Mutation Analysis for JavaScript Web Application Testing

Kazuki Nishiura and Yuta Maezawa
The University of Tokyo
Tokyo, Japan
{*k-nishiura, maezawa*}*@nii.ac.jp*

Hironori Washizaki
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Shinichi Honiden
The University of Tokyo, National Institute of Informatics
Tokyo, Japan
honiden@nii.ac.jp

***Abstract*—When developers test modern web applications that use JavaScript, challenging issues lie in their event-driven, asynchronous, and dynamic features. Many researchers have assessed the adequacy of test cases with code coverage criteria; however, in this paper, we show that the code coverage-based approach possibly misses some faults in the applications. We propose a mutation analysis approach for estimating the fault-finding capability of test cases. We assume that developers can find overlooked fault instances and improve the test cases with the estimated capability. To create a set of faulty programs, i.e., mutants, we classify the JavaScript features in web applications and then define a comprehensive set of mutation operators. We conducted a case study on a real-world application and found that our approach supported the improvement of test cases to expose hand-seeded faults by an extra ten percent.**

***Keywords*-JavaScript; Mutation Analysis and Testing; Web Applications; Test Criteria**

I. Introduction

Developers implement client-side JavaScript programs (henceforth, JS programs) to make their web applications interactive. JavaScript provides APIs for handling user events, requesting asynchronous messages, and dynamically manipulating Document Object Models (DOM)[1] to rewrite the contents on a web page. Ocariza et al. reported that 97 of the 100 most visited web sites utilized JavaScript [1].

Since event-driven, asynchronous, and dynamic features can increase the complexity of web applications using JS programs (henceforth, JSWAs), researchers have sought to address challenging issues of testing JSWAs [2], [3]. In those researches, they have leveraged code coverage criteria to assess the adequacy of test cases. However, these approaches might not show an absence of faults, even if they explore all statements and branches without exception. This is because JS programs have dynamic characteristics; e.g., assigning any values to a non-existent property does not throw any exceptions (See Section II-C).

Mutation analysis is a fault-based technique that provides strong test criteria. The technique injects artificial faults into the software under test. Fault injection is done by applying mutation operators that represent fault types that developers would like to discover. By running test cases on faulty versions of software, developers can estimate the fault-finding capability of test cases.

[1]http://www.w3.org/DOM

Researchers have indicated the usefulness of mutation analysis for JSWAs [4]. Recently, some researchers have started applying mutation analysis techniques on JSWAs [5]–[7]. However, they focus on the specific characteristics of applications (e.g., preventing cross-site scripting vulnerability) or heuristically choose mutation operators. Their approach may give a high score to test cases that consider some types of faults but do not consider others.

In this research, we try to define a comprehensive set of mutation operators that cover JavaScript's features by conducting a feature analysis of JavaScript used in web applications. We define ten mutation operators and implement a tool for performing mutation analysis on JSWAs. Using the operators, our tool generates mutants from an original JS program. By executing test cases on mutants, developers can learn the fault-finding capability of the test cases and see unexposed fault instances, i.e., unkilled mutants. Developers can then add or modify the test cases so that more faults can be exposed. We evaluated our tool by surveying real faults from a public bug repository and carrying out a case study on a real-world application.

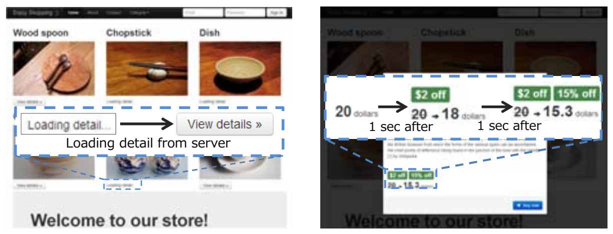
Our contributions are as follows:

1) Proposal of a comprehensive set of mutation operators focusing on the features of JavaScript in web applications.
2) The AjaxMutator, an implementation of our approach.
3) A short survey on real faults and a case study on a real-world application whose results show that our tool can help developers improve their test cases.

