Guided Mutation Testing for JavaScript Web Applications

Shabnam Mirshokraie, *Student Member, IEEE Computer Society*,
Ali Mesbah, *Member, IEEE Computer Society*, and
Karthik Pattabiraman, *Member, IEEE Computer Society*

Abstract—Mutation testing is an effective test adequacy assessment technique. However, there is a high computational cost in executing the test suite against a potentially large pool of generated mutants. Moreover, there is much effort involved in filtering out equivalent mutants. Prior work has mainly focused on detecting equivalent mutants after the mutation generation phase, which is computationally expensive and has limited efficiency. We propose an algorithm to select variables and branches for mutation as well as a metric, called $FunctionRank$, to rank functions according to their relative importance from the application's behaviour point of view. We present a technique that leverages static and dynamic analysis to guide the mutation generation process towards parts of the code that are more likely to influence the program's output. Further, we focus on the JavaScript language, and propose a set of mutation operators that are specific to web applications. We implement our approach in a tool called MUTANDIS. The results of our empirical evaluation show that (1) more than 93 percent of generated mutants are non-equivalent, and (2) more than 75 percent of the surviving non-equivalent mutants are in the top 30 percent of the ranked functions.

Index Terms—Mutation testing, JavaScript, equivalent mutants, guided mutation generation, web applications

1 INTRODUCTION

MUTATION testing is a fault-based testing technique to assess and improve the quality of a test suite. The technique first generates a set of mutants, i.e., modified versions of the program, by applying a set of well-defined mutation operators on the original version of the system under test. These mutation operators typically represent subtle mistakes, such as typos, commonly made by programmers. A test suite's adequacy is then measured by its ability to detect (or 'kill') the mutants, which is known as the adequacy score (or mutation score).

Despite being an effective test adequacy assessment method [1], [2], mutation testing suffers from two main issues. First, there is a high *computational cost* in executing the test suite against a potentially large set of generated mutants. Second, there is a significant amount of effort involved in distinguishing *equivalent mutants*, which are syntactically different but semantically identical to the original program [3]. Equivalent mutants have no observable effect on the application's behaviour, and as a result, cannot be killed by test cases. Empirical studies indicate that about 45 percent of all undetected mutants are equivalent [4], [5]. Establishing mutant equivalence is an undecidable problem [3]. According to a recent study [5], it takes on average 15 minutes to manually assess one single first-order mutant. While detecting equivalent mutants consumes considerable amount of time, there is still no fully automated technique that is capable of detecting all the equivalent mutants [5].

There has been significant work on reducing the cost of detecting equivalent mutants. According to the taxonomy suggested by Madeyski et al. [5], three main categories of approaches deal with the problem of equivalent mutants: (1) detecting equivalent mutants [6], [7], (2) avoiding equivalent mutant generation [8], and (3) suggesting equivalent mutants [4]. Our proposed technique falls in the second category (these categories are further described in Section 10).

In this paper, we propose a generic mutation testing approach that guides the mutation generation process towards effective mutations that (1) affect error-prone sections of the program, (2) impact the program's behaviour and as such are potentially non-equivalent. In our work, *effectiveness* is defined in terms of the severity of the impact of a single generated mutation on the applications observable behaviour. Our technique leverages static as well as dynamic program data to rank, select, and mutate potentially behaviour-affecting portions of the program code.

Our mutation testing approach is generic and can be applied to any programming language. However, in this paper, we implement our technique for JavaScript, a loosely-typed dynamic language that is known to be error-prone [9], [10] and difficult to test [11], [12]. In particular, we propose a set of JavaScript specific mutation operators, capturing common JavaScript programmer mistakes. JavaScript is widely used in modern web applications, which often consist of thousands of lines of JavaScript code, and is critical to their functionality.

To the best of our knowledge, our work in this paper is the first to provide an automated mutation testing technique, which is capable of guiding the mutation generation towards behaviour-affecting mutants in error-prone portions of the code. In addition, we present the first JavaScript

• *The authors are with the Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, V6T1Z4, Vancouver, BC, Canada.*
E-mail: {shabnamm, amesbah, karthikp}@ece.ubc.ca.

Manuscript received 19 Dec. 2013; revised 16 Sept. 2014; accepted 28 Oct. 2014. Date of publication 19 Nov. 2014; date of current version 15 May 2015. Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2371458

mutation testing tool in this work. This paper is a substantially expanded and revised version of our paper from early 2013 [13].

The key contributions of this work are:

- A new metric, called *FunctionRank*, for ranking functions in terms of their relative importance based on the application's dynamic behaviour;
- A method combining dynamic and static analysis to mutate branches that are within highly ranked functions and exhibit high structural complexity;
- A process that favours behaviour-affecting variables for mutation, to reduce the likelihood of equivalent mutants;
- A set of JavaScript-specific mutation operators, based on common mistakes made by programmers.
- An implementation of our mutation testing approach in a tool, called MUTANDIS,[1] which is freely available from https://github.com/saltlab/mutandis/.
- An empirical evaluation to assess the efficacy of the proposed technique using eight JavaScript applications.

Our results show that, on average, 93 percent of the mutants generated by MUTANDIS are non-equivalent. Further, the mutations generated have a high bug severity rank, and are capable of identifying shortcomings in existing JavaScript test suites. While the aim of this work is not particularly generating hard-to-kill mutants, our experimental results indicate that the guided approach does not adversely influence the stubbornness of the generated mutants.

2 Running Example and Motivation

Equivalent mutants are syntactically different but semantically equivalent to the original application. Manually analyzing the program code for detecting equivalent mutants is a daunting task especially in programming languages such as JavaScript, which are known to be challenging to use, analyze and test. This is because of (1) the dynamic, loosely typed, and asynchronous nature of JavaScript, and (2) its complex interaction with the Document Object Model (DOM) at runtime for user interface state updates.

Fig. 1 presents a snippet of a JavaScript-based game that we will use as a running example throughout this paper. The application contains four main functions as follows:

1) `startPlay` function calls `setup` to set the dimension of all `div` DOM elements.
2) `setup` function is responsible for setting the `height` value of the `css` property of all the DOM elements with the given class name. The actual dimension computation is performed by calling the `getDim` function.
3) `getDim` receives two parameters `width` and `height` based on which it returns the calculated dimension.
4) Finally, `endGame` sets the `height` value of the `css` property of a DOM element with id `startCell`, to indicate a game termination.

1. MUTANDIS is a Latin word meaning "things needing to be changed."

```
1  function startPlay(){
2   ...
3   for(i=0; i<$("div").get().length; i++){
4     setup($("div").get(i).prop('className'));
5   }
6   endGame();
7  }

9  function setup(cellClass){
10  var elems=document.getElementsByClassName(↩
        cellClass);
11  if(elems.length == 0)
12    endGame();
13  for(i=0; i<elem.length; i++){
14    dimension= getDim($(elems).get(i).width(), $(↩
          elems).get(i).height());
15    $(elems).get(i).css('height', dimension+'px');
16  }
17 }

19 function getDim(width, height){
20  var w = width*2, h = height*4;
21  var v = w/h;
22  if(v > 1)
23   return (v);
24  else
25   return (1/v);
26 }

28 function endGame(){
29  ...
30  $('#startCell').css('height', ($('body').width()↩
        +$('body').height())/2+'px');
31  ...
32 }
```

Fig. 1. JavaScript code of the running example.

Even in this small example, one can observe that the number of possible mutants to generate is quite large, i.e., they span from a changed relational operator in either of the branching statements or a mutated variable name, to completely removing a conditional statement or variable initialization. However, not all possible mutants necessarily affect the behaviour of the application. For example, changing the "==" sign in the `if` statement of line 11 to "<=", will not affect the application. This is because the number of DOM elements can never become less than zero, and hence the injected fault does not semantically change the application's behaviour. Therefore, it results in an equivalent mutant.

In this paper, we propose to guide the mutation generation towards behaviour-affecting, non-equivalent mutants as described in the next section.

3 Overview of Approach

An overview of our mutation testing technique is depicted in Fig. 2. Our main goal is to narrow the scope of the mutation process to parts of the code that affect the application's behaviour, and/or are more likely to be error-prone and difficult to test. We describe our approach below. The numbers below in parentheses correspond to those in the boxes of Fig. 2.

In the first part of our approach, we (1) intercept the JavaScript code of a given web application, by setting up a proxy between the server and the browser, and instrument the code, (2) execute the instrumented program by either crawling the application automatically, or by running the existing test suite (or a combination of the two), and (3) gather detailed execution traces of the application under test.

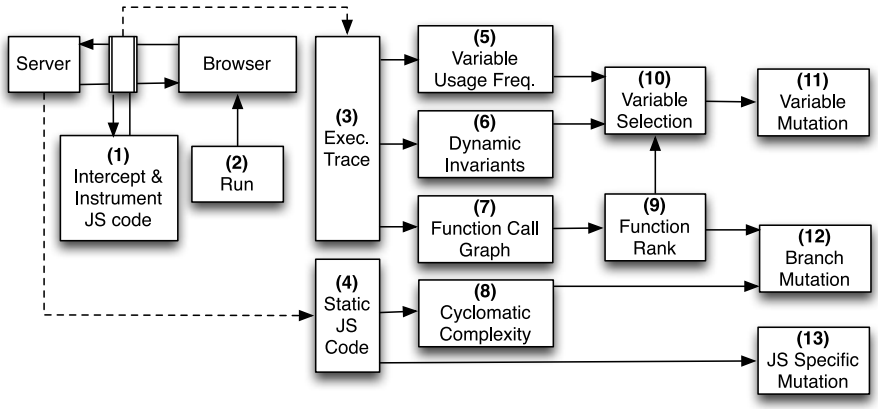


Fig. 2. Overview of our mutation testing approach.

