Vol.2015-SE-187 No.1
2015/3/12

JavaScriptを対象とした等価ミュータント自動検出に関する研究

嶌津 達也[1,a)]　高田 眞吾[1]　丹野 治門[2]　生沼 守英[2]

概要：ミューテーションテストはテストスイートの質を評価するための手法である．そこでは，元のプログラムに対して簡単な変更を行い，その２つのバージョンに対して，同じ入力を与えた場合出力結果が異なるかどうかに注目する．しかし，どんな入力を与えても出力結果が同じとなる等価ミュータントの問題がある．等価ミュータントを検出するための手法に関する研究は存在するが，昨今多くの Web アプリケーションで用いられている JavaScript 言語を対象とした研究はあまりない．本研究では，JavaScript を対象とした等価ミュータントの自動検出を様々な観点から行い，評価した．

キーワード：ミューテーションテスト，等価ミュータント，JavaScript

1. 序論

ソフトウェアテストはソフトウェアの品質を保証する上で非常に重要な工程である．しかし，ソフトウェアは日々複雑になっており，テストが十分かどうかを判断することは難しい．そこでミューテーションテストというテストスイートの質を評価する手法が存在する．

ミューテーションテストは元のプログラムに対して人工的にバグを埋め込むことでテストスイートの質を評価する手法である．この手法は，元のプログラムとバグを埋め込んだプログラム（ミュータント）に同じ入力を与えてそれらの出力を比較し，それらの出力が異なっていればバグを検出できたと考える．これを kill と呼び，ミュータントを kill できれば現実の同様のバグも検出できるであろうという考え方にこの手法は基づいている．

ミューテーションテストはテストスイートの質を評価する強力な手法であるが，大きく分けて２つの問題を抱えている．ひとつは計算量が非常に大きいことである．これは生成されるミュータントそれぞれに対してテストケースを繰り返し実行しなければならないからである．また，もうひとつの問題として等価ミュータントという問題がある．本研究では後者の問題に注目する．

等価ミュータントとは，ミューテーションを行ったプログラムの振る舞いが元のプログラムと変わらないようなミュータントのことである．等価ミュータントはテストスイートの質の評価を妨げてしまうため，等価ミュータントの検出に関する様々な研究が行われてきた．

既存の研究では，Java や C，Fortran といった言語を対象に研究がされている [1]．しかし，JavaScript を対象としたミューテーションテストはこれらの言語と比べてあまり研究が進んでいない．これは JavaScript が多くの Web ブラウザで利用できるようになって広く普及したのが 1990 年代の後半であり，注目され始めたのが比較的最近であることが理由として考えられる．また，JavaScript に特化したミューテーションツールは 2013 年に初めて開発された [2] ため，JavaScript におけるミューテーションテストの歴史は他の言語よりも浅い．そのため，どのような等価ミュータント自動検出手法が有効であるかは分かっていない．

そこで，本研究では他言語で研究されてきた既存の等価ミュータントの自動検出手法が JavaScript において有効であるかを検証する．

本研究の貢献は，次の二つである．

(1) JavaScript におけるカバレッジ，変数の値に着目した等価ミュータント自動検出手法の有効性の評価
(2) JavaScript における等価ミュータント自動検出の既存手法の適用時の問題点の発見

以降，2 章ではミューテーションテストと等価ミュータントについて述べる．3 章では等価ミュータント検出に関する既存研究とそれらに対する問題を述べる．4 章では既存の JavaScript 用ミューテーションツールを拡張した検証

[1] 慶應義塾大学
Keio University
[2] NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Tokyo, Japan
[a)] tatsuya-shimazu@a5.keio.jp

機構を提案する．5 章では評価について述べ，最後に 6 章で結論を述べる．

2. ミューテーションテスト

本章ではミューテーションテストについて述べる．

2.1 ミューテーションスコア

ミューテーションテストによるテストスイートの質の評価指標としてミューテーションスコアがよく用いられる [1]．この計算式は次のような式で表される．

$$Mutation\_Score = \frac{number\ of\ killed\ mutants}{number\ of\ non{-}equivalent\ mutants} \quad (1)$$