II. Background

A. Mutation analysis

Mutation analysis is a fault-based technique to assess the adequacy of test cases [8], [9]. First, the technique makes a small change to a program under test in order to create faulty programs called *mutants*. The changes depend on fault-seeding rules called *mutation operators*. Then, the technique tests both the original program and each mutant with the given test cases. If one of the mutants gives a different test result from the original, the mutant is said



(a) An item list page (b) An item detail view

Figure 1: E-commerce web application

```
1  $(document).ready(function()  {
2    $(".item").each(function()  {
3      var itemId = $(this).attr("data-item-id");
4      var loadingMsg = $("<span>");
5      loadingMsg.addClass("message" + itemId);
6      loadingMsg.text("Loading detail...");
7      $(this).append(loadingMsg);
8      requestItemDetail(itemId);
9    });
10 });
11
12 function requestItemDetail(itemId) {
13   $.getJSON('item.php?id=' + itemId,
14     function(data) {
15       var button = $("#itemButton" + data.id);
16       button.text("View detail...");
17       button.click(
18           showDetail.bind(null, data.body, data.price,
19           data.discount1, data.discount2));
20       $(".message" + data.id).get(0).textContent='';
21       button.show();
22   });
23 }
24
25 function showDetail(bodyHtml, price, discount1, discount2){
26   // ...........
27   detailBody.html(bodyHtml);
28   // ...........
29   setTimeout(showDiscount.bind(details, discount1), 1000);
30   setTimeout(showDiscount.bind(details, discount2), 2000);
31 }
```

Fault1 …$("#itemButton1"…

Fault2 …data.text…

Fault3 …textConent = '';

Fault4 …2000); …1000);

Figure 2: JavaScript program and possible faults

```
public void testShowingDetail() {
  WebDriver driver = new FirefoxDriver();
  driver.get(TARGET_URL);
  WebDriverWait wait = new WebDriverWait(driver, WAIT_LIMIT_SEC);
  wait.until(visibilityOfElementLocated(By.id("itemButton10")));
  WebElement showDetailButton
      = driver.findElement(By.id("itemButton10"));
  assertEquals("View detail...", showDetailButton.getText());
  showDetailButton.click();
  wait.until(visibilityOfElementLocated(By. className("modal")));
  driver.findElement(By.className("buy-now")).click();
}
```

Figure 3: Test case for JavaScript program in Figure 2 (extracted)

to be *killed*. Killed mutants indicate that the test cases can find such faults. Hence, the technique measures the ratio of the number of killed mutants to the number of all created mutants as a *mutation score*. By referring to the mutation score, developers can estimate the adequacy of their test cases.

Although mutation analysis can provide strong criteria for estimating test adequacy [10], its result depends on the definitions of the mutation operators. In this research, we define a comprehensive set of mutation operators that covers all the mandatory features of JSWAs.

B. JavaScript web applications

Modern web applications combine various technologies as described in [11] to realize their interactive web pages. Because JavaScript binds all of these technologies, we argue that JavaScript is a central technology.

To make their applications interactive, developers implement JS programs as follows: When users operate an application, JS programs i) continuously process user requests with event handlers, ii) asynchronously receive the necessary data, and iii) dynamically update web page contents. Analyzing JS programs is a challenging issue because of its event-driven, asynchronous, and dynamic features. Here, we focus on these JavaScript features when defining the mutation operators.

C. Motivating example

We explain the inadequacy of the coverage criteria for testing JSWAs using a typical e-commerce application[2] as our motivating example (Fig. 1). Additionally, we show some of the JS program of the application using jQuery[3] in Figure 2. Here, $\$(''\#foo'')$ and $\$(''.bar'')$ are function calls that select DOM elements whose *ID* and *class* attributes correspond to "foo" and "bar", respectively. $Function\#bind(thisArg[, arg1[, arg2[, ...]]])$ is a built-in method of a *Function* object that sets its $this$ keyword and arguments as provided values.

