情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2009-SE-166 No.18
2009/11/6

派生開発におけるミューテーションテスト手法のシステムテストへの応用

丸地康平† 今井健男† 太田暁率† 片岡欣夫†

派生開発では，既存のテスト資産を活用しテストケースを作成する．ゆえに，テスト工程の質を保証するには，テスト資産のテストケースの不足や冗長を見つけるテストケース評価技術が重要となる．ミューテーションテストはテスト評価技術であり，テスト削減に有効であることが知られる．しかし，時間コストを要する技術であり，システムテストへの適用は困難である．
本稿では，派生開発のシステムテストに対し，ミューテーションテストの適用方法を提案する．適用実験により，提案方法がミューテーションテスト自体に要する時間の削減や，テストケース不足の発見に有用であることを示す．

Application of mutation testing to system test for derivative development

Kohei Maruchi†, Takeo Imai†, Akinori Ohta† and Yoshio Kataoka†

In derivative development, test assets of previous versions are reused. Therefore, test case evaluation techniques that find the lack or redundancy of test cases in the test assets are required for quality assurance of test process. Mutation based testing is one of the test case evaluation technique and is known effective as test reduction. But mutation based testing takes too much time to be applied to system test.
This paper proposes a method for applying mutation based testing for system test for derivative development. Our experiment showed this method was effective for reducing time required to mutation based testing itself and finding lack of test cases.

1. はじめに

既存のソフトウエアに対し機能追加や一部機能の変更を行う派生開発では，既存のソースコードを検証したテストケース（テスト資産）が存在する．ゆえに，テストケース作成工数を抑えたるため，テスト資産を流用したテスト実施を行うことが多い．しかし，テスト資産の不足や重複がある場合，以降の開発において，検証すべき事項の漏れや，テスト工数増加の原因となる．そのため，テスト資産自体の質が要求され，テストケースの不足や重複を発見するテストケース評価技術が重要となる．特に派生開発では，テスト資産が肥大化しやすく，適切なテストケース評価技術を用いて，効果的なテスト削減（test-suite reduction または，test-suite minimization）[1]を行うことが重要となる．

ミューテーションテスト[2]はテスト評価技術に有用な技術の一つである．特にミューテーションテストをベースにしたテスト削減は，テストセットの不具合検出効率（fault detection effectiveness）の低下を抑えつつ，テストケース数を削減する技術である[3]．しかし，ミューテーションテストはテストケースの評価に時間を要する技術であり，単体レベルのテストケースへの実用化は進んでいるが，システムレベルのテストへは，我々の知る限り応用されていない．

本稿では，派生開発におけるシステムテストを評価するため，ミューテーションテストを適用する方法を提案する．本提案方法では，システムテストから派生開発で追加・変更した箇所への単体テストを抽出し，この単体テストに対しミューテーションテストを適用し，その評価結果をシステムテストの評価に用いる．本提案方法は，ミューテーションテスト適用箇所を，派生開発における追加，変更箇所に限定するため，時間コストを抑えることが期待できる．そこで，提案方法の適用実験を行い，時間コストが削減されているか，システムテストへの有用な評価結果を得られるか分析を行う．

本稿の構成は，2 節でミューテーションテストの説明を行う．3 節でシステムテストへの適用方法を提案する．4 節で，提案方法の適用実験を行う．その結果の考察を，評価時間を要さずに実施できるか（時間の評価），実験結果を用いシステムテストへの評価にどれだけ役に立つのか（有用性の評価）の観点から行う．

2. ミューテーションテストについて

本節では，ミューテーションテストの説明を行う．2.1 節でミューテーションテストの概要とテストケース評価への活用方法を，2.2 節でミューテーションテストの実施手順を紹介する。

2.1 ミューテーションテスト概要

ミューテーションテストとは，機械的に不具合をソースコードに埋め込み，検出で

† （株）東芝 研究開発センター システム技術ラボラトリー
System Engineering Laboratory, Corporate Research & Development Center, Toshiba Corp.

ⓒ2009 Information Processing Society of Japan

きる不具合の数でテストケースを評価するテストケース評価技術である．図 1は，ソースコードに不具合を 10 個埋め込んだ様子を表した模式図である. 各テストケースが丸で囲った不具合を検出する時, 下記のようにテストケースを評価することができる.