非等価ミュータント（non-equivalent mutant）とは何らかのテストケースによって kill 可能なミュータントのことである．このミューテーションスコアを測定することでそのテストスイートの質を定量的に評価することができる．また，ミューテーションスコアはテストスイートを改善させる指標でもある．ミューテーションスコアの最大値は 1 であり，これは非等価ミュータントの数と kill できたミュータントの数が同じときである．この場合，テストスイートは十分な質であることが分かる．もしミューテーションスコアが 1 ではない場合，テスターは kill できなかったミュータントを参考にして，それを kill 可能なテストケースをテストスイートに追加する．これにより，テストスイートの検出できるバグの種類が増えるので，テストスイートの改善が可能となる．

2.2 等価ミュータント

等価ミュータントとは，ミューテーションを行ったプログラムの振る舞いが元のプログラムと変わらないようなミュータントのことである．そのため，等価ミュータントはどのようなテストケースを用いても出力結果が変わらない．

前節で述べたミューテーションスコアはテストスイートの質の評価と改善の指標であるが，このミューテーションスコアの計算をする際には等価ミュータントを取り除かなければならない．等価ミュータントは元のプログラムと同じ振る舞いをしてしまうので，本来は質が良いテストスイートでも等価ミュータントが多ければ，kill 可能なミュータントの割合は少なくなってしまう．そのため，等価ミュータントを含めるとミューテーションスコアが不当に低くなってしまう．また，等価ミュータントを kill 可能なテストケースは作成することができないので，テストスイートの改善に役立たない．

等価ミュータントを除外することはミューテーションテストにおいて重要な課題であるが，この等価ミュータントを手動で検出するのは非常に手間がかかる．例えば，Schuler らのケーススタディでは，ひとつのミュータントが等価であるかを手動で判定するのに平均 15 分かかった [3]．このような等価ミュータントを完全に自動的に検出する技術は未だなく，一般的には解決不可能な問題であると言われている [4]．

3. 関連研究

この章ではミューテーションテストにおける等価ミュータントの検出に関する既存研究について述べる．

3.1 等価ミュータント検出支援

Hierons らの研究 [5] では，静的スライスを用いた等価ミュータント検出を支援する手法を提案した．静的スライスとは，プログラムのある一文の変数に着目し，その変数の計算に関係のあるコードを抽出する技術である．Hierons らは出力に影響する変数に着目して静的スライスを行い，プログラムを単純化することでミュータントが等価であるかの判断を容易にした．しかし，Hierons らの研究では静的スライスを行った後は手動によって等価ミュータントの検出が必要であるという問題がある．

3.2 等価ミュータント自動検出

Offutt らの研究 [6] では，等価ミュータントの検出を，実行可能パスがミューテーションの影響を受けるかどうかの問題であるとして，次の条件を基にした制約解決アルゴリズムを提案した．このアルゴリズムは次の 3 つの条件をすべて満たしていればそのミュータントは非等価であると判定する．逆に，ミューテーションがこのうちの一つでも満たしていなければそのミュータントは等価であると判定する．

- 適用したミューテーションに到達可能である．
- ミューテーションを行ったステートメントの実行後に不正な状態を引き起こす．
- ミューテーションが最終状態に影響を与える．

このアルゴリズムを用いて 11 のプログラムを対象として実験を行った結果，等価ミュータントの 48%を検出することができた．しかし，この手法を JavaScript に適用しようとした場合，JavaScript は Web アプリケーションで用いられることが多いため，Web ブラウザ上の表示などの最終状態に影響を与えているかを考慮することが問題となる．

Schuler らの研究 [7] では変数の不変条件に着目した等価ミュータント検出手法を提案した．この手法はミュータントが変数の不変条件に違反するものは非等価ミュータントである可能性が高いという考えに基づいている．しかし，この手法はそもそも不変条件に違反するミュータントが少ないため，本来非等価であるミュータントの多くを等価と判定してしまうという欠点があった．

Schuler らはその後の研究 [3] で，各ステートメントの実行回数とメソッドの返り値に着目した等価ミュータント自

動検出手法を提案している．ミュータントの各ステートメントの実行回数，もしくはメソッドの返り値が元のプログラムと比べて変化していれば，そのミュータントを非等価であると判定する手法である．この手法は不変条件に着目する手法と比べて判定の精度を大きく改善した．