This application first shows an item list page (Fig. 1a). The page does not initially contain item details about these items to reduce the amount of data that has to be communicated for fast rendering. After the whole page has been loaded, the JS program asynchronously sends requests for the detail of each item to a server (lines 8, 12-23) while displaying a *loading* message (lines 4-7). When it receives a response containing the discount information about the item, the JS program removes the text message (line 20). Instead, it registers an event handler for displaying the information to a $view\ detail$ button (lines 17-19) and then displays the button (line 21). When the user clicks the button, the program rewrites the DOM to show a detailed view of the item (Fig. 1b) without any page transitions (line 27). In this view, the application displays the information using timer events for a visual effect (lines 29 and 30).

Figure 3 shows the test code implemented with `Selenium WebDriver`[4] for testing the program described above. WebDriver enables test designers to emulate user operations such as mouse clicks by implementing them

[2]http://www.honiden.nii.ac.jp/~k-nishiura/e-commerce-example
[3]http://jquery.com

[4]http://seleniumhq.org

Table I. Faults that cannot be exposed with code coverage criteria

Fault	Related feature	Unexpected behavior in our motivating example	Lines in Fig. 2
1	User event target	Registers a "click" event without an exception even if there is no "button".	15 and 17
2	Async. comm. response	Displays an *undefined* value by referring to a non-existent "data.text" property.	18 and 27
3	DOM attr. manipulation	Continuously displays "Loading detail" because of a misspelled "textConent" property.	4-6 and 20
4	Timer event interval	Displays an incorrect price even though both timer events are handled because the intervals are incorrect.	29 and 30

in test code. Given our test code in Figure 3, the framework proceeds as follows: WebDriver first launches Firefox and opens a target web application (lines 2 and 3). The framework waits until the application displays a $view\ detail$ button (line 5). Upon finding the button, the framework clicks the $detail$ button (line 9) and finally clicks a buy button (line 11). Note that when testing web applications, developers need to consider the timing of the user operations. Therefore, the methods of wait object (lines 5 and 10) can be used as assertions that raise an exception when a given condition is not satisfied within a certain period of time.

The test code provides 100% statement and branch coverage for the JS program shown in Figure 2. However, it is not able to find the faults shown in Figure 2. We argue that this inadequate capability derives from the dynamic features of JavaScript. Table I explains these faults. Regarding **faults 1 and 4**, for example, although the JS program explicitly determines the user event targets and timer event intervals, it does not check for their existence or correctness at runtime. As for **faults 2 and 3**, it does not throw any exceptions even if the running application does not have any corresponding properties or DOM elements. Hence, exploring an entire program without exceptions does not always indicate an absence of faults. In the next section, we propose a mutation analysis considering JavaScript features.

III. Mutation Analysis

Figure 4 shows an overview of our mutation approach:

1) Developers implement test cases to test whether a JSWA runs as expected.
2) Our tool generates mutants of JS programs with our proposing mutation operators.
3) Our tool executes the test cases on the mutants to check if the test cases detect the mutants. Developers can know the adequacy of the test cases (i.e., mutation score) and which mutants remain unkilled with given test cases.
4) Developers add test cases to kill the unkilled mutants. Then, our tool recalculates the mutation score by running additional test cases on unkilled mutants. This process is repeated until the mutation score reaches a certain threshold [12].

In this way, our approach can help developers make test cases with better fault-finding capability.

To expose faults in JS programs, we need to define mutation operators by focusing on JavaScript features. Therefore,

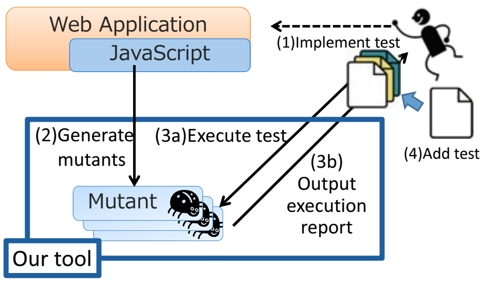


Figure 4: Workflow to improve test suites by mutation analysis

we first conducted a feature analysis on JavaScript. Then, we defined the mutation operators based on the results of the analysis.