A) Test case 1 は，全体の 4/10 の不具合を検出する
B) Test case 1 と 2 と 3 は，合わせて 9/10 の不具合を検出する
C) Test case 3 は，テストケース 2 に包含されている

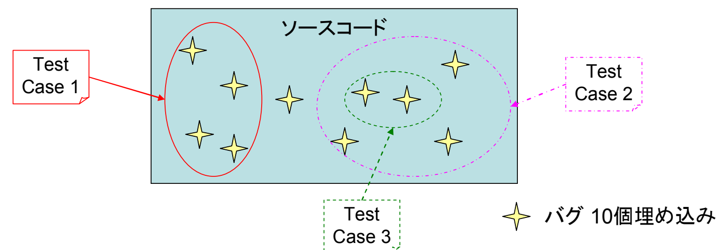

図 1 ミューテーションテスト概要図

A)の評価方法は，各テストケースを埋め込んだ不具合をどれだけ発見するかで個別に評価する. B)の評価方法は, テストケース全体でいくつの不具合を検出したか調べ，どのテストケースでも検出できなかった不具合の存在を明らかにする．検出できなかった不具合があれば，その不具合を検出するテストケースが不足していると考えることができる．検出できない不具合を見つけるテストケースを追加し，テストケースの質を向上することが可能である．C)の評価方法は，検出する不具合集合の包含関係に着目する．包含されているテストケース（Test case3）よりも包含しているテストケース（Test case2）の優先度が高いとランク付けを行うことや，包含されているテストケース（Test case3）は重複していると評価することができる．

2.2 ミューテーションテストの詳細説明

ミューテーションテストは，ソースコードを改変してミュータント（後述）を作成し，ミュータントを用いてテストケースを評価する．そして，その結果としてミューテーションスコアや区別できなかったミュータント（区別したミュータント）を得る（図 2)．

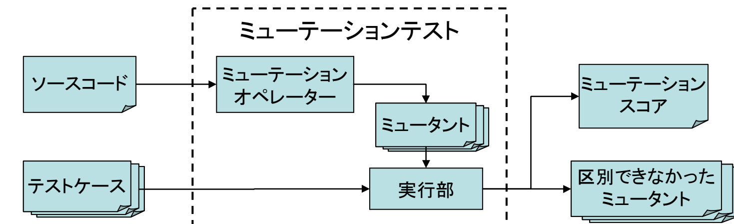

図 2 ミューテーションテスト構成図

ソースコードは，ミューテーションオペレータにより改変され，不具合を埋め込まれる．不具合が埋め込まれたソースコードはミュータントと呼ばれる．1 ミュータント辺り，1 つの不具合が埋め込まれる．そのため，ミューテーションオペレータが埋め込む不具合の数だけ，ミュータントが作成される．表 1 は Java 向けのミューテーションツール Jumble [4]が用いるオペレータである．例として，Conditionals オペレータの説明をする．Conditionals オペレータは，ソースコード中の条件文を否定に改変する．例えば，ソースコードに「if(x>y)」のような条件文があれば，その条件文のみを「if(!(x>y))」と変えたコードをミュータントとして作成する．このように，Conditionals オペレータは，ソースコード内の条件文の数だけ，ミュータントの作成を行う．このように，ミューテーションオペレータが作成する不具合はソースコードの記述を変更する程度であるため，実際に現場で混入する不具合とは剥離がある．この点は，ミューテーションテストの大きな課題であるが，ミューテーションオペレータの信頼性を確認した実験結果が知られている[5]．

表 1 ミューテーションツール Jumble で用いるオペレータ

Jumble's Operators	改変方法
Conditionals	条件文を否定に変える。x>y → !(x>y)
Binary Arithmetic Operation	2項演算子を変える。+ ⇔ -、& ⇔ | etc
Increments	++ ⇔ -- 、 += ⇔ -=
Inline Constants	インライン文字定数を変える。
Class Pool Constants	クラスプールの文字定数を"_jumble_"に変える
Return Values	Non zero → 0 、 0 → 1
Switch Statements	caseの値を変えて、case文を入れ替える

実行部（図 2）は，このように生成したミュータントを，入力のテストケースで動作させ，テストが成功するか失敗するかを調査する．テストの成否は，テストケース

ⓒ2009 Information Processing Society of Japan

が持つテスト実行後の期待値を用い，ミュータントを動作させた結果と期待値が等しいか調べることで行う．テスト成功であれば，不具合の混入を発見していないため，テストケースはミュータントを検出できなかったことになる．逆に，テスト失敗であれば，不具合の混入を検出したことになる．

