2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Efficient JavaScript Mutation Testing

Shabnam Mirshokraie Ali Mesbah Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
{*shabnamm, amesbah, karthikp*} *@ece.ubc.ca*

***Abstract*—Mutation testing is an effective test adequacy assessment technique. However, it suffers from two main issues. First, there is a high computational cost in executing the test suite against a potentially large pool of generated mutants. Second, there is much effort involved in filtering out equivalent mutants, which are syntactically different but semantically identical to the original program. Prior work has mainly focused on detecting equivalent mutants after the mutation generation phase, which is computationally expensive and has limited efficiency. In this paper, we propose a technique that leverages static and dynamic program analysis to guide the mutation generation process a-priori towards parts of the code that are error-prone or 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. We empirically evaluate MUTANDIS on a number of web applications to assess the efficacy of the approach.**

***Keywords*-mutation testing; JavaScript; equivalent mutant;**

I. INTRODUCTION

Mutation testing is a fault-based technique to assess and improve the quality of a test suite. The main idea is to create mutants (i.e., modified versions of the program) and check if the test suite is effective at detecting the mutants. The number of mutants detected (or 'killed') by a test suite is a measure of its effectiveness, which is also known as the adequacy score of the test suite.

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 between 10-40 percent of mutants are equivalent [4], [5]. Establishing mutant equivalence is an undecidable problem [3], and hence, the detection of equivalent mutants involves a considerable amount of manual effort.

Various attempts have been made to reduce the cost of detecting equivalent mutants, for instance through program slicing [6], [7], compiler optimization [4], constraint test data generation [5], [8], or evolutionary techniques [9], [10]. More recently, equivalent mutant detection has been investigated by assessing the impact of generated mutants on the application's expected behaviour in terms of program invariant violations [11] and code coverage [12]. While these approaches are effective in detecting equivalent mutants, they take the approach of first generating mutants and then examining the mutants for equivalence, which is computationally expensive and inefficient.

In this paper, we propose a generic mutation testing approach that guides the mutation generation process towards (1) effective mutations that affect error-prone sections of the program, (2) mutations that have a clear impact on the program's behaviour and as such are potentially non-equivalent. Our technique leverages static as well as dynamic program data to rank, select, and mutate critical behaviour-affecting portions of the program code.

Our generic mutation testing approach can be applied to any programming language. In this paper, we implement our technique for JavaScript, a loosely-typed dynamic language that is known to be error-prone [13], [14] and difficult to test [15], [16]. 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 mutation testing tool.

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, which is freely available;
- An empirical evaluation to assess the efficacy of the proposed technique using five web applications and two JavaScript libraries;