A. JavaScript features in web applications

[13] describes three characteristics that distinguish JSWAs from traditional web applications; event driven model, asynchronous communication, and DOM manipulation. We conducted a feature analysis on each feature and developed feature diagrams [14] as shown in Figure 5.

Event driven model: In comparison with traditional web applications, JSWAs leverage JS programs for processing user operations and elapsing time without page transitions. For instance, when a user clicks the "item detail" button (user event), our motivating example displays the discount information after a second (timer event).

When implementing user events in JS programs, developers determine the `target`, `event type`, and `callback function`. The target corresponds to the built-in Window object or to DOM elements such as buttons. JS programs register a callback function to the event type of the target. As for timer events, JS programs register a `callback function` to an `interval` of elapsed time. Developers can also optionally determine `repeat`, i.e., whether the program repeatedly handles the timer event.

Asynchronous communication: Asynchronous communication enables web applications to continuously accept user operations while waiting for server responses. For instance, our motivating example lets the users browse an item list while it loads the details of each item.

The two main constituents of asynchronous communication are `requests` and `responses`. A request must contain a destination URL and a request method (e.g., GET, POST, etc.). Request parameters such as item IDs are optionally included. A server processes the request and sends a response to the application. A response contains the

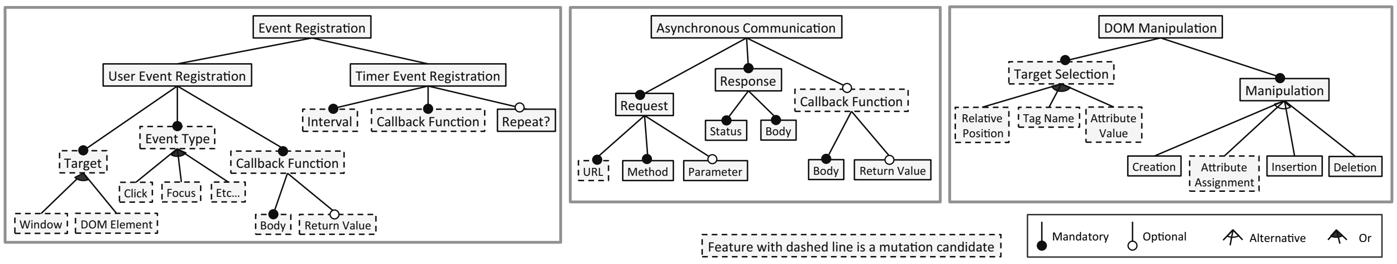


Figure 5: Feature models of JavaScript in web applications

Table II. Proposing mutation operators based on features of JavaScript in web applications and example mutations.

JavaScript feature	Operator name	Original code	Mutated code
User event registration	Event target replacement	buyButton.click(requestBuy)	**cancelButton**.click(requestBuy)
	Event type replacement	button.click(showDetail)	button.**mouseover**(showDetail)
	Event callback replacement	cancelButton.click(closeModal)	cancelButton.click(**requestBuy**)
Timer event registration	Timer interval replacement	setTimeout(callback, 1000)	setTimeout(callback, **2000**)
	Timer callback replacement	setTimeout(showDiscount1, 1000)	setTimeout(**showDiscount2**, 1000)
Asynchronous communications	Request target replacement	$.get('item.php', showItem)	$.get(**'item_list.php'**, showItem)
	Request onsuccess callback replacement	$.get('item.php', showItem)	$.get('item.php', **buyItem**)
DOM manipulation	Nearby DOM element	$("#items").append(newItem)	**$("#items").parent()**.append(newItem)
	Attribute assignment target replacement	element.id = "cancelButton"	element.**textContent** = "cancelButton"
	Attribute assignment value replacement	element.id = "cancelButton"	element.id = **"buyButton"**

status code for signaling success, failure, etc. and a body text. Callback functions are invoked according to the status code if the developers choose to implement them.