ミューテーションオペレータは，1 ミュータントに 1 不具合を入れるため，期待値の違いを見るだけで,テストケースがどの不具合を検出したか把握することができる．もし, 1 ミュータントに複数の不具合を入れると, 不具合を検出したことは分かるが, どの不具合を検出したか期待値の違いだけでは判別できない．

ミューテーションテストは，テストケースとミュータント全ての組み合わせで実行させるため，評価時間を要する．ソースコードの規模が大きくなるほど，ミュータント数は増え，1 回当たりの試行時間も増える．そのため，単体レベルのテストへの実用化は進んでいるが，規模の大きいシステムテストへの実用化はされていない現状である．

以上がミューテーションテストの実施手順である．結果として，ミューテーションスコアと区別できなかった（区別できた）ミュータントを得る．

ミューテーションスコアは，ミューテーションテストの評価結果であり，下記のように算出する．

（テストケースが検出した不具合の数）／（全ミュータントの数）

図 1 の各テストケース 1，2，3 はそれぞれ，2/5(4/10)，1/2(5/10)，1/5(2/10)のミューテーションスコアとなる．

区別できなかったミュータントは，ミューテーションテストの評価結果であり，テストケースにより検出されなかった不具合を含むミュータントである. これらを用い，どの不具合を検出するテストケースが不足しているか知ることができる．同様に区別できたミュータントも，ミューテーションテストの評価結果である．

3. システムテスト評価方法の提案

本節では，検証対象を限定し，ミューテーションテストに要する時間コストを削減することを特徴としたシステムテスト評価方法を提案する．派生開発のテスト工程において，新規・変更箇所を検証対象として限定することができるため，本方法は派生開発に適した評価方法である．図 3 が提案方法の概要を表した構成図であり， 図 4 の手順で行う．

まず，派生開発のためのソースコード作成，修正を行う（図 4 の step1）．この際，新規追加や修正した関数・メソッドを検証対象モジュール（図 3）と呼ぶ．次に，検証対象モジュールに対する入力ログと出力ログを採取するロガーを作成する（図 4 の step2）．ロガーは，入力ログとして，関数であれば関数名，引数，出力ログとして，その戻り値を記憶する．入力ログと出力ログはセットで管理し，対応するテストケース（システム）とも関連させて管理する．ここで，テスト実施中のログを採取しながら，システムテストを行う（図 4 の step3）．このとき，不具合が発見された場合は，ソースの修正からやり直す．不具合が発見されなかったら，システムテストは終了となり，次のステップからテスト評価作業となる．

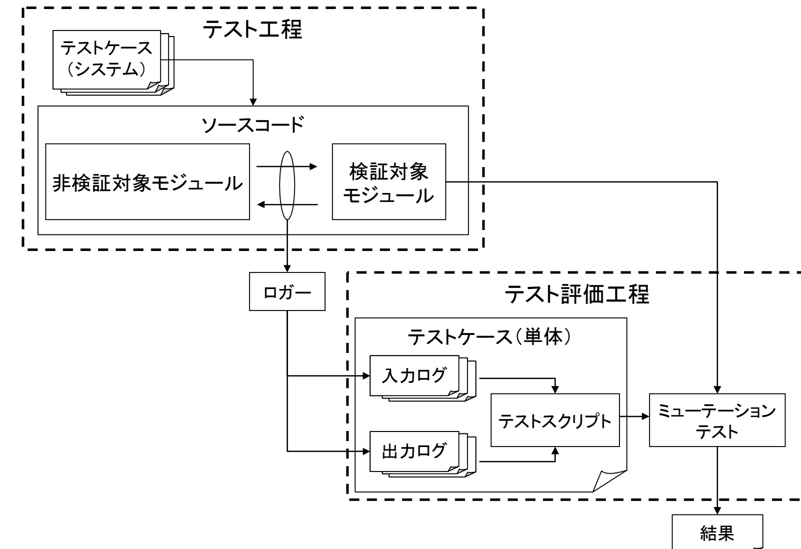

図 3　提案方法構成図

Step1:ソースの修正・追加をする
Step2 :ロガーを作成する
Step3 :システムテストを実施する
– 不具合がある場合は，*Step1*へ戻る
Step4 :ログを用いて，検証対象モジュールの単体テストを作る
Step5 :単体テスト、検証対象モジュールを入力とし，ミューテーションテストを行う