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.23

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

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

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

27 function endGame(){
28  ...
29  $(#end).css('height', getDim($('body').width(), $('↩
        body').height())+'px');
30  ...
31 }
```

Figure 1. JavaScript code of the running example.

Our results show that, on average, 93% of the mutants generated by MUTANDIS are non-equivalent. Further, the mutations have a high bug severity rank, and are capable of identifying shortcomings in existing JavaScript test suites.

II. 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.

Figure 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 DOM elements with a class name of `allCells`;
2) `setup` function is responsible for setting the `height` value of the `css` property of all the DOM elements with the given tag. 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 `end`, to indicate a game termination.

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 10 to "<", will not affect the application. This is because the number of DOM elements can never become less than zero and hence `endGame` will not be called in line 11. However, `endGame` will eventually be called in line 6 in `startPlay`, 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.

III. Our Approach

An overall overview of our mutation testing technique is depicted in Figure 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 Figure 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.

We then extract the following pieces of information from the execution traces, namely (5) variable usage frequency, (6) dynamic invariants, and (7) the dynamic call graph of the application.

Using the dynamic call graph, we (9) rank the program's functions in terms of their relative importance from the application's behaviour point of view. Further, within the highly ranked functions, our technique (10) identifies the 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 to reduce the likelihood of equivalent mutants.

In addition to variables, our technique mutates branch statements, including loops and conditional statements. Functions with high cyclomatic complexity are known to be more error-prone and challenging to test [17], [18], 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 (in 9) that also exhibit high cyclomatic complexity.

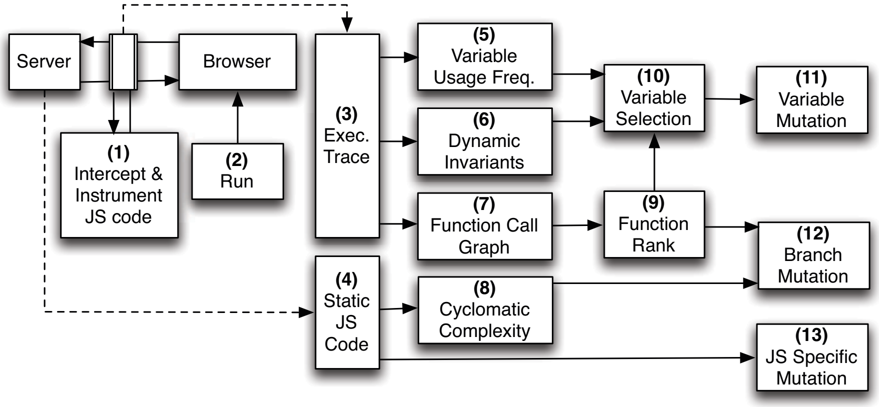

Figure 2. Overview of our mutation testing approach.

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

In the rest of this section, we describe in detail our methodology for ranking functions (boxes 8 and 9), ranking (boxes 5 and 6) and selecting (box 10) variables, and performing the actual mutations themselves (boxes 11, 12 and 13).

A. Ranking Functions for Selective Mutation

To rank functions for selective *variable* mutation, we define a new metric called $FunctionRank$ based on a function call graph inferred from the execution traces. To rank functions for *branch* mutation, we take the cyclomatic complexity of the functions into account in addition to the $FunctionRank$.

Ranking Functions for Variable Mutation. In order to rank and select functions for generating variable mutations, we propose a specialized version of the $PageRank$ metric [19], called $FunctionRank$ that takes dynamic function calls into account. This metric measures the relative importance of each function at runtime.

The original $PageRank$ algorithm 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 set edge weights based on dynamic information calls obtained from execution traces. However, the $PageRank$ formula requires that the weights on the outgoing edges sum to 1. Therefore, we need to normalize the edge weights from each function in our formula.

Let $l(f_i, f_j)$ be the weight assigned to edge (f_i, f_j), in which function j is called by function i. We compute l by measuring the number of times that function i calls j during the execution. $FunctionRank$ is calculated as:

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

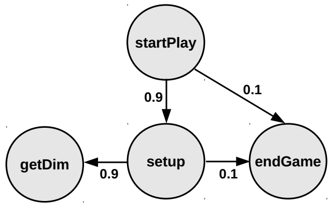

Figure 3. Call graph of the running example.

where, $FR(f_i)$ is the $FunctionRank$ value of function i, $l(f_j, f_i)$ is the the frequency of calls from function j to i, $M(f_i)$ is the set of functions that call function i, and n is the total number of functions. To solve equation 1, $l(f_i, f_j)$ is normalized such that for each i, $\sum_{j=1}^{n} l(f_i, f_j) = 1$.

Since $PageRank$ requires that the weights on the outgoing edges to sum to 1, we add outgoing edges from functions with no calls to all other functions in the program. Therefore, their $FunctionRank$ values are divided uniformly among all the functions. The calculation of $FunctionRank$ is performed iteratively, until the values converge. Thus, the $FunctionRank$ of a function i depends on:

1) Number of functions that call i;
2) $FunctionRank$ value of the functions that call i (incoming edges);
3) Number of run-time calls to i;

Intuitively, a function that is called by several functions with high $FunctionRank$ and high call frequency receives a high $FunctionRank$ itself.

Our approach assigns each function a $FunctionRank$ value between 0 and 1. These values are used to rank and select functions for variable mutation. The higher the $FunctionRank$ value of a given function, the more likely it is to be selected for mutation.

Figure 3 shows the function call graph obtained from our running example. The labels on the edges are edge weights calculated according to the function call frequency. Assuming that the number of DOM elements with class name `allCells` is 9 (line 3 in Figure 1), the call frequencies of functions `setup` and `endGame` become 0.9 and 0.1 respectively when they are called by `startPlay`

Table I
$FunctionRank$ VERSUS $PageRank$ (RUNNING EXAMPLE).

Function Name	$FunctionRank$ (%)	$PageRank$ (%)
startPlay	14.5	15.4