3.3 JavaScript 用のミューテーションツール

JavaScript を対象としたミューテーションテストはまだあまり研究が進んでおらず，2013 年に Mirashokraie らによって初の JavaScript 用のミューテーションツール，Mutandis が作成された [2]．Mutandis は対象の JavaScript プログラムに対して自動的にミューテーションを行い，ミュータントを生成する．また，ミューテーションオペレータとして既存のミューテーションオペレータ以外にも独自に JavaScript 特有のミューテーションオペレータを提案して用いている．しかし，JavaScript を用いた Web アプリケーションの主な特徴の一つであるイベント駆動を考慮したミューテーションオペレータは提案されていない．

JavaScript を対象としたミューテーションツールは他にも開発されており，Nishiura らは AjaxMutator というミューテーションツールを開発している [8]．AjaxMutator では JavaScript の主な特徴であるイベント駆動，非同期通信，DOM 操作の 3 つを考慮したミューテーションオペレータを実装している．

4. 等価ミュータント検出機構

本研究では AjaxMutator[8] を一部変更，拡張したものを用いて，変数の値，カバレッジに着目した等価ミュータント自動検出手法が JavaScript において有効であるかを検証した．検証に用いる JavaScript を対象としたミューテーションテストの機構の概要を**図 1** に示す．図 1 は，ミュータントの生成から等価ミュータント検出までの流れを図示したものである．本章では，検証に用いる機構について，ミューテーション部，テスト実行部，等価ミュータント検出部の 3 つに分けて述べる．

4.1 ミューテーション部

ミューテーション部では対象とする JavaScript のプログラムを読み込み，ミューテーションオペレータに従ってソースコードにミューテーションを行い，ミュータントを生成する．生成したミュータント（JavaScript プログラム）を Web アプリケーションが読み込み，バグを含んだ Web アプリケーションを生成する．

ミューテーション部は，AjaxMutator を拡張することによって実現した．ミューテーションオペレータは AjaxMutator で提供されているものに加えて，様々な言語で一般的に使用されているものを用いた．本機構が対象とするミューテーションオペレータは次の通りである．

- ユーザイベント登録
  - イベントターゲットの置換
  - イベントタイプの置換
  - イベントコールバックの置換
- タイマーイベント登録
  - タイマー時間の置換
  - タイマーコールバックの置換
- 非同期通信
  - リクエストターゲットの置換
  - リクエスト成功時のコールバックの置換
- DOM 操作
  - 隣接する DOM 要素への置換
  - 割り当て先の属性の置換
  - 属性に割り当てる値の置換
- その他のミューテーションオペレータ
  - 計算演算子の置換
  - 関係演算子の置換
  - 変数への代入値の置換
  - var 宣言の削除

4.2 テスト実行部

テスト実行部では生成したミュータントを含んだ Web アプリケーションに対して，あらかじめ用意されたテストケースをブラウザ上で自動的に実行する．このテストケースには対象とする Web アプリケーションに対する操作を記述する．なお，テストケースには元の Web アプリケーションの操作をすべて行えるものを用いる．あるミュータントを含んだ Web アプリケーションにおいて，一つでもテストケースに記述された操作が行えなかった場合は，元の Web アプリケーションで行えた操作ができなくなっているのでバグを発見できた（kill）と考える．もし，すべての操作が行えた場合は元の Web アプリケーションと同様の操作が行えるためバグを発見できなかったと判断する．

テスト実行部も AjaxMutator を拡張することによって実現した．AjaxMutator では元の Web アプリケーションに対してテストを実行していなかった．しかし，本研究では元の Web アプリケーションの実行情報が必要となるため，元の Web アプリケーションに対してもテストの実行を行う．この実行情報は等価ミュータント検出部で必要となる変数の値，カバレッジである．詳しくは次節で述べる．

次にテスト自動実行の流れを述べる．

(1) ブラウザを起動する．
(2) テスト対象の Web アプリケーションのページを開く．
(3) テストケースを実行する．
(4) 実行情報を保存する．
(5) 2〜4 の操作を全テストケースで行う．
(6) テスト結果とミュータントの kill の結果を保存する．
(7) ブラウザを終了する．