図 4　提案方法手順

テスト評価作業では，まず，システムテスト実施中に採取したログより，単体テストの作成を行う（図 4 の step4）．単体テストは，入力ログ，出力ログ，テストスクリ

ⓒ2009 Information Processing Society of Japan

Vol.2009-SE-166 No.18
2009/11/6

プトで構成する．テストスクリプトは，図 5 の手順で動作する．まず，入力ログとそれに対応する出力ログを読みこむ（図 5 の step1）．次に，入力ログより，システムテスト実施中に行われた関数呼び出しの再現を行い（図 5 の step2），再現した戻り値と対応する出力ログが等しいかを確認する（図 5 の step3）．再現した関数であるため，再現結果と出力ログは等しくなることが期待される．等しければテスト成功，そうでなければテスト失敗と判断する．このテストスクリプトは再現した関数が同一であれば常にテスト成功となる．そのため，ミューテーションテストでテスト失敗になる場合は，ミュータントが発見されたことを意味する．

最後に，作成したテストケース（単体）と検証対象モジュールを入力として，ミューテーションテストを実施する（図 4 の step5）．テストケース（単体）では，システムテスト実施中の検証対象関数の動作を再現している．そのため，ここで行うミューテーションテストはシステムテストがミュータントを発見できるか調べていると見なすことができる．ゆえに，ミューテーションテストの結果は，対応するテストケース（システム）の評価に用いることができると考えられる．

Step1:ログを読み込む
Step2 :入力ログより，関数を再現する
Step3 :関数の再現結果の戻り値と対応する出力ログを比較する
- 比較が一致する場合は，*Step4*へ
- 比較が一致しない場合は，*Step5*へ

Step4 :テスト成功と判断し，テストを終了する
Step5 :テスト失敗と判断し，テストを終了する

図 5 テストスクリプトの動作手順

本方法の特長は，ミューテーションテストによるテスト評価自体を検証対象モジュールのみに特化することにより，ソースコードもテストケースも単体テストレベルの規模でミューテーションテストを実施できることである．そのため，ミューテーションテストの課題である評価時間を削減することが期待できる．

4. 提案方法の適用実験

本節では，3 節で提案したシステムテスト評価方法の適用実験を行い，提案方法の評価を行う．4.1 節で実験の内容を説明し，4.2 節で提案方法による評価結果を紹介する．4.3 節で実験結果の分析調査を行い，4.4 節で結果を整理し，考察を行う．

4.1 提案方法の適用実験

(1) 検証対象モジュール

今回の実験では，検証対象モジュールとして，我々が Java で開発した Eclipse プラグインのソフトウエアを用いる．Eclipse [6]は，IBM によって開発されたソフトウエアの統合開発環境であり，プラグインとして様々な機能を組み込みできるように設計されている．

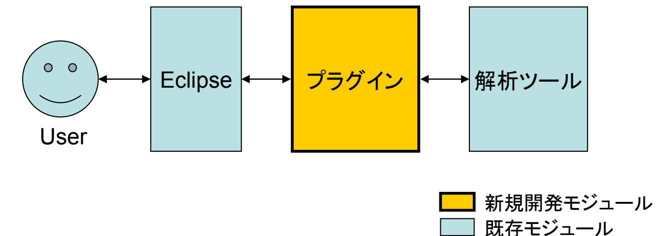

図 6 実験モチーフ概要図

図 6 がモチーフとするシステムの概要図である．プラグインが今回検証のターゲットとする箇所である．プラグインは Eclipse が受理したマウス操作，キーボード入力といったユーザ操作を解釈し，解析ツールに必要な指令を出す．解析ツールは，ユーザの入力に対しある解析を行う．その結果は，プラグインを通して，ユーザに表示される．プラグインや解析ツールは，本研究とは独立して，我々が開発したソフトウエアである．

今回の実験では，プラグインの解析依頼を行うクラス（以降，解析依頼クラスと呼ぶ）を検証対象モジュールとする．解析依頼クラスは，3 つのメソッドを持ち，130 行程度のクラスである．

(2) テストケース（システム）

今回の適用実験で用いるテストケースは，モチーフであるプラグインを検証するために作成したシステムテストを元に作成する．元となるシステムテストは，プラグインの仕様を洗い出し，それらを網羅的に動作確認する．テスト実施時の操作はマウス操作，キーボード入力といった人手により行い，動作確認も目視により人手で行う．