We then extract the following pieces of information from the execution traces, namely (5) variable usage frequency, (6) dynamic invariants, and (7) the functions' call frequency. In addition to dynamically inferred information from the execution traces, we also construct the function call graph of the application by incorporating both static and dynamic information.

Using the function call graph and the dynamic call frequencies, we (9) rank the program's functions in terms of their relative importance from the application's behaviour point of view. The higher a function's ranking, the more likely it will be selected for mutation in our approach.

Further, within the highly ranked functions, our technique (10) identifies variables that have a significant impact on the function's outcome based on the usage frequency and dynamic invariants extracted from the execution traces, and (11) selectively mutates only those variables to reduce the likelihood of equivalent mutants.

In addition to variables, our technique mutates branch conditions, including loops. Functions with high cyclomatic complexity (CC) are known to be more error-prone and challenging to test [14], [15], as the tester needs to detect and exercise all the different paths of the function. We therefore (4) statically analyze the JavaScript code of the web application, and (8) measure its cyclomatic complexity. To perform branch mutation (12), we target the highly ranked functions (selected in 9) that also exhibit high cyclomatic complexity.

In addition to the generic mutation operators, our technique considers (13) a number of JavaScript specific mutation operators, based on an investigation of common errors made by web programmers. These specific operators are applied without any ranking or selection process.

Our overall guided mutation testing algorithm is presented in Algorithm 1. In the following three sections, we describe in detail our technique for ranking functions (Section 4), ranking and selecting variables (Section 5), and performing the actual mutations, including the mutation operators (Section 6).

4 Ranking Functions

In this section, we present our function ranking approach, which is used for selective variable and branch mutation.

4.1 Ranking Functions for Variable Mutation

In order to rank and select functions for variable mutation generation, we propose a new metric called $FunctionRank$, which is based on $PageRank$ [16], but takes dynamic function calls into account. As such, $FunctionRank$ measures the relative importance of each function at runtime. To calculate this metric, we use a function call graph inferred from a combination of static and dynamic analysis (line 6 in Algorithm 1). Our insight is that the more a function is used, the higher its impact will be on the application's behaviour. As such, we assign functions that are highly ranked, a higher selection probability for mutation.

Function call graph. To create a function call graph, we use dynamic as well as static analysis. We instrument the application to record the callee functions per call, which are encountered during program execution. However, the obtained dynamic call graph may be incomplete due to the presence of uncovered functions during the program execution. Therefore, to achieve a more complete function call graph, we further infer the static call graph through static analysis. We detect the following types of function calls in our static analysis of the application's code:

1) Regular function calls e.g., `foo()`.
2) Method calls e.g., `obj.foo()`.
3) Constructors e.g., `new foo()`.
4) Function handlers e.g., `e.click(foo)`.
5) Anonymous functions called by either a variable or an object property where the anonymous function is saved.

We enhance our dynamically inferred call graph of the executed functions by merging the graph with the statically obtained call graph containing uncovered functions. Note that constructing function call graph for the JavaScript applications using static analysis is often unsound due to highly dynamic nature of the JavaScript language. In JavaScript functions can be called through dynamic property access (e.g., `array[func]`). They can be stored in object properties with different names, and properties can be dynamically added or removed. Moreover, JavaScript functions are first class meaning that they can be passed as parameters. While static program analysis cannot reason about such dynamic function calls in JavaScript applications, relying on pure dynamic analysis can

also lead to an incomplete call graph because of the unexecuted functions that are part of the uncovered code at run-time. Therefore, in our approach we choose to first construct our function call graph based on dynamic information obtained during the execution, and then make use of static analysis for those functions that are remained uncovered during the execution.

Dynamic call frequencies. While the caller-callee edges in the call graph are constructed through static analysis of the application's code, the call frequency for each function is inferred dynamically from the execution traces (line 3 in Algorithm 1). The call graph also contains a mock node, called *main* function, which represents the entire code block in the global scope, i.e., global variables and statements that are not part of any particular function. The *main* node does not correspond to any function in the program. In addition, function event handlers, which are executed as a result of triggering an event, are linked to the *main* node in our dynamic call graph.

Algorithm 1: Guided Mutation Algorithm.

input : A Web application *App*, the maximum number of variable mutations $MaxVarMut$ and branch mutations $MaxBrnMut$
output: The mutated versions of application *Mutants*

1 $App \leftarrow$ INSTRUMENT(App)
begin
2 $trace \leftarrow$ COLLECTTRACE(App)
3 $\{callFrq_{f_{i,j}}, varUsgFrq_{f_i}, invars_{f_i}\}$ $\leftarrow$ GETREQUIREDINFO($trace$)
4 $l = m = 0$
5 **while** $l < MaxVarMut$ **do**
6 $\{FR(f_i)_{i=0}^{n}\}$ $\leftarrow$ FUNCTIONRANK($callGraph, callFrq_{f_{i,j}}$)
7 $mutF \leftarrow$ SELECTFUNC($(FR(f_i))_{i=0}^{n}$)
8 $\alpha \leftarrow \frac{1}{1-ReadVar_{f_i}}$
9 $candidVars_{mutF}$ $\leftarrow invars_{mutF} \cup \{v_i | varUsgFrq_{mutF}(v_i) > \alpha\}$
10 $\{pr(v_i \in candidVars_{mutF})\} \leftarrow \frac{1}{|candidVars_{mutF}|}$
11 $mutVar \leftarrow$ SELECTVAR($candidVars_{mutF}, pr(v_i)$)
12 $mutant_l \leftarrow$ VARIABLEMUTATION($mutF, mutVar, varMutOps$)
13 $l++$
end
14 $varMutants \leftarrow \bigcup mutant_{l=1}^{MaxVarMut}$
15 **while** $m < MaxBrnMut$ **do**
16 $\{pr(f_i)_{i=0}^{n}\} \leftarrow \frac{fcc(f_i) \times FR(f_i)}{\sum_{j=1}^{n} fcc(f_i) \times FR(f_i)}$
17 $mutF \leftarrow$ SELECTFUNC($(pr(f_i))_{i=0}^{n}$)
18 $mutBrn \leftarrow$ SELECTRANDOMBRN($mutF$)
19 $mutant_m \leftarrow$ BRANCHMUTATION($mutBrn, brnMutOps$)
20 $m++$
end
21 $brnMutants \leftarrow \bigcup mutant_{m=1}^{MaxBrnMut}$
22 $Mutants \leftarrow varMutants \cup brnMutants$
23 return *Mutants*
end

The functionrank metric. The original *PageRank* algorithm [16] assumes that for a given vertex, the probability of following all outgoing edges is identical, and hence all edges have the same weight. For *FunctionRank*, we instead apply edge weights proportional to the dynamic call frequencies of the functions.

Let $l(f_j, f_i)$ be the weight assigned to edge (f_j, f_i), in which function i is called by function j. We compute l by measuring the frequency of function j calling i during the execution. We assign a frequency of 1 to edges directing to unexecuted functions. The *FunctionRank* metric is calculated as

$$FR(f_i) = \sum_{j \in M(f_i)} FR(f_j) \times l(f_j, f_i), \tag{1}$$

where, $FR(f_i)$ is the *FunctionRank* value of function i, $l(f_j, f_i)$ is the frequency of calls from function j to i, and $M(f_i)$ is the set of functions that call function i.

The initial *PageRank* metric requires the sum of weights on the outgoing edges to be 1. Therefore, to solve equation (1), we need to normalize the edge weights from each function in our formula such that for each i, $\sum_{j=1}^{n} l(f_i, f_j) = 1$. To preserve the impact value of call frequencies on edges when compared globally in the graph, we normalize $l(f_i, f_j)$ over the sum of weights on all edges. Since outgoing edges from function f_i should sum to 1, an extra node called *fakeNode* is added to the graph. Note that the extra *fakeNode* is different from the mock *main* node added earlier. *fakeNode* contains an incoming edge from f_i, where

$$l(f_i, fakeNode) = 1 - \sum_{j=1}^{n} l(f_i, f_j). \tag{2}$$

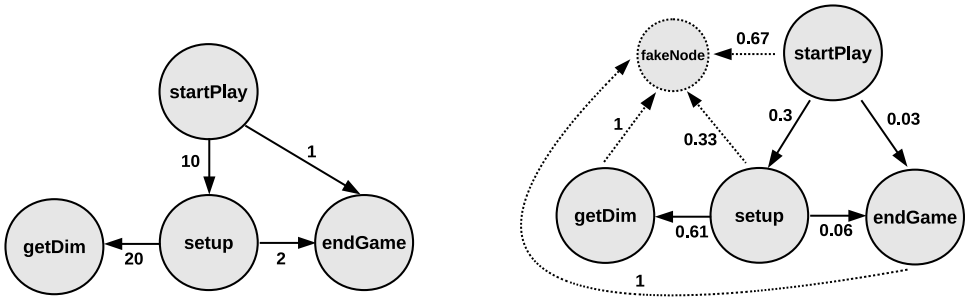
Functions with no calls are also linked to the *fakeNode* through an outgoing edge with weight 1.