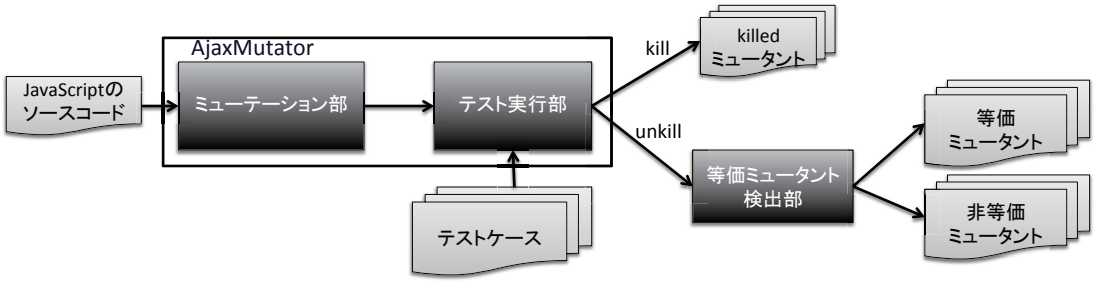

図 1: 検証機構の概要

上記の操作を元のプログラム，ミュータントに対して繰り返し行う．

4.3 等価ミュータント検出部

等価ミュータント検出部ではテスト実行部で kill されなかったミュータントを対象として，各等価ミュータント自動検出手法を用いて等価，非等価を判別する．等価ミュータント検出部では，まずテスト実行部で記録した実行情報を読み込む．この読み込んだ実行情報を基に各等価ミュータント検出手法でミュータントが等価，非等価を判別する．最後にすべてのミュータントの判別結果を一覧として保存する．

本研究では次に挙げる等価ミュータント自動検出手法を検証の対象とした．

- カバレッジに着目した等価ミュータント自動検出手法
- 変数に着目した等価ミュータント自動検出手法

次に，上記に挙げた等価ミュータント自動検出手法について述べる．

4.3.1 カバレッジに着目した等価ミュータント自動検出手法

実行時のカバレッジに着目した手法は Schuler らが Java 言語を対象に提案した [3]．Schuler らが行った研究ではソースコードの 1 行毎の実行回数に着目して等価ミュータントを検出している．本研究ではカバレッジ計測のために JSCover[*1] を用いた．JSCover によってブラウザ上で動作する Web アプリケーションの JavaScript プログラムにインストルメント処理を行う．これにより Web アプリケーション内の JavaScript プログラムのコードカバレッジを計測する．本研究では次の 3 種類のカバレッジに着目してそれぞれ等価ミュータントの検出を行った．

- ラインカバレッジ
  各行の実行回数
- ファンクションカバレッジ
  各関数の実行回数
- ブランチカバレッジ
  各分岐の実行回数

この手法の等価ミュータントの検出方法は，すべてのテストケースで実行時のカバレッジが一致していれば等価ミュータントであると判定する．手順として，まず元のプログラムとそのミュータントに対してそれぞれ同じテストケースを実行する．そして，そのテストケースの実行時のカバレッジが一箇所でも異なる場合は非等価ミュータントであると判定する．もし，すべて一致している場合は次のテストケースの実行時のカバレッジを元のプログラムとミュータントで比較する．

4.3.2 変数に着目した等価ミュータント自動検出手法

変数に着目した手法ではすべてのテストケースで変数の値が一致していれば等価ミュータントと判定する．手順として，まず元のプログラムとミュータントに対して同じテストケースを実行する．そして，テスト終了時の変数の値に一つでも値が異なる変数が存在すれば非等価ミュータントであると判定する．もしすべて一致している場合は次のテストケースの終了時の変数の値を元のプログラムとミュータントで比較する．

なお，変数に着目した等価ミュータント自動検出手法の基となった手法としてメソッドの返り値に着目した手法が Schuler らによって提案されている [3]．Schuler らの手法では，元のプログラムとそのミュータントに対して同じテストケースを実行し，メソッドの返り値がすべて一致していれば等価ミュータントであると判定している．

5. 評価

ケーススタディは Quizzy[*2]（PHP と AJAX ベースのライブラリ）を用いて作成された Web アプリケーションに対して行った．この Web アプリケーションは Nishiura らが AjaxMutator の評価のために作成したものである．また，このケーススタディに用いたテストケースも Nishiura らが作成したものである．対象の Web アプリケーション