Note that JavaScript also provides a means for synchronous communications, but we do not regard it as a feature of JSWAs. Synchronous communication blocks UI threads, and best practice is not to use them.[5]

DOM manipulation: JavaScript provides DOM APIs for manipulating web page elements on the client-side. Such partial updates can make applications more responsive than refreshing whole web pages with page transitions. For instance, our motivating example leverages DOM manipulations to display the item detail view.

DOM manipulations consist of `target` DOM elements and `methods` of manipulating these elements. JS programs select this target element by its position relative to another element, tag name, or ID/class value. As for the method, the programs create, insert, delete the element, or alternatively, assign a value to an attribute of the element.

B. Proposed mutation operators

Here, we describe our ten mutation operators for JS programs (See Table II) and explain how we developed them.

User and timer event registrations: Since developers intentionally implement the optional features of user and timer events, we assume that they typically embed faults in mandatory features, for example, faults 1 and 4 in our example. Hence, we decided to focus on the mandatory features where our approach makes little changes (henceforth, mutation candidates). When mutating a mutation candidate, our approach replaces it with another candidate. For example, consider the `event target` in Table II. Our approach replaces the event target "buyButton" with "cancelButton". Similarly, it also mutates the event types of user events, the timer intervals of timer events, and the callback functions of both.

Asynchronous communications: Although asynchronous communication has two mandatory features, their responses are outside the scope of our study. This study focuses on client-side logic, but the responses depend on the server-side logic. As for the requests, we select only destination URLs as mutation candidates because the differences in the request method should be properly processed by the server-side logic. Additionally, we claim that leveraging responses plays an important role in JSWAs such as when preparing the item details in our motivating example. Therefore, our approach deals with the on-success callback functions as mutation candidates, although this feature is optional.

DOM manipulation: We define a mutation operator called the `nearby DOM element`. This definition is based on our heuristic that developers tend to incorrectly select a DOM element that is near a proper one. Therefore, our approach replaces the target DOM element with its parent or child element.

As for DOM manipulations, our limited implementation does not yet cover creating, inserting, and deleting DOM elements, although such an implementation is planned as future work. Note that our tool can create mutants that insert/delete DOM elements at improper positions and our tool ran as expected to improve test cases in our case study (Section IV-B). As for assigning attributes, we define two mutation operators, one for replacing the attributes

[5]http://blogs.msdn.com/b/wer/archive/2011/08/03/why-you-should-use-xmlhttprequest-asynchronously.aspx

Table III. Real faults in WordPress

Ticket #	JavaScript feature	Brief explanation
1895	User event	Program does not properly register event handlers to user events.
8812		
2184	DOM manipulation	Program creates improper DOM elements
9740		Program selects improper DOM elements

themselves and one for the assigned values.

We implemented our approach in a prototype tool called *AjaxMutator*. This tool is publicly available.[6]

IV. Evaluation Design

To assess the usefulness of our approach, we conducted a short survey about real faults and a case study using our tool. Our research questions are as follows:

RQ1 Can JavaScript features really cause faults?
RQ2 Can developers improve test cases with our tool to find faults that remain unexposed by following the code coverage criteria?
RQ3 Can developers improve the test cases with our tool in a reasonable amount of time?

We first describe design of the survey and case study, and then discuss their results in the next section.

A. Survey: Faults in WordPress

We leveraged the public bug repository of WordPress[7] to survey real faults. In accordance with [15], we searched this repository using the keywords "JavaScript", "js", and "console" and then selected only `closed` bugs from the search results. Next, we manually extracted faults that a JS program clearly caused from the selected bugs. Finally, we extracted the faults related to the JavaScript features discussed in this paper.

B. Case Study: Evaluation of Test Cases for Quizzy

We conducted a case study on a quiz application called Quizzy.[8] This application has 5561 lines of code, including 310 lines of a JS program. We prepared two initial test cases using WebDriver. One represented a normal use case in which users answer quizzes and see their total score. Another is for testing invalid use cases in which users click an answer button before selecting any answer candidates. We conducted the mutation analysis with our tool and added test cases to kill unkilled mutants.