A recursive function is represented by a self-loop to the recursive node in the function call graph. The original *PageRank* does not allow for self-loop nodes (i.e., a web page with a link to itself). Self-loop to a node infinitely increases its rank without changing the relative rank of the other nodes. Therefore, such nodes are disregarded in the original *PageRank* formula. However, recursive functions are inherently important as they are error-prone and difficult to debug, and they can easily propagate a fault into higher level functions. To incorporate recursive functions in our analysis, we break the self-loop to a recursive function $Recf_i$ by replacing the function with nodes f_i and f_{ci} in the function call graph. We further add an edge $l(f_i, f_{ci})$, where l is the call frequency associated with the recursive call. All functions that are called by $Recf_i$ will get an incoming edge from the added node f_{ci}. This way, all the functions called by $Recf_i$ are now linked to f_{ci} (and indirectly linked to f_i). After the *FunctionRank* metric is computed over all functions, we assign the new *FunctionRank* value of the recursive node as follows: $FR(Recf_i) = FR(f_i) + FR(f_{ci})$, where $FR(Recf_i)$ is the new *FunctionRank* value assigned to the recursive function $Recf_i$.

We initially assign equal *FunctionRank* values to all nodes in 1. The calculation of *FunctionRank* is performed recursively, until the values converge. Thus, the *FunctionRank* of a function i depends on

1) the number of functions that call i.
2) the *FunctionRank* values of the functions that call i (incoming edges).
3) the number of dynamic calls to i.

Hence, a function that is called by several functions with high *FunctionRank*s and high call frequencies receives a high *FunctionRank* itself.



(a) Call graph with call numbers.

(b) Call graph with call frequencies.

Fig. 3. Call graph of the running example.

At the end of the process, the extra function $fakeNode$ is removed and the $FunctionRank$ value of all other functions is multiplied by $\frac{1}{1-FR_{fakeNode}}$, where $FR_{fakeNode}$ is the calculated $FunctionRank$ of $fakeNode$.

Overall, our approach assigns each function a $FunctionRank$ value between 0 and 1. These values are used to rank and select functions for variable mutation (lines 6-7 in Algorithm 1). The higher the $FunctionRank$ value of a given function, the more likely it is to be selected for mutation.

Fig. 3 depicts the function call graph obtained from our running example (Fig. 1). The labels on the edges of Fig. 3a show the number of calls to each function in the graph. Fig. 3b shows the modified graph with the extra node $fakeNode$ added to compute the normalized function call frequency values. In our example, assuming that the number of `div` elements is 20 (line 3 in Fig. 1), `setup` will be called 20 times and `endGame` will be called once (lines 4 and 6). Now, assume that the number of DOM elements with the class name specified as the input to function `setup` varies each time `setup` is called (line 10) such that two elements have a length of zero and the total length of the rest is 20. Then, function `endGame` is called twice (in line 12) when the length of such elements is zero, and `getDim` is called 20 times in total (line 14). Therefore, the call frequencies of functions `setup` and `endGame` become 0.3 and 0.03 respectively when they are called by `startPlay` in lines 4 and 6. Similarly, the call frequencies of `getDim` and `endGame` become 0.61 and 0.06, respectively, when called by `setup`.

Note that if the weight on an outgoing edge of a given function is simply normalized over the sum of the weights on all the outgoing edges of that function, then the call frequencies for both `setup` and `getDim` become 0.91 when they are called by `startPlay` and `setup`, respectively. However, as shown in Fig. 3a the number of times that function `getDim` is called is twice that of `setup`. To obtain a realistic normalization, we have introduced the $fakeNode$, as shown in Fig. 3b.

Table 1 shows the computed $FunctionRank$ values, using equation (1), for each function of the running example. The values are presented as percentages. `getDim` achieves the highest $FunctionRank$ because of the relatively high values of both the incoming edge weight (where `getDim` is called by `setup` in line 14 in Fig. 1), and the $FunctionRank$ of its caller node, `setup`. These ranking values are used as probability values for selecting a function for mutation.

To illustrate the advantage of $FunctionRank$, we show the same calculation using the traditional $PageRank$ metric, i.e., without considering dynamic edge weights. As shown in Table 1, `endGame` obtains the highest ranking using $PageRank$. However, this function has not been used extensively during the application execution, and hence has only limited impact on the behaviour of the application. In contrast, when $FunctionRank$ is used, `endGame` falls to the third place, and is hence less likely to be chosen for mutation.

4.2 Ranking Functions for Branch Mutation

To rank functions for *branch* mutation, in addition to the $FunctionRank$, we take the cyclomatic complexity of the functions into account (lines 16-17 in Algorithm 1).

The cyclomatic complexity measures the number of linearly independent paths through a program's source code [17]. By using this metric, we aim to concentrate the branch mutation testing effort on the functions that are error-prone and harder to test.

We measure the cyclomatic complexity frequency of each function through static analysis of the code. Let $fcc(f_i)$ be the cyclomatic complexity frequency measured for function f_i, then $fcc(f_i) = \frac{cc(f_i)}{\sum_{j=1}^{n} cc(f_j)}$, where $cc(f_i)$ is the cyclomatic complexity of function f_i, given that the total number of functions in the application is equal to n.

We compute the probability of choosing a function f_i for branch mutation using the previously measured $Function\ Rank$ ($FR(f_i)$) as well as the cyclomatic complexity frequency ($fcc(f_i)$). Let $p(f_i)$ be the probability of selecting a function f_i for branch mutation, then

$$p(f_i) = \frac{fcc(f_i) \times FR(f_i)}{\sum_{j=1}^{n} fcc(f_j) \times FR(f_j)}, \tag{3}$$

TABLE 1
Computed $FunctionRank$ and $PageRank$ for the Running Example

Function Name	FunctionRank (%)	PageRank (%)
`getDim`	34.5	27.0
`setup`	25.0	23.0
`endGame`	21.3	34.6
`startPlay`	19.2	15.4

TABLE 2
Ranking Functions for Branch Mutation (Running Example)

Function Name	cc	fcc	Selection Probability (p)
getDim	4	0.4	0.51
setup	3	0.3	0.27
startPlay	2	0.2	0.14
endGame	1	0.1	0.08

where $fcc(f_i)$ is the cyclomatic complexity frequency measured for function f_i, and n is the total number of functions.

Table 2 shows the cyclomatic complexity, the frequency, and the function selection probability measured for each function in our example (Fig. 1). The probabilities are obtained using equation (3). As shown in the table, `getDim` achieves the highest selection probability as both its $FunctionRank$ and cyclomatic complexity are high.

5 Ranking Variables

Applying mutations on arbitrarily chosen variables may have no effect on the semantics of the program and hence lead to equivalent mutants. Thus, in addition to functions, we measure the importance of variables in terms of their impact on the behaviour of the function. We target local and global variables, as well as function parameters for mutation.

In order to avoid generating equivalent mutants, within each selected function, we need to mutate variables that are more likely to change the expected behaviour of the application (lines 7-12 in Algorithm 1). We divide such variables into two categories: (1) those that are part of the program's dynamic invariants ($invars_{mutF}$ in line 9); and (2) those with high usage frequency throughout the application's execution ($varUsgFrq_{mutF}(v_i) > \alpha$ in line 9).

5.1 Variables Involved in Dynamic Invariants

A recent study [18] showed that if a mutation violates dynamic invariants, it is very likely to be non-equivalent. This suggests that mutating variables that are involved in forming invariants affects the expected behaviour of the application with a high probability. Inspired by this finding, we infer invariants from the execution traces, as depicted in Fig. 2. We log variable value changes during run-time, and analyze the collected traces to infer stable dynamic invariants. The details of our JavaScript invariant generation technique can be found in [19]. From each obtained invariant, we identify all the variables that are involved in the invariant and mark them as potential variables for mutation.

In our running example (Fig. 1), an inferred invariant in `getDim` yields information about the inequality relation between function parameters `width` and `height`, e.g., ($width > height$). Based on this invariant, we choose `width` and `height` as potential variables for mutation.

5.2 Variables with High Usage Frequency

In addition to dynamic invariants, we consider the impact of variables on the expected behaviour based on their dynamic usage. We define the *usage frequency* of a variable as the number of times that the variable's value has been read during the execution in the scope of a given function. Let $u(v_i)$ be the usage frequency of variable v_i, then $u(v_i) = \frac{r(v_i)}{\sum_{j=1}^{n} r(v_j)}$, where $r(v_i)$ is the number of times that the value of variable v_i is read, given that the total number of variables in the scope of the function is n.

We identify the usage of a variable by identifying and measuring the frequency of a given variable being read in the following scenarios: (1) variable initialization, (2) mathematical computation, (3) condition checking in conditional statements, (4) function arguments, and (5) returned value of the function. We assign the same level of importance to all the five scenarios.

From the degree of a variable's usage frequency in the scope of a given function, we infer to what extent the behaviour of the function relies on that variable. Leveraging the collected execution traces, we compute the usage frequencies in the scope of a function. We choose variables with usage frequencies above a threshold α as potential candidates for the mutation process. We automatically compute (line 8 in Algorithm 1) this threshold for each function as

$$\alpha = \frac{1}{ReadVariables_{f(i)}}, \tag{4}$$

where $ReadVariables_{f(i)}$ is the total number of variables that at some point during the execution their value have been read within function $f(i)$.

Going back to the `getDim` function in our running example of Fig. 1, the values of function parameters `width` and `height`, as well as the local variables `w` and `h` are read just once in lines 19 and 20, when they are involved in a number of simple computations. The result of the calculation is assigned to the local variable `v`, which then is checked as a condition for the `if-else` statement. `v` is returned from the function in either line 22 or 24, depending on the outcome of the `if` statement. In this example, variable `v` has the highest usage frequency since it has been used as a condition in a conditional statement as well as the returned value of the `getDim` function.