[*1] http://tntim96.github.io/JSCover/

[*2] http://quizzy.sourceforge.net/

表 1: 生成したミュータントのテスト結果 (単位：個)

	killed	unkilled	総数
ミュータント	160	84	244

に対して生成されたミュータント数，killed ミュータントの数，unkilled ミュータントの数を**表 1** に示す．

ケーススタディの評価指標として precision と recall を用いる．precision と recall は（2）式と（3）式で表す．本研究における precision は等価と判定した中に，実際どれだけの等価ミュータントが含まれているのかの割合を表す．precision が高いほど，等価と判定したときの信頼性が高いことを示している．また，recall は等価ミュータントのうち，等価と判定できたものの割合を表す．recall が高いほど，実際に等価であるミュータントを見逃さないことを示している．

$$precision = \frac{\text{正しく等価と判定されたミュータント}}{\text{等価と判定されたミュータント}} \times 100 \quad (2)$$

$$recall = \frac{\text{正しく等価と判定されたミュータント}}{\text{実際に等価であるミュータント}} \times 100 \quad (3)$$

これら 2 つの指標，precision と recall はトレードオフの関係にある．そのためどちらか一方が高いだけでは精度が良いと判断することができない．そこで F 値を用いた．F 値とは precision と recall を統合した値でこの値が高いほど判定精度が良いことを示す．F 値は（4）式で表す．

$$F\_measure = \frac{2 \times precision \times recall}{precision + recall} \quad (4)$$

5.1 評価結果

各等価ミュータント自動検出手法の判定の結果をそれぞれ**表 2**, **表 3**, **表 4**, **表 5** に示す．また，precision，recall，F 値を用いた結果を**表 6** に示す．variable が変数による等価ミュータント自動検出手法を表し，line，function，branch がそれぞれラインカバレッジ，ファンクションカバレッジ，ブランチカバレッジによる等価ミュータント自動検出手法を表す．

表 6 より，precision はどの手法もあまり差はないが，recall はカバレッジによる等価ミュータント自動検出手法がすべて 100%と非常に高い精度となった．また，F 値を比較すると，ラインカバレッジとファンクションカバレッジの精度が最も高いことがわかる．この結果からカバレッジを用いた等価ミュータント自動検出手法は JavaScript において有効であると考えられる．

5.2 考察

提案機構の有効性と問題点について，本節で考察を行う．

5.2.1 変数による等価ミュータント自動検出手法の recall

表 6 より，変数による等価ミュータント自動検出手法は recall が低く，有効な手法とは言えない．この理由として，

表 2: 変数による判定結果 (単位:個)

		手動判定	
		等価	非等価
自動判定	等価	58	11
	非等価	10	5

表 3: ラインカバレッジによる判定結果 (単位:個)

		手動判定	
		等価	非等価
自動判定	等価	68	12
	非等価	0	4

表 4: ファンクションカバレッジによる判定結果 (単位:個)

		手動判定	
		等価	非等価
自動判定	等価	68	12
	非等価	0	4

表 5: ブランチカバレッジによる判定結果 (単位:個)

		手動判定	
		等価	非等価
自動判定	等価	68	14
	非等価	0	2

表 6: 各等価ミュータント自動検出手法の判定精度 (単位 : %)

	variable	line	function	branch
precision	84	85	85	83
recall	85	100	100	100
F 値	85	92	92	91

変数の初期値にミューテーションを行ったことが原因の一つと考えられる．サンプルコード 1 に例を示す．サンプルコード 1 のように初期値にミューテーションを行った変数が上書きされてから使われる場合，変数の初期値に関わらず挙動は変わらない．そのため，実際には等価ミュータントである．しかし，テストケースの最後の操作で init() を呼び出す操作を行うと，変数が初期値に戻るため，最終的な変数の値が異なる．変数による等価ミュータント自動検出手法はテストケースの実行終了時の変数を比較しているため，このミュータントは非等価ミュータントと判定されてしまう．このような誤判定が考えられるため，変数による等価ミュータント自動検出手法は有効な手法とは言えない．

```
function init() {
  target = null; → target = -1;
  // ミューテーション部分
}
init();
...
target = someObject;
...
```

サンプルコード 1: 初期値のミューテーションの例

5.2.2 ミューテーションオペレータと等価ミュータント自動検出手法の精度の関係

ケーススタディでは，いくつかのミューテーションオペレータに，誤判定が集中していた．それらのミューテーションオペレータについて，各等価ミュータント自動検出手法の判定精度が悪かった理由を考察する．