After that, we evaluated how well our tool can assess the fault-detecting capability of the test cases with hand-seeded faults. We asked an undergraduate student with two years industrial experience developing JSWAs to seed typical faults into the application. While he seeded faults, we did not explain our work to him.

[6]https://github.com/knishiura-lab/AjaxMutator
[7]http://core.trac.wordpress.org
[8]http://quizzy.sourceforge.net

Table IV. Detail of our initial and improved test suites

Test suite	#TC	#A	#W	Cov(%)	MS	FF(%)
Initial	2	6	16	95	45.9	89.5
Improved	+5	+21	+21	100	67.0	100.0

The manual setup of the application, the test cases implemented, and the details of the seeded faults are publicly available.[9]

V. Results and discussion

Reality of JavaScript faults (RQ1): In our survey, we found 26 closed bugs that JS programs clearly caused. Among these bugs, we extracted four faults related to our focus in Table III. Note that the other faults were logic faults such as those related to parsing strings, and we expect that the existing approaches can deal with them. However, the faults related to JavaScript features are less studied, and they are more difficult to find, as we discussed in Section II-C. Hence, we claim that JavaScript features can cause real faults that should be exposed.

Improving test cases (RQ2): In our case study, the participant seeded 20 faults in the Quizzy application. Note that we dealt with 19 faults in total, because one seeded fault did not change the behaviour of the applications. These 19 faults consisted of one user event fault, one asynchronous communication fault, two DOM manipulation faults, and other faults such as typos.

Table IV shows the details of the initial and improved test suites. To compare the sizes of the test suites, we list the number of test cases (`#TC`), assertion statements (`#A`), and wait statements (`#W`). To evaluate the adequacy of the test cases, we measured statement coverage using jsCoverage[10] (`Cov`), mutation score (`MS`), and the ratio of found faults to the 19 seeded faults (`FF`).

Although the initial test suite covered all of the statements where faults were seeded, it exposed only 17 faults among the 19 faults (about 10% of the seeded faults were unfound). In addition, their mutation score was low. For instance, in this application, users can choose an option in two ways: by clicking a radio button or clicking a label. Initial test cased only care for clicking a radio button, so they did not kill the mutants that only affected labels. Additional test cases were implemented to kill such unkilled mutants. By adding five test cases, we could improve mutation score by about 20 and it was able to expose all hand-seeded faults. These results suggest that developers can find unexposed faults by increasing the mutation score with our mutation analysis.

As for the mutation analysis, our tool generated 109 mutants, and we divided them into four groups (Fig. 6). The blue bar in the figure stands for mutants killed by our

[9]https://github.com/knishiura-lab/
[10]http://siliconforks.com/jscoverage

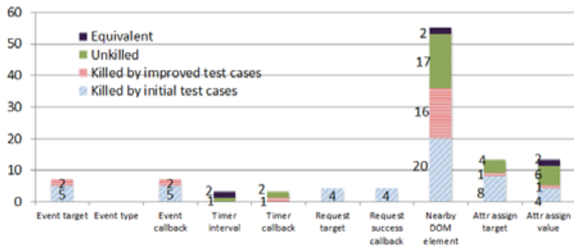


Figure 6: Details of mutants generated by our tool.

initial test cases. The red bar indicates the mutants which the initial test cases could not kill but the improved test cases could. The green and purple bars represent unkilled mutants with both test cases and equivalent mutants.

Note that eliminating equivalent mutants is a challenging issue in the field of mutation analysis [9]. As for the unkilled mutants with either test case, we found another challenging issue for mutation analysis caused by the robustness of modern web browsers; browsers could automatically infer and set proper values at runtime independently of the JS programs. Despite these issues, we believe that our tool can generate enough mutants for developers to improve test cases in order to find unexposed faults.