Overall, we gather a list of potential variables for mutation, which are obtained based on the inferred dynamic invariants and their usage frequency (line 9 in Algorithm 1). Therefore, in our running example, in addition to function parameters `width` and `height`, which are part of the invariants inferred from `getDim`, the local variable `v` is also among the potential variables for the mutation process because of its high usage frequency. Note that the local variables `w` and `h` are not in the list of candidates for variable mutation as they have a low usage frequency and are not part of any dynamic invariants directly.

6 Mutation Operators

We employ generic mutation operators as well as JavaScript specific mutation operators for performing mutations.

6.1 Generic Mutation Operators

Our mutant generation technique is based on a single mutation at a time. Thus, we need to choose an appropriate

TABLE 3
Generic Mutation Operators for Variables and Function Parameters

Type	Mutation Operator
Local/Global Variable	Change the value assigned to the variable.
	Remove variable declaration/initialization.
	Change the variable type by converting `x = number` to `x = string`.
	Replace arithmetic operators (+, −, ++, −−, + =, − =, /, ∗) used for calculating and assigning a value to the selected variable.
Function Parameter	Swap parameters/arguments.
	Remove parameters/arguments.

candidate among all the potential candidates obtained from the previous ranking steps of our approach. Our overall guided mutation process includes:

- Selecting a function as described in Section 4.1 and mutating a *variable* randomly selected from the list of potential candidates obtained from the variable ranking phase (Section 5),
- Selecting a function as described in Section 4.2 and mutating a *branch statement* selected randomly (lines 16-19 in Algorithm 1).

Table 3 shows the generic mutation operators we use for mutating global variables, local variables as well as function parameters/arguments. Table 4 presents the operators we use for changing `for` loops, `while` loops, `if` and `switch-case` statements, as well as `return` statements.

6.2 JavaScript-Specific Mutation Operators

We propose the following JavaScript-specific mutation operators, based on an investigation of various online resources (see below) to understand common mistakes in JavaScript programs from the programmer's point of view. In accordance to the definition of mutation operator concept, which is representing typical programming errors, the motivation behind the presented selection of operators is to mimic typical JavaScript related programming errors. *To our knowledge, ours is the first attempt to collect and analyze these resources to formulate JavaScript mutation operators.*

TABLE 4
Generic Mutation Operators for Branch Statements

Type	Mutation Operator
Loop Statement	Change literal values in the condition (including lower/upper bound).
	Replace relational operators (<, >, <=, >=, ==, ! =, ===, ! ==).
	Replace logical operators (‖, &&).
	Swap consecutive nested `for/while`.
	Replace arithmetic operators (+, −, ++, −−, + =, − =, /, ∗).
	Replace `x++/x--` with `++x/--x` (and vice versa).
	Remove `break/continue`.
Conditional Statement	Change literal values in the condition.
	Replace relational operators (<, >, <=, >=, ==, ! =, ===, ! ==).
	Replace logical operators (‖, &&).
	Remove `else if` or `else` from the `if` statement.
	Change the condition value of `switch-case` statement.
	Remove `break` from `switch-case`.
	Replace 0/1 with `false/true` (and vice versa) in the condition.
Return Statement	Remove `return` statement.
	Replace `true` with `false` (and vice versa) in `return (true/false)`.

Adding/Removing the `var` keyword. Using `var` inside a function declares the variable in local scope, thus preventing overwriting of global variables ([9], [20], [21]). A common mistake is to forget to add `var`, or to add a redundant `var`, both of which we consider.

Removing the global search flag from `replace` . A common mistake is assuming that the string `replace` method affects all possible matches. The `replace` method only changes the first occurrence. To replace all occurrences, the global modifier needs to be set ([22], [23], [24]).

Removing the integer base argument from `parseInt` . One of the common errors with parsing integers in JavaScript is to assume that `parseInt` returns the integer value to base 10, however the second argument, which is the base, varies from 2 to 36 ([22], [25]).

Changing `setTimeout` function. The first parameter of the `setTimeout` should be a function. Consider `f` in `setTimeout(f, 3,000)` to be the function that should be executed after 3000 milliseconds. The addition of parentheses "()" to the right of the function name, i.e., `setTimeout(f(), 3000)` invokes the function immediately, which is likely not the intention of the programmer. Furthermore, in the `setTimeout` calls, when the function is invoked without passing the expected parameters, the parameter is set to `undefined` when the function is executed (same changes are applicable to `setInterval`) ([21], [26], [27]).

Replacing `undefined` with `null` . A common mistake is to check whether an object is `null`, when it is not defined. To be `null`, the object has to be defined first ([9], [22], [24]). Otherwise, an error will result.

Removing `this` keyword. JavaScript requires the programmer to explicitly state which object is being accessed, even if it is the current one. Forgetting to use `this`, may cause binding complications ([9], [22], [28]), and result in errors.

Replacing `(function()!==false)` by `(function())` . If the default value should be true, use of `(function())` should be avoided. If a function in some cases does not return a value, while the programmer expects a boolean outcome, then the returned value is `undefined`. Since `undefined` is coerced to false, the condition statement will not be satisfied. A similar issue arises when replacing `(function() === false)` with `(!function())` ([24]).

In addition, we propose a list of DOM specific mutation operators within the JavaScript code. Table 5 shows a list of DOM operators that match DOM modification patterns in either pure JavaScript language or the jQuery library. We further include two mutation operators that target the `XmlHttpRequest` type and state as shown in Table 5.

We believe these specific operators are important to be applied on their own. Hence, they are applied randomly without any ranking scheme, as they are based on errors commonly made by programmers.

TABLE 5
DOM, jQuery, and XmlHttpRequest (XHR) Operators

Type	Mutation Operator
DOM	Change the order of arguments in `insertBefore/replaceChild` methods.
	Change the name of the id/tag used in `getElementById` and `getElementByTagName` methods.
	Change the attribute name in `setAttribute`, `getAttribute`, and `removeAttribute` methods.
	Swap `innerHTML` and `innerText` properties.
JQUERY	Swap {#} and {.} sign used in selectors.
	Remove {$} sign that returns a JQUERY object.
	Change the name of the property/class/element in the following methods: `addClass`, `removeClass`, `removeAttr`, `remove`, `detach`, `attr`, `prop`, `css`.
XHR	Modify request type (Get/Post), URL, and the value of the boolean asynch argument in the `request.open` method.
	Change the integer number against which the `request.readyState/request.status` is compared with; {0, 1, 2, 3, 4} for `readyState` and {200, 404} for `status`.

7 TOOL IMPLEMENTATION: MUTANDIS

We have implemented our JavaScript mutation testing approach in a tool called MUTANDIS. MUTANDIS is written in Java and is publicly available for download.[2]

To infer JavaScript dynamic invariants, we use our recently developed tool, JSART [19]. For JavaScript code interception, we employ a proxy between the client and the server. This enables us to automatically analyze the content of HTTP responses before they reach the browser. To instrument or mutate the intercepted code, Mozilla Rhino[3] is used to parse JavaScript code to an AST, and back to the source code after the instrumentation or mutation is performed. The execution trace profiler is able to collect trace data from the instrumented application code by exercising the web application under test through one of the following methods: (1) exhaustive automatic crawling using CRAWLJAX [29], (2) the execution of existing test cases, or (3) a combination of crawling and test suite execution.

8 EMPIRICAL EVALUATION

To quantitatively assess the efficacy of our mutation testing approach, we have conducted a case study in which we address the following research questions:

RQ1 How efficient is MUTANDIS in generating non-equivalent mutants?

RQ2 How effective are *FunctionRank* and selective variable mutation in (i) generating non-equivalent mutants, and (ii) injecting non-trivial behaviour-affecting faults?

RQ3 How useful is MUTANDIS in assessing the existing test cases of a given application?

The experimental data produced by MUTANDIS is available for download.[2]

2. https://github.com/saltlab/mutandis/
3. http://www.mozilla.org/rhino/

8.1 Experimental Objects

Our study includes eight JavaScript-based objects in total. Four are game applications, namely, SameGame, Tunnel, GhostBusters, and Symbol. One is a web-based task management application called TuduList. Two, namely SimpleCart and JQUERY, are JavaScript libraries. The last application, WymEditor, is a web-based HTML editor. All the experimental objects are open-source applications. One of our main inclusion criteria was for the applications to extensively use JavaScript on the client-side. Although the game applications used in our study are small size web applications, they all extensively and in many different ways use JavaScript.

Table 6 presents each application's ID, name, and resource, as well as the static characteristics of the JavaScript code, such as JavaScript lines of code (LOC) excluding libraries, number of functions, and the cyclomatic complexity across all JavaScript functions in each application.

8.2 Experimental Setup

To run the analysis, we provide the URL of each experimental object to MUTANDIS. Note that because SimpleCart and JQUERY are both JavaScript libraries, they cannot be executed independently. However, since they come with test cases, we use them to answer RQ3.

We evaluate the efficiency of MUTANDIS in generating non-equivalent mutants (*RQ1*) for the first five applications in Table 6. We collect execution traces by instrumenting the custom JavaScript code of each application and executing the instrumented code through automated dynamic crawling. We navigate each application several times with different crawling settings. Crawling settings differ in the number of visited states, depth of crawling, and clickable element types. We inject a single fault at a time in each of these five applications using MUTANDIS. The number of injected faults for each application is 40; in total, we inject 200 faults for the five objects. We automatically generate these mutants from the following mutation categories: (1) variables, (2) branch statements, and (3) JavaScript-specific operators. We then examine each application's behaviour to determine whether the generated mutants are equivalent.

