2	0	0	6	0	2	1	3	7
P4	107	9	55	10	19	1	1	0	3	18
P5	54	0	0	4	26	0	0	2	20	14
P6	35	3	0	0	12	0	2	0	9	12
P8	501	31	0	216	0	20	8	0	217	145
P9	11	3	0	3	5	0	0	0	1	2
P10	27	0	0	0	27	0	0	0	0	0

表 5: 不足しているミューテーション演算子

No.	mop	RFA	RCF	RElC	RExC	CUA	RAA	RMC	その他
P2	9	0	0	0	0	4	4	0	13
P3	7	0	0	1	0	1	1	0	12
P4	18	0	0	0	0	0	1	2	24
P5	14	0	0	0	0	0	0	0	24
P6	12	0	0	5	0	0	0	0	10
P8	145	0	0	1	0	112	18	1	28
P9	2	0	0	0	0	2	1	2	0

RFA: Remove Function Argument　　RCF: Remove Conversion Function
RElC: Remove Element from Container　　RExC: Remove Expression from Condition
CUA: Change Used Attribute　　RAA: Remove Attribute Access
RMC: Remove Method Call

RQ2 への回答：ミューテーション演算子の不足が要因となっている Peaceful テストは 207 件検出されており，提案手法は意図した事例を抽出可能であると確認できた．

4.3. RQ3：ミューテーション演算子の不足を発見できるか

4.3.1. 評 価 手 法

RQ2 で mop に分類された Peaceful テスト 207 件を対象に，不足しているミューテーション演算子の発見が可能かを著者 1 名によって調査する．初めに，PyTation のミューテーション演算子に相当する操作を手動で適用した場合にテストが失敗するか，すなわちミュータントを検出できるかを確認する．検出できた場合，その操作に相当するミューテーション演算子が不足していたことを発見できたとみなす．PyTation のミューテーション演算子に相当する操作を適用しても Peaceful テストのままであった場合は，それ以外に Killing テストになる操作がないかを手動で探す．なお，一つのテストが複数の操作によって Killing テストになりうるため，分類後の各項目の合計は mop の数と一致するとは限らない．

4.3.2. 結　　果

RQ3 の調査結果を 表 5 に示す．調査の結果，提案手法によって PyTation のミューテーション演算子を発見できることが確認された．これらは PyTation の評価に用いられていない P6 と P8，P9 からも発見されている．発見されたミューテーション演算子に着目すると，オブジェクト指向な CUA（属性の変更）と RAA（属性の削除）が見つかりやすい傾向にあった．これらは他のミューテーション演算子と比べてプログラ

表 6: 新たに発見できたミューテーション演算子

変更操作	件数
条件式に not を挿入	29
例外を組み込みの例外の型に置換	17
基底クラスを削除	14
戻り値を None に変更	14
raise される例外を Exception に変更	14
基底クラスを基底クラスの祖先クラスに置換	10
戻り値を同じ型の新しく生成したインスタンスに変更	5
戻り値 None を""に変更	3
比較演算子を反転	1
定数から 1 を減算	1
基底クラスを同じモジュールの別の属性に置換	1
yield を return に置換	1
代入文の右辺を型注釈から推論したデフォルト値に置換	1

ムの振る舞いをより大きく変化させる．そのため，対象箇所へ適用した際にテストが失敗しやすく，発見されやすい傾向にあったのではないかと考えられる．一方，RFA（関数のオプション引数の削除）と RCF（型変換関数の削除），RExC（複合条件式から条件式を一つ削除）は本実験では発見されなかった．RFA と RExC については，mutmut にも引数の削除や条件式に関する操作が組み込まれていることが要因であると考えられる．RCF については，そもそも適用可能な箇所が少なかったため，発見されなかったのだと考える．