- タイマー時間の置換

このミューテーションオペレータはタイマー処理に用いる引数にミューテーションを行う．そのため，このミューテーションによってタイマーとして使われる時間が変更されるというケースがある．サンプルコード 2 にその例を示す．

```
setTimeout(callback,1000)
→ setTimeout(callback,2000)
```

サンプルコード 2: タイマー時間の置換の例

サンプルコード 2 の例では元のプログラムは 1000 ミリ秒後に callback 関数を実行するが，このオペレータを適用したミュータントでは 2000 ミリ秒後に callback 関数を実行する．このミューテーションにより，ミュータントでは callback 関数の実行が 1000 ミリ秒遅れる．これは元の Web アプリケーションと動作が異なるので非等価ミュータントである．しかし，今回用いた等価ミュータント自動検出手法では変数やカバレッジには影響しないので，非等価ミュータントと判定することはできない．このような変更を検出するためには Web アプリケーションに対するテストの実行時間を計るといったことが必要であると考えられる．

- 隣接する DOM 要素への置換

このミューテーションオペレータはある DOM 要素をその親もしくは子の DOM 要素に置き換えるミューテーションを行う．このミューテーションによって，ある DOM 要素に登録されるイベントが親もしくは子の DOM 要素に登録されてしまうケースがある．サンプルコード 3 にその例を示す．

```
$("#showItems").click(showItem)
→ $("#showItems").parent().click(showItem)
```

サンプルコード 3: 隣接する DOM 要素への置換の例

サンプルコード 3 の例では元のプログラムは id が showItems である DOM 要素をクリックしたときに showItem 関数を呼び出すというイベントを登録する．しかし，ミュータントでは id が showItems である DOM 要素の親要素をクリックしたときに showItem 関数を呼び出すというイベントを登録してしまう．そのため，元の Web アプリケーションと動作が異なるので非等価ミュータントである．今回用いた等価ミュータント自動検出手法の中で，カバレッジによる手法がこの変更を検出するためには，テストケースが showitems の DOM 要素の親要素をクリックするといった操作を行っていなければならない．なぜなら，カバレッジによる手法はテストケースの実行情報を用いているので showItems の DOM 要素の親要素をクリックした場合のみでしかカバレッジに影響が出ないからである．しかし，showItems の DOM 要素の親要素をクリックした場合は showItem 関数が呼び出されるので，元のプログラムと挙動が異なる．つまり，テストケースを実行した時点で kill されるので，等価ミュータント自動検出手法を用いる必要がない．したがって，今回用いた等価ミュータント自動検出手法では非等価と判定することはできない．このような変更を検出するためには各 DOM 要素に登録されているイベントを常に記録するといったことが必要であると考えられる．

- 計算演算子，関係演算子の置換

このミューテーションオペレータは + を −，> を < など，演算子を置き換えるミューテーションを行う．このミューテーションによって，変数の値の変化や実行パスが通過する分岐の変化が起こる場合がある．サンプルコード 4 とサンプルコード 5 にその例を示す．

```
z = x + y → z = x - y
```

サンプルコード 4: 計算演算子の置換の例

```
if(x == 0) → if(x != 0)
```

サンプルコード 5: 関係演算子の置換の例

サンプルコード 4 やサンプルコード 5 のような例はほとんどがテストケースによって kill できる．しかし，テストケースによって kill されず，かつ手動で非等価と判定されたミュータントは，ミューテーションによる影響が Web ブラウザ上でのアニメーションの挙動に現れるというケースがある．そのような例をサンプルコード 6 に示す．

```
z = x + y; → z = x - y;
  // ミューテーション部分
$("items").animate({left: "0px"}, z);
```

サンプルコード 6: 計算演算子の置換の非等価の例

サンプルコード 6 の例の場合，変数 z が変化することによってアニメーションの速度が変化する．そのため，カバレッジに影響はないが，元の Web アプリケーションと動作が異なるので非等価ミュータントである．しかし，このようなミュータントはカバレッジを用いた等価ミュータント自動検出手法では非等価と判定することはできない．また，この後で z が上書きされるような場合，変数を用いた等価ミュータント自動検出手法では非等価と判定することはできない．このような変更を検出するためにはテストの実行時間を記録したり，テスト実行中の様子を記録するといったことが必要であると考えられる．