The determination of whether the mutant is equivalent is semi-automated for observable changes. An observable change is a change in the behaviour of the application which can be observed as the application is automatically executed in the browser. Note that in web applications DOM is an observable unit of the application, which is shown in the browser. We execute the same sequence of events in the mutated version as it is used in the original version of the application. The resulting observable DOM of the mutated version in the browser is visually compared against the original version. If we notice any observable change during the execution, the mutant is marked as non-equivalent. This way we can eliminate the burden of manual analysis of the applications' source code for every mutants. For non-observable changes, we manually inspect the application's source code to determine whether the mutant is equivalent.

TABLE 6
Characteristics of the Experimental Objects

App ID	Name	JS LOC	# Functions	CC	Resource
1	SameGame	206	9	37	http://crawljax.com/same-game
2	Tunnel	334	32	39	http://arcade.christianmontoya.com/tunnel
3	GhostBusters	277	27	52	http://10k.aneventapart.com/2/Uploads/657
4	Symbol	204	20	32	http://10k.aneventapart.com/2/Uploads/652
5	TuduList	2767	229	28	http://tudu.ess.ch/tudu
6	SimpleCart (library)	1702	23	168	http://simplecartjs.org
7	jQuery (library)	8371	45	37	https://github.com/jquery/jquery
8	WymEditor	3035	188	50	https://github.com/wymeditor

To make sure that changes in the applications' behaviour, from which the non-equivalency is determined, are not cosmetic changes we use the bug severity ranks used by Bugzilla, a popular bug tracking system. The description and the rank associated with each type of bug severity is shown in Table 7. We choose non-equivalent mutants from our previously generated mutants (for *RQ1*). We then analyze the output of the mutated version of the application and assign a bug score according to the ranks in Table 7.

To address *RQ2*, we measure the effectiveness of Mutandis in comparison with random-based mutation generation. Moreover, to understand the impact of applying *FunctionRank* and rank-based variable mutation in generating non-equivalent mutants as well as injecting behaviour-affecting faults we compare:

1) The proposed *FunctionRank* metric with *PageRank*.
2) Our selective variable mutation with random variable mutation.

Similar to RQ1, we use the ranks provided by Bugzilla to measure the criticality of the injected faults on the non-equivalent mutants.

Unfortunately, no test suites are available for the first five applications. Thus, to address *RQ3*, we run our tool on the SimpleCart, jQuery, and WymEditor that come with Qunit[4] test cases. We gather the required execution traces of the SimpleCart library by running its test cases, as this library has not been deployed on a publicly available application. However, to collect dynamic traces of the jQuery library, we use one of our experimental objects (SameGame), which uses jQuery as one of its JavaScript libraries. Unlike the earlier case, we include the jQuery library in the instrumentation step. We then analyze how the application uses different functionalities of the jQuery library using our approach. The execution traces of the WymEditor are collected by crawling the application. We generate 120 mutants for each of the three experimental objects. Mutated statements, which are not executed by the test suite are excluded. After injecting a fault using Mutandis, we run the test cases on the mutated version of each application. We determine the usefulness of our approach based on (1) the number of non-equivalent generated mutants, and (2) the number of non-equivalent *surviving* mutants. A non-equivalent surviving mutant is one that is neither killed nor equivalent, and is an indication of the incompleteness of the test cases. The presence of such mutants can help testers to improve the quality of their test suite. For mature test suites, we expect the number of non-equivalent surviving mutants to be low [30]. We further compare Mutandis against random mutation testing to evaluate the effect of our approach on the stubbornness of the generated mutants. Stubborn mutants are non-equivalent mutants that remain undetected by a high quality test suite [31].

4. http://docs.jquery.com/QUnit

8.3 Results

8.3.1 Generated Non-Equivalent Mutants (RQ1)

Table 8 presents our results for the number of non-equivalent mutants and the severity of the injected faults using Mutandis. For each web application, the table shows the number of mutants, number of equivalent mutants, the number of non-equivalent mutants, the percentage of equivalent mutants, and the average bug severity as well as the percentage of the severity in terms of the maximum severity level.

As shown in the table, the number of equivalent mutants varies between 2-4, which corresponds to less than 10 percent of the total number of mutants.

> *On average, the percentage of equivalent mutants generated by* Mutandis*is 7 percent, which points to its efficiency in generating non-equivalent mutants.*

We observe that more than 70 percent of these equivalent mutants generated by Mutandis originate from the branch mutation category. The reason is that in our current approach, branch expressions are essentially ranked according to the variables used in their expressions without considering whether mutating the expression changes the actual boolean outcome of the whole expression (e.g.; `if (trueVar || var){...}` where the value of `trueVar` is always `true`, and thus mutating `var` to `!var` does not affect the boolean outcome of the expression). We further notice cases in our experimental objects where the programmer writes essentially unused hard-coded branch expressions. For instance, in Tunnel, we observed a couple of `return true/false` statements at exit point of the

TABLE 7
Bug Severity Description

Bug Severity	Description	Rank
Critical	Crashes, data loss	5
Major	Major loss of functionality	4
Normal	Some loss of functionality, regular issues	3
Minor	Minor loss of functionality	2
Trivial	Cosmetic issue	1

TABLE 8
Mutants Generated by MUTANDIS

Name	# Mutants	# Equiv Mutants	# Non-Equiv Mutants	Equiv Mutants (%)	Bug Severity Rank (avg)	Bug Severity (%)
SameGame	40	2	38	5.0	3.9	78
Tunnel	40	4	36	10.0	3.8	76
GhostBusters	40	3	37	7.5	3.2	64
Symbol	40	3	37	7.5	3.9	78
TuduList	40	2	38	5.0	3.8	76
Avg.	40	2.8	37.2	7.0	3.7	74.4

functions that have high *FunctionRank* and cyclomatic complexity value. However, the returned value is never used by the caller function and hence, mutating the return boolean value as part of branch mutation generates an equivalent mutant. This is the main reason that we observe 10 percent of equivalent mutants (the highest in Table 8) for the Tunnel application. Moreover, we notice that certain types of mutation operators affect the number of equivalent mutants. For example for a number of mutations we observe that replacing >= (<=) sign with > (<) keeps the program's behaviour unchanged since either the lower/upper bound is never reached or the programmer specify extra bounds checking before returning the final value.

Fault severity of the generated mutants. The fault severity of the injected faults is also presented in Table 8. We computed the percentage of the bug severity as the ratio of the average severity rank to the maximum severity rank (which is 5). As shown in the table, the average bug severity rank across all applications is 3.72 (bug severity percentage is 74.4 percent on average). We observed only a few faults with trivial severity (e.g; cosmetic changes). We also noticed a few critical faults (3.8 percent on average), which caused the web application to terminate prematurely or unexpectedly. It is worth noting that full crashes are not that common for web applications, since web browsers typically do not stop executing the entire web application when an error occurs. The other executable parts of the application continue to run in the browser in response to user events [10]. Therefore, it is very rare for web applications to have type 5 errors, and hence the maximum severity rank is often 4.

> *More than 70 percent of the injected faults causing normal to major loss of functionality are in the top 20 percent ranked functions, showing the importance of FunctionRank in the fault seeding process.*

Moreover, we noticed that the careful choice of a variable for mutation is also as important as the function selection. For example, in the SameGame application, the `updateBoard` function is responsible for redrawing the game board each time a cell is clicked. Although `updateBoard` is ranked as an important function according to its *FunctionRank*, there are two variables within this function that have high usage frequency compared to other variables. While mutating either of these variables causes major loss of functionality, selecting the remaining ones for mutation either has no effect or only marginal effect on the application's behaviour. Furthermore, we observed that the impact of mutating variables that are part of the invariants as well as the variables with high usage frequency can severely affect the application's behaviour. This indicates that both invariants and usage frequency play a prominent role in generating faults that cause major loss of functionality, thereby justifying our choice of these two metrics for variable selection (Section 5).

8.3.2 Effectiveness of *FunctionRank* and Selective Variable Mutation (RQ2)

The results obtained from MUTANDIS, random mutation, *PageRank*, and random variable mutation in terms of the percentage of equivalent mutants and bug severity rank are shown in Figs. 4 and 5, respectively.

As shown in Fig. 4, the percentage of equivalent mutants generated by MUTANDIS is always less than or equal to the ones generated by the other three approaches. Not surprisingly, random mutation obtains the largest percentage of equivalent mutants (ranges from 7.5-15 percent). This indicates that our selective variable mutation plays a more prominent role in reducing the percentage of equivalent mutants generated by MUTANDIS.

> *On average,* MUTANDIS *reduces the number of equivalent mutants by 39 percent in comparison with random mutation generation.*

> *On average, FunctionRank and selective variable mutation reduce the number of equivalent mutants by 12 and 26 percent, respectively when compared with PageRank and random variable mutation.*

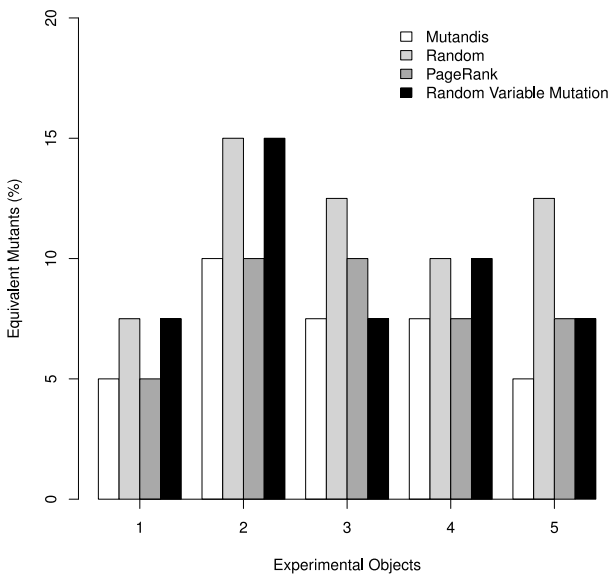


Fig. 4. Equivalent Mutants (percent) generated by MUTANDIS, random, PageRank, and random variable selection.