適用実験で用いるテストケースは，元のシステムテストから解析依頼クラスに関わるテストケースのみを抽出し作成する．解析依頼クラスに関わるのは，解析依頼の入力組み合わせと解析依頼の操作指令である．解析依頼のテスト入力として，元のシス

ⓒ2009 Information Processing Society of Japan

テムテストで直交表を用い絞った全 14 の組み合わせをそのまま流用する．解析依頼の操作には，元のシステムテストでは，計 4 種類の操作が存在するが，その内 1 つは最後に行った解析を再度実行する Repeat 操作である．本操作は他の 3 種類の操作を行うため，Repeat 操作のテスト実施を省いた（この判断は誤っていたことが，本実験結果より明らかになる）．

以上より，今回用いたテストケース数は，14 種類の入力組み合わせに対し，3 種類の操作を行うため，計 42（=14×3）個となる．

(3) ミューテーションツール

今回の実験で行うミューテーションテストは，ミューテーションツール Jumble [4] を用いて行った．Jumble は Java 用のユーテーションツールであり，Eclipse のプラグインも存在する．ソースコードの改変はクラス単位で行い，入力するテストケースは JUnit [7]の形式で記述する必要がある．表 1 が Jumble で行うミューテーションオペレータである．Jumble を解析依頼クラスに適用した際，表 2 の計 12 個のミュータントを得た．

表 2　Jumble が作成したミュータント

ID	行数	改変内容	改変詳細
1	41	文字列を変える	"spec$" -> "__jumble__"
2	43		"" -> "__jumble__"
3	74		"cannot find a function: " -> "__jumble__"
4	81		"cannot find the source file corresponding to " -> "__jumble__"
5	30	返り値を変える	changed return value (areturn)
6	58	条件式を変える	negated conditional
7	63		negated conditional
8	67		negated conditional
9	73		negated conditional
10	71		negated conditional
11	88	返り値を変える	changed return value (areturn)
12	103		changed return value (areturn)

(4) ロガー

ロガーは，今回人手でコード内に記述する．各メソッドの関数呼び出し直後に入力ログを採取するロガー記述を挿入し，return 文の直前に出力ログを採取するロガー記述を挿入した．解析依頼クラスは static メソッドしか存在しなかったため，入力ログとしては，メソッドの引数とメソッド名のログで，メソッドの再現が可能であった．どのメソッドの戻り値も，解析ツールの解析結果クラスのインスタンス（正確には解析結果クラスのインスタンスのリスト）である．解析結果クラスのフィールドには解析に関連する情報が多く含まれるため，出力ログとして，フィールドの値全てを採取するのではなく，検証対象である解析依頼クラスに関連するフィールドに限定しログを採取した．また，解析依頼クラスには例外を投げるメソッドもあるため，例外を throw する直前に出力ログを採取するロガーを記述し，どの例外を投げたか分かるようにした．

(5) テストケース（単体）

Jumble を用いるためには，入力のテストケースを，JUnit テストの形式で記述する必要がある．ゆえに，JUnit テストの形式で図 5 の手順を記述した．図 7 が実際の記述を説明する擬似コードである．(2)，(3)行目で，入力ログ，出力ログを読み出す．入力ログは，メソッドの再現に用い，出力ログは結果の比較に用いる．図 7 は簡易例であるため，入出力にそれぞれ一つの値しかないが，必要に応じて複数の値を読み出す．例外が発生する場合は，出力ログとして，例外発生と種類が識別できるようにする．

入力ログを用い，図 7 の(5)行目のように，メソッドの動作を再現する．複数のメソッドのログを同時に採取している場合は，入力ログにメソッドの識別子を設け，その値により条件分岐し，再現すべきメソッド実行を選択できるようにする．

メソッドの再現が終わったら，再現結果と出力ログの結果が等しいか調べる．図 7 の(6)行目の assertEquals(A,B)メソッドがそれを行う．assertEquals(A,B)メソッドは，JUnit のライブラリにあるメソッドであり，結果が期待値と等しいか判断するメソッドである．すなわち，引数 A，B の値が等しければテスト成功，そうでなければ，テスト失敗と判断する．比較すべき値が複数ある場合は，(6)行目と同様の判断が複数回行う．解析依頼クラスの場合，戻り値となるオブジェクトの一部フィールドの値が比較対象となるため，複数回比較を行った．