以上のような 3 種類のミューテーションオペレータはいずれもミューテーションが JavaScript 特有の動作に影響を及ぼしていた．これらのミューテーションを検出するためには DOM ツリー，イベント駆動，非同期通信といった JavaScript の特徴に着目した等価ミュータント自動検出手法を考案する必要がある．

5.3 各等価ミュータント自動検出手法の判定時間

手動での等価判定は一つのミュータントにつき平均 15 分かかるいうケーススタディが Schuler らによって示されている [3]．これに対して，変数による等価ミュータント自動検出手法は 244 個のミュータントの判定にかかった時間は 1 秒未満であった．しかし，カバレッジによる等価ミュータント自動検出手法は一つのミュータントにつき平均約 1 分かかった．これはカバレッジを計測する際，JSCover をプロキシサーバとして用いたので，JSCover を用いない場合よりもテストケースを実行するのに時間がかかったためである．なお，記録されたカバレッジから等価判定を行う時間はすべてのミュータントを合わせても 1 秒未満であった．また，手動で判定する場合は unkilled ミュータントのみの等価性を判定すれば良いが，今回用いた手法ではすべてのミュータントに対して等価判定に利用するための情報の取得をしなければならない．そのため，その分余計な時間がかかってしまっている．

今回のケーススタディではすべてのミュータントで実行情報取得にかかる時間を考慮しても等価判定の時間は短縮することができた．しかし，今回用いた等価ミュータント自動検出手法はテストケースの数，テストケースの長さ，ミュータントの数に左右されてしまう．今後は様々な規模の Web アプリケーションを用いて，等価ミュータントの自動検出にかかる時間を計測することが必要であると考えられる．

6. 結論

本研究では，JavaScript を対象とした等価ミュータント自動検出手法の検証機構を提案した．また，ケーススタディを行い，各手法の有効性を評価した結果，カバレッジを用いた等価ミュータント自動検出手法が有効であることを確認した．

今後の課題としては様々な規模の実 Web アプリケーションにおける各手法の評価を行うことが挙げられる．本研究では小規模な Web アプリケーションを用いて評価を行ったが，実際の Web アプリケーションでも評価を行う必要があると考えられる．

また，JavaScript 特有の挙動の変化を検出できる手法の考案も今後の課題に挙げられる．他の言語で提案された等価ミュータント自動検出手法では，ミューテーションが DOM ツリーやアニメーションといった JavaScript 特有の挙動にのみ与えている影響を検出することはできなかった．

様々なミューテーションオペレータを用いた場合の等価ミュータント自動検出手法の評価も今後の課題である．現時点では JavaScript に関連するミューテーションオペレータを主に用いて評価を行った．しかし，本研究で用いたミューテーションオペレータ以外を用いた調査も必要であると考えられる．

参考文献

[1] Jia, Y. and Harman, M.: An analysis and survey of the development of mutation testing, *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp. 649–678 (2011).
[2] Mirshokraie, S., Mesbah, A. and Pattabiraman, K.: Efficient JavaScript Mutation Testing, *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 74–83 (2013).
[3] Schuler, D. and Zeller, A.: Covering and Uncovering Equivalent Mutants, *Software Testing, Verification and Reliability*, Vol. 23, No. 5, pp. 353–374 (2013).
[4] Budd, T. A. and Angluin, D.: Two notions of correctness and their relation to testing, *Acta Informatica*, Vol. 18, No. 1, pp. 31–45 (1982).
[5] Hierons, R., Harman, M. and Danicic, S.: Using program slicing to assist in the detection of equivalent mutants, *Software Testing, Verification and Reliability*, Vol. 9, No. 4, pp. 233–262 (1999).
[6] Offutt, A. and Pan, J.: Detecting Equivalent Mutants and the Feasible Path Problem, *The Eleventh Annual Conference on Computer Assurance*, pp. 224–236 (1996).
[7] Schuler, D., Dallmeier, V. and Zeller, A.: Efficient Mutation Testing by Checking Invariant Violations, *Eighteenth International Symposium on Software Testing and Analysis*, pp. 69–80 (2009).
[8] Nishiura, K., Maezawa, Y., Washizaki, H. and Honiden, S.: Mutation Analysis for JavaScript Web Application Testing, *The 25th International Conference on Software Engineering and Knowledge Engineering (SEKE'13)*, pp. 159–165 (2013).