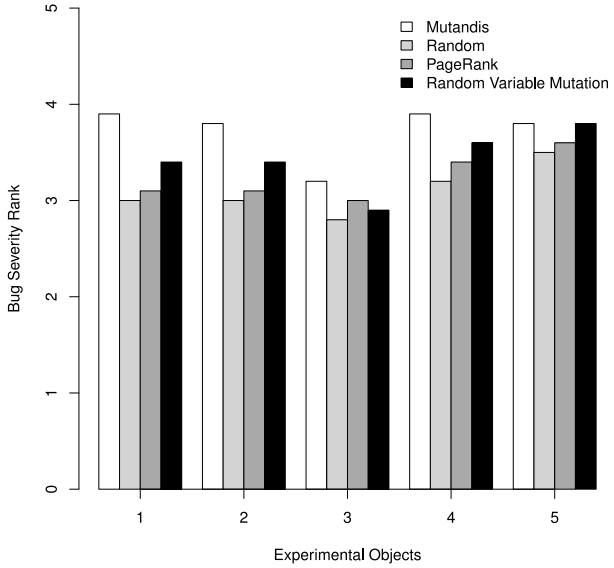


Fig. 5. Bug Severity Rankd (Avg) achieved by MUTANDIS, random, PageRank, and random variable selection.

We observed that for three applications (ID=1, 2, 4) the main reason behind the reduction in the number of equivalent mutants is the use of selective variable mutation, as by replacing selective variable mutation with random mutation, the percentage of equivalent mutants significantly increases (ranges from 33-50 percent increment) For these applications, we observed that although high rank functions are selected for mutation, modifying a non-behavioural affecting part of the selected function's code (i.e., a useless branch or variable) results in generating an equivalent mutant. Therefore, the choice of the variable or branch to mutate is very important.

However, for application with ID 3 (GhostBusters), $FunctionRank$ plays a prominent role in reducing the number of equivalent mutants. Fig. 4 shows that for this application the percentage of equivalent mutants becomes the same as MUTANDIS, when we use random variable mutation coupled with $FunctionRank$. We observed that in the aforementioned application, major variables in the program have high usage frequency. Moreover, these variables are shared among detected invariants, thus making the selection of a specific variable for mutation less effective compared to other applications. For the last application (ID 5), we observed that $FunctionRank$ and selective variable mutation are both effective in terms of generating non-equivalent mutants.

Fig. 5 compares the severity of the injected faults according to the ranks provided in Table 7. The results show that MUTANDIS achieves the highest rank among the other approaches. Our mutation generation technique increases the criticality of the injected faults by 20 percent in comparison with random mutation approach.

We observed that by replacing $FunctionRank$ with $PageRank$, the severity of the behaviour-affecting faults drops by 13 percent, which indicates that $FunctionRank$ outperforms $PageRank$ in terms of its impact on the behaviour of the application towards more critical failures.

We further noticed that using the proposed selective variable mutation increases the bug severity by 9 percent on average. While this indicates the importance of using the proposed variable mutation technique, it reveals that our rank-based function selection technique plays a more prominent role in increasing the severity degree of the injected faults compared to our variable selection strategy. For example, in application with ID 2 (Tunnel), function `updateTunnel` contains the main logic of the application, and it is among the top-ranked functions. Since `updateTunnel` is significantly used throughout the application' execution as its high rank indicates, modifications to the variables of the function affects the expected behaviour of the application, and cause the application to show more severe bugs. Our function ranking technique is able to guide the mutation process towards selecting `updateTunnel` function, and thus increasing the overall bug severity degree. On the other hand, more than 90 percent of the local and global variables used in function `updateTunnel` are involved with crucial reading and writing of properties. While mutating such important variables generates non-equivalent mutants, it will not significantly improve the criticality of the injected faults *among the non-equivalent mutants* compared to random selection of variables. This implies that our variable selection strategy plays a more prominent role in generating non-equivalent mutants rather than increasing the severity degree of the mutation.

8.3.3 Assessing Existing Test Cases (RQ3)

The results obtained from analyzing the mutants generated by MUTANDIS on the test cases of SimpleCart, JQUERY library, and WymEditor are presented in Table 9. The columns under "MUTANDIS", and "Random" present the results obtained by using our approach and random mutation generation respectively. The table shows the number of test cases, branch coverage achieved by the test suite, number of mutants, number of equivalent mutants, number of non-equivalent mutants, number of mutants detected by the test suite (killed mutants), the percentage of non-equivalent mutants and the equivalent mutants, the percentage of non-equivalent surviving mutants, and the mutation score. To compute the percentage of equivalent mutants in presence of the test suite, we follow the guidance suggested by [4], where, $Equiv(\%) = \frac{\#Equiv}{\#TotalMutants - \#Killed} \times 100$. Similarly, the percentage of non-equivalent mutants is: $Non\text{-}Equiv(\%) = \frac{\#Non\text{-}Equiv}{\#TotalMutants - \#Killed} \times 100$. The percentage of non-equivalent surviving mutants is: $\frac{\#NonEquivSurvivingMutants}{\#TotalNonEquivMutants} \times 100$.

Mutation score is used to measure the effectiveness of a test suite in terms of its ability to detect faults [32]. The mutation score is computed according to the following formula: $(\frac{K}{M-E}) \times 100$, where K is the number of killed mutants, M is the number of mutants, and E is the number of equivalent mutants.

Quality of test suites. The test suites of both JQuery and WymEditor are frequently updated in response to issues raised by the users. Both JQUery and WymEditor have around 71 percent branch coverage. This points to the overall high quality of the test cases considering how difficult it is to write unit-level test cases for JavaScript code. Note that despite the low branch coverage of SimpleCart, we gather execution traces of this application based on the available

TABLE 9
Mutation Score Computed for SimpleCart, jQuery, and WymEditor

Name	# JS Test Cases	JS Branch Coverage (%)	# TotalMutants	Mutandis							Random						
				# Equiv.	# Non-Equiv.	# Killed	Non-Equiv. (%)	Equiv. (%)	Non-Equiv. Surviving (%)	Mutation Score (%)	# Equiv.	# Non-Equiv.	# Killed	Non-Equiv. (%)	Equiv. (%)	Non-Equiv. Surviving (%)	Mutation Score (%)
SimpleCart	83	41	120	2	118	80	95	5	32	67	8	112	78	81	19	30	70
JQuery	644	73	120	3	117	106	79	21	9	90	6	114	107	54	46	6	94
WymEditor	253	71	120	6	114	97	74	26	15	85	9	111	99	57	43	11	89

test suite. Therefore, the process of mutation generation is performed according to the executed part of the application from the test suite point of view. We also observed that for the three applications in Table 9, a substantial percentage of uncovered branches are related to check for different browser settings (i.e., running the application under IE, FireFox, etc).

Surviving mutants. As shown in the table, less than 30 percent of the mutants generated by Mutandis are equivalent. SimpleCart achieves a mutation score of 67, which means there is much room for test case improvement in this application. For SimpleCart, we noticed that the number of non-equivalent, surviving mutants in the branch mutation category is more than twice the number in the variable mutation category. This shows that the test suite was not able to adequately examine several different branches in the SimpleCart library, possibly because it has a high cyclomatic complexity (Table 6). On the other hand, the QUnit test suite of the jQuery library achieves a high mutation score of over 90 percent, which indicates the high quality of the designed test cases. However, even in this case, 9 percent of the non-equivalent mutants are not detected by this test suite.

We further observed that:

> *More than 75 percent of the surviving non-equivalent mutants are in the top 30 percent of the ranked functions.*

This again points to the importance of $FunctionRank$ in test case adequacy assessment.

As far as RQ3 is concerned:

> Mutandis *is able to guide testers towards designing test cases for important portions of the code from the application's behaviour point of view.*

Stubbornness of the generated mutants. Comparing the percentage of equivalent mutants as well as surviving non-equivalent mutants generated by Mutandis to those generated by random mutation in Table 9, reveals that while our approach decreases the percentage of equivalent mutants (55 percent on average), it *does not negatively affect the stubbornness of the mutants*. To better show the effectiveness of Mutandis in decreasing the number of equivalent mutants, we compute odds ratio, which is a useful measure of effect size for categorical data [5]; the odds of non-equivalent mutants generated by approach M is computed as $odds_{Non\text{-}Equiv\ in\ M} = \frac{\#Non\text{-}Equiv_M - \#killed_M}{\#Equiv_M}$.

Regarding our results, $odds\ ratio_{Non\text{-}Equiv} = \frac{odds_{Non\text{-}Equiv\ in\ Mutandis}}{odds_{Non\text{-}Equiv\ in\ Random}} = 2.6$, which is the odds of non-equivalent mutants generated by Mutandis divided by the odds of non-equivalent mutants using random mutation generation. This indicates that the odds of non-equivalent mutants generated by Mutandis is 2.6 times higher than the random mutation strategy. We similarly measure the $odds\ ratio_{killed}$ for the number of killed mutants. The $odds\ ratio_{killed}$ of 0.98 indicates that compared with random mutation generation, our approach does not sacrifice stubbornness of the mutants. We further discuss the stubbornness of the mutants in Section 9.