今回対象とする解析依頼クラスのように，対象とするメソッドが例外を発生する場合は，図 7 の(4)行目や(7)-(10)行目のように，メソッド再現中に発生した例外を拾う機構が必要となる．(8)，(10)行目では，出力ログと再現結果で同じ例外が発生するか確認する．同じであれば，テスト成功，そうでなければテスト失敗と判断する．この際，出力ログでは例外が発生しており，再現した際にはミュータントの影響で，例外が発生しないケースに注意する．その場合は，図 7 の(8)，(10)行目には到達せずに，(6)行目に到達する．このようなケースは，テスト時には例外が発生し，再現時には例外が発生していないため，異なる振る舞いでありテスト失敗であると判断する必要がある．そのため，図 7 の(6)行目でその判断するようにした．

ⓒ2009 Information Processing Society of Japan

```
public static void  testExample() {               (1)

        /*入力ログを読み出す*/
        input=○ ○ ○ ○;                            (2)

        /*出力ログを読み出す*/
        output=× × × × ;                          (3)

        try{                                      (4)
                /* メソッドを再現 */
                result=MimicMethod (input);       (5)

                /* 結果の比較 */
                assertEquals( result ,output);    (6)
        }
        catch( △△){                               (7)
                /* 結果の比較 */
                assertEquals(△△ ,output);         (8)
        }
        catch( ▲▲){                               (9)
                /* 結果の比較 */
                assertEquals(▲▲ ,output);         (10)
        }

}
```

図 7　JUnit テストケース説明用サンプル

4.2　実験結果

4.1 節の内容でテストケースを評価した結果を，操作ごとにまとめたのが表 3 である．テスト入力の全 14 パターンを各操作で行った結果である．

表 3　操作ごとの結果

行った操作	ミューテーションスコア	区別したミュータントID
操作A	58% (7/12)	1,2,4,6,7,9,11
操作B	58% (7/12)	1,2,4,6,7,9,11
操作C	58% (7/12)	1,2,4,6,7,9,11

表 3 のように, 全ての操作において, ミューテーションスコアが 58%と等しくなり，区別したミュータントも全く同じであった．解析依頼クラスのメソッドは，解析依頼操作 A，B，C で同じように呼び出される設計であるため，3 つの操作の結果が全く同じになったことは予想通りの結果である．

また，ミューテーションテストで課題となる評価時間も数秒で終わり，検証対象を絞り時間コストが削減できたことも確認した．

4.3　実験結果の分析

本小節では，4.2 節の実験結果を分析し，システムテストへ有用なフィードバックを得られるか調べる．

表 3 の結果より，区別できなかったミュータントは，ミュータント ID 3，5，8，10，12 の計 5 つのミュータントであることが分かる．これらミュータントが何故区別できなかったのか分析を行い，区別できるテストケースの補足を行えるか調査を行った．

分析の結果，区別できなかったミュータントの内，2 つ（ID 5，12）は Eclipse からの操作では通らないコードが改変されていたためであることが判明した．区別できなかったミュータントの分析により, 使われないコードの発見に役立つことを確認した．

一方，残る 3 つのミュータントが区別できなかった原因は以下であった．

・テスト入力に不足がある（原因 1）
直交表で作成した 14 の入力組み合わせでは発見できないミュータントであった．

・テスト操作に不足がある（原因 2）
テスト操作から外した repeat 操作時でないと, 発見できないミュータントであった．

・テスト判断で見逃している（原因 3）
作成した JUnit テストケースでの，テスト成否判断に抜けが有り，発見できないミュータントであった．

テスト成否判断の抜けは, 図 8 のような JUnit テストケースを書いたために生じた．図 8 は JUnit テストケースの一部であり, 図 7 の(5)行目に相当するメソッドを再現する箇所と，(6)行目に相当する結果を比較する箇所である．メソッド再現による結果はリストで返るため，その要素を一つずつ正しいか確認を行う．ここで，元のコード実行時には result リストの要素に結果が登録されるが，ミュータント実行時には result リストが空の場合を考える．この場合，異なる結果であるため，テスト失敗と判断されるべきである．しかし，図 8 の JUnit テストケースでは，図 8 (B)行目の for 文に入らないため，assertEquals メソッドによる比較が行われず，結果テスト成功と判断されていた．以上のように，意図的ではなく偶然のミスであるが，JUnit テストケースにお