setup	27.5	23
endGame	18.7	34.6
getDim	39.3	27

Table II
RANKING FUNCTIONS FOR BRANCH MUTATION (RUNNING EXAMPLE).

Function Name	cc	fcc	Selection Probability (p)
startPlay	2	0.2	0.1
setup	3	0.3	0.29
endGame	1	0.1	0.06
getDim	4	0.4	0.55

in lines 4 and 6. Assume that the number of DOM elements with the tag name specified as the input to function `setup` is also 9.[1] Then, the call frequencies of `getDim` and `endGame` become 0.9 and 0.1 respectively when they are called by `setup`. Using equation 1, we are able to calculate $FunctionRank$ values associated with each of the functions in the graph. Table I shows the obtained $FunctionRank$ values for each function as percentages. `getDim` achieves the largest $FunctionRank$ because of the relatively high values of both the incoming edge weight (where `getDim` is called by `setup` in line 13 in Figure 1), and $FunctionRank$ of its caller node (`setup`). The assigned ranking values are later used as a probability of choosing a function for mutation.

To illustrate the advantage of $FunctionRank$, we show the same calculation using $PageRank$ (without considering dynamic edge weights) in Table I. As shown in the table, `endGame` obtains the highest ranking using $PageRank$, and is likely to be chosen for mutation. However, it has not been significantly used during the application execution. In contrast, by using $FunctionRank$, `endGame` falls to the third place, and is hence unlikely to be chosen for mutation.

Ranking Functions for Branch Mutation. As mentioned before, we use the cyclomatic complexity of a function in addition to its $FunctionRank$ to select functions for branch mutation. The cyclomatic complexity measures the number of linearly independent paths through a program's source code [20]. By using this metric, we aim to concentrate the branch mutation testing effort on the modules that are error-prone and hard to test in the program.

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 $FunctionRank$ ($FR(f_i)$) 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:

$$p(f_i) = \frac{fcc(f_i) \times FR(f_i)}{\sum_{j=1}^{n} fcc(f_i) \times FR(f_i)}, \qquad (2)$$

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

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

B. 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 also 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 variable mutants, we need to mutate variables within the highly ranked functions that are more likely to change the expected behaviour of the application. We divide such variables into two categories: (1) those that are part of the program's dynamic invariants; and (2) those with high usage frequency throughout the application's execution.

Variables Involved in Invariants. A recent study by Schuler et al. [11] finds 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 JavaScript invariants from the execution traces as shown in Figure 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 [21]. 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 (Figure 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.

Variables with High Usage Frequency. In addition to the invariants, we also consider the impact of variables on the expected behaviour based on their dynamic usage (See Figure 2). We define the *usage frequency* of a variable as the number of times that the variable's value has been dynamically read in the scope of a given function. Let $u(v_i)$ be the usage frequency of variable v_i: $u(v_i) = \frac{r(v_i)}{\sum_{j=1}^{n} r(v_j)}$,

[1] The number of such elements can vary each time `setup` is called. However for the sake of simplicity, we assume a fixed number of DOM elements in this example.

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. In our current approach, we give the same importance to all five scenarios.

From the degree of a variable 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. α is automatically computed for each function as $\frac{1}{ReadVariables_{f(i)}}$, where $ReadVariables_{f(i)}$ is the total number of variables that have been read within function $f(i)$.

Going back to the `getDim` function in our running example of Figure 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.

Thus, we gather a list of potential variables for mutation, which are obtained based on the inferred dynamic invariants and their usage frequency. 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 however, that the local variables `w` and `h` are not present in the list of candidates for variable mutation.

C. Mutation Process

We target variables, branch statements, and JavaScript specific operators to perform the actual mutation step. Our mutant generation technique is based on a single mutation at a time. Thus, we need to choose an appropriate candidate among all the potential candidates obtained from the previous ranking steps of our approach.

Our mutation process includes (1) mutating a random variable selected from the list of potential candidates that we obtain from the variable ranking phase, (2) mutating a random branch statement, and (3) applying a number of JavaScript specific operators. Note that the first two generic mutation operator types are applied using the ranking techniques, while the third type is applied regardless of the ranking, as these JavaScript specific operators are known to be error-prone. Hence, we believe that they are important enough to be checked on their own.

Table III
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.

Table IV
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)`.

Table III shows the mutation operators for mutating global variables, local variables as well as function parameters/arguments. Table IV shows the operators for changing `for` loops, `while` loops, `if` and `switch-case` statements, as well as `return` statements. In the rest of this section we discuss the JavaScript-specific mutation operators.