Reasonable time (RQ3): It took us an hour to prepare test cases and another three hours to improve them. We argue that a few hours is a reasonably short among all cost required to assure the quality of JSWAs that may contain unexposed faults. Additionally, we can explore applying automated test case generation techniques [2], [3], [16] to our tool in order to reduce the manual cost. As for the execution cost, it took the tool 20 minutes to conduct the mutation analysis on the initial test cases. After that, we added test cases to increment mutation score, and recalculated mutation score, which took another 20 minutes. Although this was a reasonably short time, because mutants are independent, we can further reduce this time by modifying the tool to test several mutants in parallel.

Internal validity threats: In a case study we use a real-world application that authors of [17] have also leveraged in their experiments. Moreover, the faults were seeded by a student who did not know about our study. However, we cannot assure these faults represent real faults on JSWAs. Additionally, we prepared the initial test cases by considering possible use cases of the application and improved them by referring unkilled mutants, however, our knowledge of the proposed method could have potentially affected the result of our case study. To avoid these threats, we plan to conduct additional case studies on a real-world development.

External validity threats: Although our set of mutation operators covers the mandatory JavaScript features, we consider to define operators for the optional features in our future work. Moreover, we selected the WordPress bug repository for our survey, and although there are only a few public repositories containing JavaScript bugs, we can define other mutation operators by modeling more real faults from other repositories. We used a single application in our case study, and it would be interesting to investigate how the mutation operators developed here work for other applications. Note that the Quizzy application has all of the JavaScript features discussed in this paper.

VI. Related Work

Mutation analysis for web applications or JavaScript programs: Mutation analysis has been widely studied since it was first introduced in the 1970s [8], [9]. Praphamontripong and Offutt argued that existing mutation analysis techniques do not consider the characteristics of web applications. They proposed mutation operators for HTML and Java Server Pages [6]. Shahriar and Zulkernine defined mutation operators for PHP and JavaScript to evaluate the adequacy of tests for avoiding cross site scripting vulnerability [5]. Alshraideh conducted mutation analysis on JS programs by applying basic mutation operators such as rewriting arithmetic operands to automate unit testing [18]. Our study developed mutation operators by focusing on characteristics of JavaScript in web applications. It has a different focus from and is complementary to the previous research.

Recently, Mirshokraie et al. have proposed some mutation operators based on the common mistakes that inexperienced engineers make [7]. They also proposed some mutation operators related to DOM and XMLHttpRequest[11] as summarized in Table V. However, they do not consider mutation operators about event-driven nature of JSWAs, while we argue event-driven is one of the main features of JSWAs.

Classification of real world faults: Marchetto et al. [19] and Guo et al. [20] surveyed typical faults in web applications. Because these surveys targeted any faults related to web applications, we argue that faults related to JavaScript are not sufficiently studied.

Ocariza et al. studied run-time JavaScript exceptions by automatically exploring popular web applications in the Alexa ranking [1]. They showed that the current quality assurance for JSWAs was so insufficient that even widely used applications threw run-time exceptions. Since their study utilized run-time exceptions as perceived by users, we claim that the actual faults behind such exceptions are not always clear.

Ferrari et al. took a bottom-up approach that defined mutation operators based on real faults [21]. However, there are only few public bug repositories for JSWAs [15]. In addition, public bug reports are typically written by users. Thus, it is difficult to discern the actual fault causing the reported application behavior. Therefore, we chose a top-down approach that analyzes the features of JavaScript

[11]JavaScript object that provides a means for HTTP communications

Category	Mutation target	Our proposal	Mirshokraie's [7]
User event	3 targets (target, type, callback)	✓	NA
Timer event	2 kinds (interval, callback)	✓	NA
Asynchronous communications	Request destination URL	✓	✓
	Asynchronous or synchronous	NA (Out of scope)	✓
	Callback for a response	✓(replace on-success callback)	✓(change conditions on which callback is invoked)
DOM	DOM element selection	✓(replace with nearby DOM element)	✓(rewrite attribute of selection method)
	DOM attribute assignment	✓	✓
	Element insertion	NA (Future work)	✓
Other JavaScript-related	9 kinds (common mistakes for inexperienced programmers)	NA	✓

✓: Implemented, NA: Not Available

Table V. Comparison of mutation operators by our proposal and those by Mirshokraie et al. [7]

in web applications and then defined mutation operators corresponding to these features.