9 Discussion

9.1 Stubborn Mutants

The aim of our mutation testing approach is to guide testers towards potentially error-prone parts of the code while easing the burden of handling equivalent mutants by reducing the number of such mutations. However, reducing the number of equivalent mutants might imply a decrease in the number of generated stubborn (or hard-to-kill) mutants, which are particularly useful for test adequacy assessment. Our initial results indicate that while the proposed guided approach reduces the number of equivalent mutants, it does not negatively affect the number of stubborn mutants generated. This finding is in line with a recent empirical study [31], in which no precise correlation was found between the number of equivalent mutants and stubborn mutants. However, we acknowledge that our finding is based on preliminary results and more research in this direction is needed.

In the following, we discuss different types of stubborn mutants we observed in our evaluation of Mutandis and how they can be utilized by a guided mutation generation technique to increase the number of hard-to-kill mutants. Based on our observations, stubbornness of mutants in JavaScript applications stems from (1) the type and ultimate location of the mutation operator, and (2) specific characteristics of JavaScript functions. We discuss each in the next two sections, respectively.

9.2 Type and Location of Operators

We notice that the type of the mutation operator as well as the ultimate location of the mutation affect the stubbornness of generated mutant. As far as the variable and branch mutations are concerned, the following mutations can result in stubborn mutants based on our observations:

- Variable mutations that happen in the body of conditional statements with more than one nested statement, where the conditions are involved with both variable as well as DOM related expressions. To satisfy such conditions, not only the variables should hold proper values, but also the proper structure as well as the properties of the involved DOM elements are required to be in place. This intertwined interaction limits the input space to only a few and challenging ones that are capable of satisfying the condition.
- Replacing the prefix unary operators with postfix unary operators, e.g., `++variable` to `variable++`.
- Replacing the logical operators in conditional statements when the statement contains more than one different logical operator (e.g., `if(A && B || C){...}` to `if(A && B && C){...}`).
- Swapping `true/false` in conditional statements when the statement contains more than two conditions (e.g., `if(A && B && C){...}` to `if(A && !B && C){...}`).
- Removing a parameter from a function call where the function contains more than three parameters.

As far as JavaScript specific mutation operators are concerned, we observed that the following two mutations result in more stubborn mutants compared with the rest:

- Adding a redundant `var` keyword to a global defined variable.
- Changing `setTimeout` calls such that the function is called without passing all the required parameters.

Our findings with respect to the type of mutation operator indicate that some classes of the operators tend to generate more stubborn mutants. While in our current approach we equally treat all classes, giving more priority to the hard-to-kill mutation operators would enhance the guided technique to potentially produce more stubborn mutants, which is part of our future work.

9.3 Characteristics of JavaScript Functions

A given JavaScript function can exhibit different behaviours at runtime. This is mainly due to two features of the JavaScript language.

First feature is related to the use of `this` in a function. The `this` keyword refers to the owner of the executed function in JavaScript. Depending on where the function is called from at runtime, the value of `this` can be different. It can refer to (1) a DOM element for which the executed function is currently an event handler of, (2) the global `window` object, or (3) the object of which the function is a property/method of. Let's assume function `func` is defined as follows: `var func = function () {console.log(this);};`. If `func` is set as the event handler of a DOM element `elem` (e.g.; `elem.addEventListener("click", func, false);`), when `elem` is clicked, `this` will become the DOM element `elem`. However, if function `func` is directly invoked (e.g.; `func();`), `this` becomes the window object. Therefore, the value of `this` can dynamically change within the same function as the program executes. Considering the highly dynamic nature of JavaScript applications, it is challenging for the tester to identify all such usage scenarios. Therefore, the mutation that occurs in these functions remains undetected unless the tester (1) correctly identifies all possible scopes from which the function can be invoked, and (2) associates each invocation with proper test oracles that are relevant to the value of `this`.

Second feature is function variadicity, meaning that a JavaScript function can be invoked with an arbitrary number of arguments compared to the function's static signature, which is common in web applications [33]. For example, if a function is called without passing all the expected parameters, the remaining parameters are set to `undefined`, and thus the function exhibits a different behaviour. Note that in cases where the programmer uses the same implementation of a given function for the purpose of different functionalities, the function achieves a high rank value according to our ranking mechanism since the function is executed several times from different scopes of the application. Testing the expected behaviour of all the possible functionalities is quite challenging, since invoking a particular functionality is often involved with triggering only a specific sequence of events capable of taking the application to the proper state. While the code coverage of the function is the same among different usage scenarios, the mutated statement remains unkilled unless a proper combination of test input and oracle is used. We believe that if our guided approach takes into account such particular usages of a function, which are barely exposed, it can reduce the number of equivalent mutants while increasing the number of hard-to-kill mutants, which forms part of our future work.

9.4 Threats to Validity

An external threat to the validity of our results is the limited number of web applications we use to evaluate the usefulness of our approach in assessing existing test cases (RQ4). Unfortunately, few JavaScript applications with up-to-date test suites are publicly available. Another external threat to validity is that we do not perform a quantitative comparison of our technique with other mutation techniques. However, to the best of our knowledge, there is no mutation testing tool available for JavaScript, which limits our ability to perform such comparisons. A relatively low number of generated mutants in our experiments is also a threat to validity. However, detecting equivalent mutants is a labour intensive task. For example it took us more than 4 hours to distinguish the equivalent mutants for jQuery in our study. In terms of internal threat to validity, we had to manually inspect the application's code to detect equivalent mutants. This is a time intensive task, which may be error-prone and biased towards our judgment. However, this threat is shared by other studies that attempt to detect equivalent mutants. As for the replicability of our study, Mutandis and all the experimental objects used are publicly available, making our results reproducible.

10 Related Work

A large body of research has been conducted to turn mutation testing into a practical approach. To reduce the computational cost of mutation testing, researchers have proposed three main approaches to generate a smaller subset of all possible mutants: (1) *mutant clustering* [34], which is an approach that chooses a subset of mutants using clustering

algorithms; (2) *selective mutation* [35], [36], [37], which is based on a careful selection of more effective mutation operators to generate less mutants; and (3) *higher order mutation* (HOM) testing [38], which tries to find rare but valuable higher order mutants that denote subtle faults [2].

Our guided mutation testing approach is a form of selective mutation. However, in addition to selecting a small set of effective mutation operators, our approach focuses on deciding which portions of the original code to select such that (1) the severity of injected faults impacting the application's behaviour increases, (2) the likelihood of generating equivalent mutants diminishes.

The problem of detecting equivalent mutants has been tackled by many researchers (discussed below). The main goal of all equivalent mutant detection techniques is to help the tester identify the equivalent mutants after they are generated. We, on the other hand, aim at reducing the probability of generating equivalent mutants in the first place.

According to the taxonomy suggested by Madeyski et al. [5], there are three main categories of approaches that address the problem of equivalent mutants: (1) detecting equivalent mutants, (2) avoiding equivalent mutant generation, and (3) suggesting equivalent mutants. As far as equivalent mutant detection techniques are concerned, the most effective approach is proposed by Offutt and Pan [6], [7], which uses constraint solving and path analysis. The results of their evaluation showed that the approach is able to detect on average the 45 percent of the equivalent mutants. However, these solutions are involved with considerable amount of manual effort and are error-prone. Among equivalent detection methods, program slicing has also been used in equivalent mutants detection [39]. The goal there is to guide a tester in detecting the locations that are affected by a mutant. Such equivalent mutant detection techniques are orthogonal to our approach. If a mutation has been statically proven to be equivalent, we do not need to measure its impact on the application's expected behaviour and we focus only on those that cannot be handled using static techniques. Moreover, static techniques are not able to fully address unpredictable and highly dynamic aspects of programming languages such as JavaScript.

Among avoiding equivalent mutant generation techniques, Adamopoulos et al. [8] present a co-evolutionary approach by designing a fitness function to detect and avoid possible equivalent mutants. Domínguez-Jiménez et al. [40] propose an evolutionary mutation testing technique based on a genetic algorithm to cope with the high computational cost of mutation testing by reducing the number of mutants. Their system generates a strong subset of mutants, which is composed of surviving and difficult to kill mutants. Their technique, however, cannot distinguish equivalent mutants from surviving non-equivalent mutants. Langdon et al. have applied multi-object genetic programming to generate higher order mutants [41]. An important limitation of these approaches is that the generated mutant needs to be executed against the test suite to compute its fitness function. In contrast, our approach avoids generating equivalent mutants in the first place, thereby achieving greater efficiency. Bottaci [42] presents a mutation analysis technique based on the available type information at run-time to avoid generating incompetent mutants. This approach is applicable for dynamically typed programs such as JavaScript. However, the efficiency of the technique is unclear as they do not provide any empirical evaluation of their approach. Gligoric et al. [43] conduct the first study on performing selective mutation to avoid generating equivalent mutants in concurrent code. The results show that there are important differences between the concurrent mutation and sequential mutation operators. The authors show that sequential and concurrent mutation operators are independent, and thus they propose sets of operators that can be used for mutation testing of concurrent codes. While we also make use of a small set of mutation operators, we aim to support sequential programs.