JavaScript-Specific Mutation Operators. We propose the following JavaScript-specific mutation operators, based on a study of common mistakes that JavaScript programmers make, collected from various resources (see below). *To our knowledge, ours is the first attempt to collect and analyze these resources to formulate JavaScript mutation operators.*

Adding/Removing the `var` keyword Using `var` inside a function declares the variable in local scope, thus preventing overwriting of global variables ([22], [23], [13]). 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

Table V
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`.

all occurrences, the global modifier needs to be set ([24], [25], [26]).

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 ([24], [27]).

Changing `setTimeout` function The first parameter of the `setTimeout` should be a function. Consider `f` in `setTimeout(f, 3000)` 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`) ([28], [23], [29]).

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 ([24], [26], [13]). 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 ([24], [30], [13]), 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())` ([26]).

In addition, we consider a number of DOM specific mutation operators within the JavaScript code. Table V 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 V.

IV. TOOL IMPLEMENTATION

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 [21]. For JavaScript code interception, we employ an enhanced version of Web-Scarab proxy. 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 navigation using our dynamic AJAX crawler CRAWLJAX [31], (2) the execution of existing test cases, or (3) a combination of crawling and test suite execution.

V. 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 is MUTANDIS in injecting critical behaviour-affecting faults?

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

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

A. Experimental Objects

Our study includes seven JavaScript-based objects in total. Four are game applications, namely, SameGame, Tunnel, GhostBusters, and Symbol. One is TuduList, which is a web-based task management application. The other two, SimpleCart and JQUERY, are JavaScript libraries. All the experimental objects are open-source applications.

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

[2] https://github.com/saltlab/mutandis/

[3] http://www.mozilla.org/rhino/

Table VI
CHARACTERISTICS OF THE EXPERIMENTAL OBJECTS.

App ID	Name	JS LOC	# Functions	# Local Vars	# Global Vars	CC	Resource
1	SameGame	206	9	32	5	37	http://crawljax.com/same-game
2	Tunnel	334	32	18	13	39	http://arcade.christianmontoya.com/tunnel
3	GhostBusters	277	27	75	4	52	http://10k.aneventapart.com/2/Uploads/657
4	Symbol	204	20	28	16	32	http://10k.aneventapart.com/2/Uploads/652
5	TuduList	2767	229	199	31	28	http://tudu.ess.ch/tudu
6	SimpleCart (library)	1702	23	15	10	168	http://simplecartjs.org
7	JQUERY (library)	8371	45	261	27	37	https://github.com/jquery/jquery

Table VII
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

B. 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, we are not able to use them in response to RQ1 and RQ2, as they are not independently executable. As a result, we use only the first five applications for answering RQ1 and RQ2.

We evaluate the efficiency of MUTANDIS in generating non-equivalent mutants (**RQ1**) for the first five applications in Table VI. 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 output to determine whether the generated mutants are equivalent. The determination of whether the mutant is equivalent is automated for observable changes, as we automatically execute the mutated version of the application in the browser. However, it requires manual analysis for non-observable changes.

To address **RQ2**, we use the bug severity ranks used by the Bugzilla bug tracking system, which have also been used by other researchers [32]. The description and the rank associated with each type of bug severity is shown in Table VII. 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 VII.

To address **RQ3**, we run our tool on the SimpleCart and JQUERY libraries. These two libraries come with Qunit[4] test cases. Unfortunately, test suites for the first five experimental objects are currently not available, and hence we exclude them for this study. 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. We generate 120 mutants for each library. 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 small.

C. Results

1) Generated Non-Equivalent Mutants (RQ1): Table VIII 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 and 4. On average, the percentage of equivalent mutants generated by MUTANDIS is 7%, which points to its efficiency in generating non-equivalent mutants.

We observe that more than 70% of the equivalent mutants originate from the branch mutation category. The reason is that in our current approach, we do not rank branches for mutation within the ranked/selected functions. This can result in mutating a branch that does not affect the application's behaviour. For instance, in Tunnel, we observed

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

Table VIII
RESULTS OF THE MUTANTS GENERATED BY MUTANDIS.

App ID	# Mutants	# Equiv Mutants	# Non-Equiv Mutants	Equiv Mutants (%)	Bug Severity Rank (avg)	Bug Severity (%)
1	40	2	38	5	3.6	72
2	40	4	36	10	3.7	74
3	40	3	37	7.5	3.2	64
4	40	3	37	7.5	3.7	74
5	40	2	38	5	3.8	76
Avg.	40	2.8	37.2	7	3.6	72

a couple of `return true/false` statements within the functions that have high $FunctionRank$ and cyclomatic complexity value. However, the returned value is not 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% of equivalent mutants (the highest in Table VIII) for the Tunnel application.

2) Fault Severity of the Generated Mutants (RQ2): The fault severity of the injected faults is also presented in Table VIII. 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 3.6 (bug severity percentage is 72% on average).