VII. Conclusion and Future Work

In this paper, we classified the features of JavaScript in web applications and defined mutation operators for them. By using our tool, developers can estimate the fault-finding capability of their test cases. We conducted a brief survey on real faults and a case study using a real-world application. The results of our evaluations indicated that our tool could expose faults including ones missed by coverage criteria. We conclude that our approach can help developers improve their test cases and find more faults.

Our future work will be in three main directions. First, we plan to refine the mutation operators for JavaScript web applications by surveying more real faults and conducting additional case studies. Second, we will use our approach to evaluate other methods that support tests such as automated testing. Third, we plan to investigate a new automated testing algorithm that can kill mutants efficiently.

References

[1] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "Javascript errors in the wild: An empirical study," in *Proc. Int'l Sym. on Softw. Reliability Eng. (ISSRE)*, 2011, pp. 100 –109.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proc. Int'l Conf. on Softw. Eng. (ICSE)*. ACM, 2011, pp. 571–580.

[3] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *Proc. Int'l Conf. on Softw. Eng. (ICSE)*. IEEE Computer Society, 2009, pp. 210–220.

[4] A. Marchetto, P. Tonella, and F. Ricca, "Testing techniques applied to ajax web applications," in *Proc. WS on Web Quality, Verication and Validation*, 2007.

[5] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proc. ICSE WS on Softw. Eng. for Secure Systems (IWSESS)*. IEEE Computer Society, 2009, pp. 47–53.

[6] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *Proc. Int'l Conf. on Softw. Testing, Verification, and Validation WSs*. IEEE Computer Society, 2010, pp. 132–141.

[7] A. Mirshokraie, Shabnam. Mesbah and K. Pattabiraman, "Efficient javascript mutation testing," in *Proc. Int'l Conf. on Softw. Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013, p. to appear.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34 –41, april 1978.

[9] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[10] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Softw. Pract. Exper.*, vol. 26, no. 2, pp. 165–176, Feb. 1996.

[11] Garrett, Jesse James. (2005, Feb.) Ajax: A new approach to web applications. [Online]. Available: www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[12] A. J. Offutt and R. H. Untch, "Mutation testing for the new century," W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ch. Mutation 2000: uniting the orthogonal, pp. 34–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=571305.571314

[13] D. A. Justin Gehtland, Ben Galbraith, *Pragmatic Ajax: A Web 2.0 Primer*. O'REILLY, 2006, ch. Ajax Explained, pp. 61–77.

[14] D. Batory, "Feature models, grammars, and propositional formulas," in *Proc. Int'l Conf. on Softw. Product Lines*. Springer-Verlag, 2005, pp. 7–20.

[15] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah, "Autoflox: An automatic fault localizer for client-side javascript," in *Proc. Int'l Conf. on Softw. Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 31–40.

[16] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. on Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[17] Y. Zheng, T. Bao, X. Zhang, and W. Lafayette, "Statically locating web application bugs caused by asynchronous calls," in *Proc. World Wide Web (WWW)*, 2011, pp. 805–814.

[18] M. Alshraideh, "Complete automation of unit testing for javascript programs," *Computer Science*, vol. 4, no. 12, pp. 1012–1019, 2008.

[19] A. Marchetto, F. Ricca, and P. Tonella, "Empirical validation of a web fault taxonomy and its usage for fault seeding," in *Proc. Int'l WS on Web Site Evolution (WSE)*. IEEE Computer Society, 2007, pp. 31–38.

[20] Y. Guo and S. Sampath, "Web application fault classification - an exploratory study," in *Proc. Int'l Sym. on Empirical Softw. Eng. and Measurement (ESEM)*. ACM, 2008, pp. 303–305.

[21] F. C. Ferrari, J. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," *Proc. Int'l Conf. on Softw. Testing, Verification, and Validation (ICST)*, pp. 52–61, 2008.