ⓒ2009 Information Processing Society of Japan

いて result リストが空の場合のテストが抜けていたために、区別できないミュータントが発生した。

```
・
・
List<Result> result;                              (A)
/* メソッドを再現 */
result=MimicMethod (input);                       (5)


for( Result  re: result){                         (B)
        /* 結果の比較 */
        assertEquals( result ,output);            (6)
}

・
・
```

図 8　JUnit テストケースの抜けの例

表 4　テスト入力とテスト操作の組み合わせ結果

Test ID	操作A		Repeat	
	区別した数	区別したmutant ID	区別した数	区別したmutant ID
1	7	1,2,6,7,9,10,11	7	1,2,6,8,9,10,11
2	2	1,4	2	1,4
3	2	1,4	2	1,4
4	2	1,4	2	1,4
5	1	2	1	2
6	2	1,4	2	1,4
7	2	2,6	2	2,6
8	2	1,4	2	1,4
9	2	1,2	2	1,2
10	2	1,4	2	1,4
11	2	1,2	2	1,2
12	2	1,4	2	1,4
13	2	1,4	2	1,4
14	2	1,2	2	1,2
15	7	1,2,3,6,7,9,10	7	1,2,3,6,7,9,10

注）mutants ID 5,12は区別できないもの

以上の結果を踏まえテストケースの補足を行う．（原因 1）より，テスト入力を増やし，テスト入力の組み合わせを全部で 15 パターンとする．Test ID 15 が追加したテスト入力である．また，テスト操作として（原因 2）より，repeat 操作を追加する．（原因 3）より，JUnit テストケースを result リストが空の場合も正しく判断できるように修正する．

再度実験を行い，テスト入力と操作の組み合わせで，区別したミュータント数とその ID をまとめたのが，表 4 である．

表 4 より，テスト入力 ID 15 を加えたことにより，ID3 のミュータントを新たに区別できるようになった．Repeat 操作を加えたことにより，ID8 のミュータントを新たに区別できるようになった．以上により，検出できなかった ID3，8 のミュータントを新たに検出できるようになり，区別できなかったミュータントの解析に人手が必要になるが，テストケース補足への有用性を確認した．

また，表 4 より，JUnit テストケースを修正したことにより，ID10 のミュータントを新たに区別できるようになったことが確認できる．しかし，JUnit テストケースはテスト評価用に作成されたものであるため，システムテストの補足に役立っているわけではない．本結果より，提案方法によるミューテーションテストで直接評価するのは、評価用に作成するテストスクリプトであり，テストケース（システム）を直接評価しているわけでは無いことを確認できる．

4.4 結果の整理，考察

本小節では，3 節で提案したシステムテスト評価方法を，4.2 節，4.3 節より得られた実験結果より整理し，考察する．評価時間を要さずに実施できるか（時間の評価），評価結果を用いどれだけ効果があるのか（有用性の評価）の二つの視点より行う．

4.4.1 時間に関する実験結果，考察

ミューテーション評価に要する時間は，非常に短時間で実施できることを確認した（4.2 節）．しかし，テストスクリプトやロガーの作成といった評価環境の準備時間（導入コスト）を要することも確認した．

準備時間の削減方法として，テストスクリプトやロガー作成の流用や自動化が考えられる．テストスクリプトの作成は，基本的に図 5 の手順となる．そのため，テストスクリプトは，言語や検証対象モジュールに依存するが，一度作成すれば，少しの変更で流用できると考えられる．一方，ロガーに関しては，同一のメソッドや関数に対しては流用可能であるが，新規メソッドには，新たなロガー作成が必要となる．ログとして採取する入力値や出力値が全て，値のみのプリミティブ型であれば，ログ採取も自動的に可能である．しかし，ポインタやオブジェクト型である場合，自動化には手の込んだ仕組みが必要となる．また，今回のように全ての入出力ログが膨大な量になる場合は，採取するログを絞る必要がある．今回は人手で行ったが，準備コストを

ⓒ2009 Information Processing Society of Japan

低減するには，この作業のコスト削減も課題となる．

4.4.2 有用性に関する実験結果，考察

区別できなかったミュータントを解析することにより，使われないコードの発見に役立つことや，テスト入力，操作の不足の発見に役立つことを確認した（4.3 節）． 一方，テストケース（単体）が提案方法におけるミューテーションテストの結果を左右することを確認した（4.3 節）．