さらに，PyTation のミューテーション演算子以外の不足も発見された．新たに発見されたミューテーション演算子を 表 6 に示す．まず，戻り値を None や""に変更する操作と比較演算子を反転させる操作は，Java 向けの Pitest [4] も類似したミューテーション演算子を備えており，既存ツールにも前例のある妥当な候補といえる．また，条件式への not の挿入や定数から 1 を引く操作，yield から return への置換は mutmut のミューテーション演算子にも類似した操作があり，こちらも候補として妥当であると考えられる．一方，型の削除や置換に関する操作は Python 特有のミューテーション演算子であると考えられる．Python は型に厳格でないため，型に関する欠陥が組み込まれたミュータントも実行可能である．これに対して，開発者は Python においても型を意識しており，今回抽出した Peaceful テストの中にも型を検証しているテストが多くみられた．したがって，型の削除や置換に関する操作も有効なミューテーション演算子だと考えられる．

RQ3 への回答：提案手法により PyTation のミューテーション演算子 4 個と新たな不足ミューテーション演算子 13 個を発見でき，提案手法の有用性を確認できた．

5. おわりに

本稿では，ミューテーション演算子の不足を体系的に発見する手法の獲得を目的として，Peaceful テストとその解析手法を提案した．評価実験の結果，Peaceful テストは実プロジェクトに存在し，その発生割合や要因の内訳はプロジェクトの特性によって大きく異なることを確認した．また，ミューテーション演算子の不足に起因する Peaceful テストは 207 件発見でき，不足しているミューテーション演算子の候補も 17 個発見した．これらの結果は，提案手法がミューテーション演算子の不足の発見に有用であることを示している．

今後の課題として，4.3.2 節で発見されたミューテーション演算子以外の操作を発見するためには，より大規模に解析を行う必要がある．そのために，提案手法の部分的な自動化や分析するテスト数の削減を行う必要がある．例えば，要因分類には自動化の余地があり，ツール仕様に起因する Peaceful テストはツールの制約を一時的に外して再実行し Killing テストになるかを見ることで機械的に分類できると考えられる．また分析対象の削減については，Peaceful テストを抽象構文木の構造などで分類し類似事例をまとめることで確認すべきテスト数自体を削減する方法や，LLM の活用が考えられる．

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.R. Woodward, “Mutation testing—its origin and evolution,” J. Inf. Softw. Technol., vol.35, no.3, pp.163–169, 1993.

[2] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” Trans. Softw. Eng. (TSE), vol.37, no.5, pp.649–678, 2010.

[3] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” In Proc. Int. Symp. Found. Soft. Eng. (FSE), pp.654–665, 2014.

[4] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “PIT: a practical mutation testing tool for Java,” In Proc. Int. Symp. Soft. Test. Anal. (ISSTA), pp.449–452, 2016.

[5] Y.-S. Ma, J. Offutt, and Y.R. Kwon, “MuJava: an automated class mutation system,” Softw. Test. Verif. Reliab., vol.15, no.2, pp.97–133, 2005.

[6] R. Just, “The major mutation framework: efficient and scalable mutation analysis for Java,” In Proc. Int. Symp. Soft. Test. Anal. (ISSTA), pp.433–436, 2014.

[7] Y. Jia and M. Harman, “MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language,” In Proc. Test. Acad. Ind. Conf. Pract. Res. Tech. (TAIC PART), pp.94–98, 2008.

[8] A. Denisov and S. Pankevich, “Mull it over: mutation testing based on LLVM,” In Proc. Int. Conf. Softw. Test. Verif. Valid. Work. (ICSTW), pp.25–31, 2018.

[9] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, “Inter-class mutation operators for Java,” In Proc. Int. Symp. Softw. Reliab. Eng. (ISSRE), pp.352–363, 2002.

[10] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Efficient JavaScript mutation testing,” In Proc. Int. Conf. Softw. Test. Verif. Valid. (ICST), pp.74–83, 2013.

[11] F. Tip, J. Bell, and M. Schäfer, “LLMorpheus: mutation testing using large language models,” Trans. Softw. Eng. (TSE), pp.1645–1665, 2025.

[12] S. Alimadadi and G. Gharachorlu, “Hybrid fault-driven mutation testing for Python,” In Proc. Int. Conf. Softw. Eng. (ICSE), 2026. To appear. arXiv:2601.19088.

[13] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Learning how to mutate source code from bug-fixes,” In Proc. Int. Conf. Softw. Maint. Evol. (ICSME), pp.301–312, 2019.

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, “Mutation testing advances: an analysis and survey,” Adv. Comput., vol.112, pp.275–378, Elsevier, 2019.