Among the equivalent mutant suggestion techniques, Schuler et al. [18] suggest possible equivalent mutants by checking invariant violations. They generate multiple mutated versions and then classify the versions based on the number of violated invariants. The system recommends testers to focus on those mutations that violate the most invariants. In a follow-up paper [4], they extend their work to assess the role of code coverage changes in detecting equivalent mutants. Kintis et al. [44] present a technique called I-EQM to dynamically isolate first order equivalent mutants. They further extend the coverage impact approach [4] to classify more killable mutants. In addition to coverage impact, the classification scheme utilizes second order mutation to assess first order mutants as killable. To generate mutants, they utilize Javalanche [4]. Our work is again different from these approaches in the sense that instead of classifying mutants, we avoid generating equivalent mutants a priori by identifying behaviour-affecting portions of the code.

Deng et al. [45] implement a version of statement deletion (SDL) mutation operator for Java within the muJava mutation system. The design of SDL operator is based on a theory that performing mutation testing using only one mutation operator can result in generating effective tests. However, the authors cannot conclude that SDL-based mutation is as effective as selective mutation, which contains a sufficient set of mutation operators from all possible operators. Therefore, they only recommend for future mutation systems to include SDL as a choice, which we have already taken into account in this paper.

Ayari et al. [46] and Fraser and Zeller [47] apply search based techniques to automatically generate test cases using mutation testing for Java applications. Harman et al. [48] propose SHOM approach which combines dynamic symbolic execution and Search based software testing to generate strongly adequate test data to kill first and higher order mutants for C programs. However, all these approaches make use of mutation testing for the purpose of test case generation, and thus to generate mutants they rely on the available mutation testing frameworks.

Zhang et al. [49] present FaMT approach which incorporates a family of techniques for prioritizing and reducing tests to reduce the time required for mutation testing. FaMT is designed based on regression test prioritization and reduction. Our approach is orthogonal to this work as our goal is to optimize the mutant generation to produce useful mutants, which can later be executed against the test suite. Our mutation generation approach can be combined with this technique to further speed up mutation testing.

Bhattacharya et al. [50] propose *NodeRank* to identify parts of code that are prone to bugs of high severity. Similar to our work, *NodeRank* uses the *PageRank* algorithm to assign a value to each node in a graph, indicating the relative importance of that node in the whole program according to the program's static call graph. The authors empirically show that such important portions of the code require more maintenance and testing effort as the program evolves. In our approach we propose a new metric, *FunctionRank*, which takes advantage of dynamic information collected at execution time for measuring the importance of a function in terms of the program's behaviour. Weighting the ranking metric with call frequencies as we do makes it more practical in web application testing, as the likelihood of exercising different parts of the application can be different. Further, to the best our knowledge, we are the first to apply such a metric to mutation testing.

11 Conclusions

Mutation testing systematically evaluates the quality of existing tests suites. However, mutation testing suffers from equivalent mutants, as well as a high computational cost associated with a large pool of generated mutants. In this paper, we proposed a guided mutation testing technique that leverages dynamic and static characteristics of the system under test to selectively mutate portions of the code that exhibit a high probability of (1) being error-prone, or (2) affecting the observable behaviour of the system, and thus being non-equivalent. Thus, our technique is able to minimize the number of generated mutants while increasing their effect on the semantics of the system. We also proposed a set of JavaScript-specific mutation operators that mimic developer mistakes in practice. We implemented our approach in an open source mutation testing tool for JavaScript, called Mutandis. The evaluation of Mutandis points to the efficacy of the approach in generating behaviour-affecting mutants.

Acknowledgments

This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme, Lockheed Martin, and Intel Corporation.

References

[1] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. Int. Conf. Softw. Eng.*, 2005, pp. 402–411.
[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2010.
[3] T. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.
[4] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Softw. Testing, Verification, Rel.*, vol. 23, no. 5, pp. 353–374, 2012.
[5] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 23–42, Sep. 2013.
[6] A. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Testing, Verification, Rel.*, vol. 7, no. 3, pp. 165–192, 1997.
[7] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proc. Int. Conf. Comput. Assurance*, 1996, pp. 224–236.
[8] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. Genetic Evol. Comput. Conf.*, 2004, pp. 1338–1349.
[9] D. Crockford, *JavaScript: The Good Parts*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2008.
[10] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2011, pp. 100–109.
[11] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 35–53, Jan./Feb. 2012.
[12] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 571–580.
[13] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 74–83.
[14] V. Basili, L. Briand, and W. Melo, "A validation of object orient design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.
[15] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 452–461.
[16] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, nos. 1–7, pp. 107–117, 1998.
[17] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[18] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proc. Int. Symp. Softw. Testing Anal.*, 2009, pp. 69–79.
[19] S. Mirshokraie and A. Mesbah, "JSART: JavaScript assertion-based regression testing," in *Proc. Int. Conf. Web Eng.*, 2012, pp. 238–252.
[20] J. Hsu. (2010). JavaScript anti-patterns [Online]. Available: http://jaysoo.ca/2010/05/06/javascript-anti-patterns/
[21] A. Osmani, *Learning JavaScript Design Patterns*. Sebastopol, CA, USA: O'Reilly Media, 2012.
[22] E. Weyl. (2010). 16 common JavaScript gotchas [Online]. Available: http://www.standardista.com/javascript/15-common-javascript-gotchas/
[23] (2010). String replace JavaScript bad design [Online]. Available: http://www.thespanner.co.uk/2010/09/27/string-replace-javascript-bad-de sign/
[24] B. L. Roy. (2005). Three common mistakes in JavaScript/EcmaScript [Online]. Available: http://weblogs.asp.net/bleroy/archive/2005/02/15/Three-common-mistakes- in-JavaScript-_2F00_-EcmaScript.aspx
[25] A. Burgess. (2011). The 11 JavaScript mistakes you are making [Online]. Available: http://net.tutsplus.com/tutorials/javascript-ajax/the-10-javascript-mis takes-youre-making/
[26] T. Ho. (2011). Premature invocation [Online]. Available: http://tobyho.com/2011/10/26/js-premature-invocation/
[27] P. Gurbani and S. Cinos. (2010). Top 13 JavaScript mistakes [Online]. Available: http://blog.tuenti.com/dev/top-13-javascript-mistakes/
[28] C. Porteneuve. (2008). Getting out of binding situations in JavaScript [Online]. Available: http://www.alistapart.com/articles/getoutbindingsituations/
[29] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.
[30] K. Jalbert and J. Bradbury, "Predicting mutation score using source code and test suite metrics," in *Proc. Int. Workshop Realizing AI Synergies Softw. Eng.*, 2012, pp. 42–46.
[31] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 919–930.
[32] M. Woodward, "Mutation testing—Its origin and evolution," *Inf. Softw. Technol.*, vol. 35, no. 3, pp. 163–169, 1993.

[33] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proc. Int. Conf. Programm. Lang. Design Implementation*, 2010, pp. 1–12.
[34] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis," in *Proc. 21st Int. Conf. Softw. Eng. Knowl. Eng.*, 2009, pp. 422–425.
[35] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Softw. Testing, Verification Rel.*, vol. 11, no. 2, pp. 113–136, 2001.
[36] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 351–360.
[37] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 435–444.
[38] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proc. 8th Int. Working Conf. Source Code Anal. Manipulation*, 2008, pp. 249–258.
[39] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Softw. Testing, Verification, Rel.*, vol. 9, no. 4, pp. 233–262, 1999.
[40] J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Inf. Softw. Technol.*, vol. 53, no. 10, pp. 1108–1123, 2011.
[41] W. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *J. Syst. Softw.*, vol. 83, no. 12, pp. 2416–2430, 2010.
[42] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *Proc. 5th Int. Workshop Mutation Anal.*, 2010, pp. 126–131.
[43] M. Gligoric, L. Z. C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 224–234.
[44] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2012, pp. 701–710.
[45] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 84–93.
[46] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1074–1081.
[47] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, Mar./Apr. 2012.
[48] M. Harman, Y. Jia, and W. Langdon, "Strong higher order mutation-based test data generation," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2011, pp. 212–222.
[49] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 235–245.
[50] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 419–429.

Shabnam Mirshokraie received the BSc degree from the Ferdowsi University of Mashhad, and the MSc degree from the Simon Fraser University. She is currently working toward the PhD degree at the University of British Columbia (UBC). Her research interests are in the area of software analysis, testing of modern web-based systems, and software maintenance. She has won a Best Paper Runner Up Award at the IEEE International Conference on Software Testing, Verification, and Validation (ICST), 2013. She is a student member of the IEEE Computer Society.

Ali Mesbah received the PhD degree in computer science from the Delft University of Technology in 2009. He is an assistant professor at the University of British Columbia (UBC) where he leads the Software Analysis and Testing (SALT) Research Lab. His main area of research is in software engineering and his research interests include software analysis and testing, web-based systems, software maintenance and evolution, fault localization and repair, program comprehension, and empirical software engineering. He received two ACM SIGSOFT Distinguished Paper Awards at the International Conference on Software Engineering (ICSE 2009 and ICSE 2014) and a Best Paper Award at the International Conference on Web Engineering (ICWE 2013). He is a member of the IEEE Computer Society.

Karthik Pattabiraman received the MS and PhD degrees from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009, respectively. After a postdoctoral year at Microsoft Research (Redmond), he joined the University of British Columbia (UBC) as an assistant professor of electrical and computer engineering. His research interests include programming languages, compilers and software engineering for building error resilient software systems. He has won a Best Paper Award at the IEEE International Conference on Dependable Systems and Networks (DSN), 2008, a Best Paper Runner Up Award at the IEEE International Conference on Software Testing (ICST), 2013 and a Distinguished Paper Award at the IEEE/ACM International Conference on Software Engineering (ICSE), 2014. He was recently the general chair for the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2013. He is a member of the IEEE Computer Society.