単体テストにより，ミューテーションテストの結果が変わることは，提案方法が，単体テストがミュータントを区別できるかを測定しており，システムテストが区別できるかを厳密には測定していないためである．例えば，図 9 のように，y = (2*x)^2 の計算をするため，y = 2*x と y = x^2 の二つのモジュールに分割したシステムを考える．このシステムに対し y = 2*x を検証対象モジュールとし，システムテストとして入力 3，期待値 36 を用い，提案方法を実施することを考える．すると，単体テストとして，入力 3，期待値 6 のテストケースが得られる．ここで，検証対象が y = -2*x に改変されるとする．この場合，単体テストでは，期待値 6 に対し，結果が-6 となるため，本提案方法ではミュータントを区別する．一方，元のシステムテストでは，期待値も結果も 36 となり，ミュータントを区別しない．

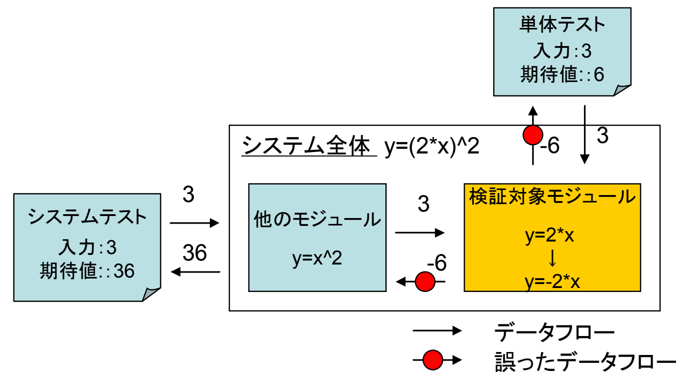

図 9 単体テストで発見する不具合をシステムテストで発見できない例

以上のように，本提案方法で区別するミュータントは，システムテストで必ずしも区別できるとは限らないことが分かる．このような区別するミュータントの違いが，本提案方法によるミューテーションテスト結果を活用する際に，重大な影響を及ぼす可能性がある．特に，この違いによりシステムテストのテスト削減への活用性にどれだけ影響するかの評価や，影響が生じる場合はその対策方法を施すことが必要となり，今後の課題である．

5. おわりに

本稿では，派生開発におけるシステムテストのテストケース評価を行うために，ミューテーションテストを適用する方法を提案した．提案方法の適用実験を行い，時間コストと有用性の視点で提案方法の評価を行った．

時間コストに関しては，ミューテーションテストによる評価時間を単体テストレベルに削減することを確認した．システムレベルのテストケースへの適用が困難であった一番の要因は評価時間を要することであり，この課題の克服は大きな前進である．しかし，提案方法を行うための準備にコストを要することも確認した．準備コストの削減が今後の課題となる．

有用性に関しては，使われないコードの発見や，テストケースの不足発見に役立つことを確認した．しかし，提案方法で区別するミュータントは，システムテストで必ずしも区別できるとは限らない．この性質が，ミューテーションテストの結果にどれだけ影響するか評価する必要がある．特に，システムテストのテスト削減へ活用するのに影響が無い範囲であるのか評価する必要がある．影響ある範囲であれば，その範囲に抑える方法の提案が今後の課題となる．

謝辞 本研究を行うに当たり，多くの知識や示唆を頂いた米国ワシントン大学の Michael Ernst 教授に，謹んで感謝の意を表する．

参考文献

[1] Harrold, M.J. et al. : A methodology for controlling the size of a test suite, ACM Transactions on Software Engineering and Methodology, vol 2, pp 270-285 (1993)
[2] DeMillo, R. et al. : Hints on Test Data Selection: Help for the Practicing Programmer, Computer, vol 11, pp 34-41 (1978)
[3] Mathur, P.A. et al.: Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An Empirical Study, Software Quality Journal, vol 4, pp.69-83 (1993)
[4] Jumble 公式サイト，http://jumble.sourceforge.net/
[5] Andrews,J.H.: Is Mutation an Appropriate Tool for Testing Experiments, Proceedings of the 27th international conference on Software engineering, pp.402-411 (2005)
[6] The Eclipse Foundation,「Eclipse」, http://www.eclipse.org/
[7] Object Mentor inc,「Junit」, http://junit.org/

ⓒ2009 Information Processing Society of Japan