Based on Table VII, we see that the injected faults cause normal to major loss of functionality. We observed only a few faults with minor severity. We also noticed a few critical faults (3.8% 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 [14].

Further, for all applications, more than 70% of the faults that cause major loss of functionality are in the top 20% percent of important functions in terms of the computed $FunctionRank$, thus showing the importance of this metric 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 marginally impacts 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 III-B).

Table IX
MUTATION SCORE COMPUTED FOR THE SIMPLECART AND JQUERY LIBRARIES.

App ID	# Tests	# Mutants	# Equiv.	# Non-Equiv.	# Killed	Non-Equiv. (%)	Equiv. (%)	Non-Equiv. Surviving (%)	Mutation Score (%)
6	83	120	2	118	80	98.3	1.7	32.2	67.8
7	644	120	3	117	106	97.5	2.5	9	90.6

As far as RQ2 is concerned, our results indicate that MUTANDIS is effective in generating mutants that cause non-trivial errors in JavaScript applications.

3) Assessing Existing Test Cases (RQ3): The results obtained from analyzing the mutants generated by MUTANDIS on the test cases of SimpleCart and JQUERY library are presented in Table IX. The table shows the number of test cases, 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. The percentage of non-equivalent surviving mutants is: $\frac{\#NonEquivSurvivingMutants}{\#TotalNonEquivMuatnts}$.

Mutation score is used to measure the effectiveness of a test suite in terms of its ability to detect faults [33]. The mutation score is computed according to the following formula: $\left(\frac{k}{M-E}\right) \times 100$, where K is the number of killed mutants, M is the number of mutants, and E is the number of equivalent mutants.

As shown in the table, less than 3% of the mutants generated by MUTANDIS are equivalents. SimpleCart achieves a mutation score of 67.8, 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 cover several different branches in SimpleCart library, possibly because it has a high cyclomatic complexity (Table VI). On the other hand, the QUnit test suite of the JQUERY library achieves a high mutation score of over 90%, which indicates the high quality of the designed test cases. However, even in this case, 9% of the non-equivalent mutants are not detected by this test suite. We further observed that all the non-equivalent injected faults in SimpleCart and JQUERY that are not killed by the test suites, are in the top 30% of the important functions in terms of $FunctionRank$. This again points to the importance of $FunctionRank$ in test case generation.

As far as RQ3 is concerned, our approach is able to guide testers towards designing test cases for important portions of the code from the application's behaviour point of view.

D. 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 (RQ3). 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. 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 fully reproducible.

VI. RELATED WORK

The problem of detecting equivalent mutants has been tackled by many researchers. Offutt and Pan [5], [8] propose an approach using constraint solving. Baldwin and Sayward [34] propose a compiler optimization technique to detect equivalent mutants. The idea is that the optimization of program code produces an equivalent program, and hence a mutant can be identified as equivalent through either an optimization or deoptimization procedure. However, these approaches are able to detect only 10% of equivalent mutants [4]. Program slicing has also been used in equivalent mutants detection [6]. 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.

Adamopoulos et al. [9] present a co-evolutionary approach by designing a fitness function to detect possible equivalent mutants. Domínguez-Jiménez et al. [10] 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 [35]. 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.

Schuler et al. [11] detect 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 [12], they extend their work to assess the role of code coverage changes in detecting equivalent mutants. While the variable selection step in our approach also makes use of program invariants, our work is again different in the sense that instead of classifying mutants, we avoid generating equivalent mutants a priori by detecting behaviour-affecting portions of the code.

Bottaci [36] 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.

In recent work, Bhattacharya et al. [32] proposed $NodeRank$ to spot parts of code that are prone to bugs of high severity. $NodeRank$ 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. 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.

VII. CONCLUSIONS AND FUTURE WORK

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 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 implemented our approach in a mutation testing tool for JavaScript, called MUTANDIS. The evaluation of MUTANDIS points to the efficacy of the approach in generating non-equivalent mutants.

Our future work will include comparing different heuristics for ranking the importance of functions and variables as well as exploring ways to rank branches for mutation testing, in addition to ranking functions and variables.

Acknowledgment: This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Discovery Grants and Strategic Project Grants programmes. We also thank the Canada Foundation of Innovation (CFI) for equipment support.

REFERENCES

[1] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. Intl. Conference on Software Engineering (ICSE)*. ACM, 2005, pp. 402–411.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 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] A. Offutt and W. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification, and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[5] A. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[6] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Proc. 1st Workshop on Mutation Analysis*, 2000, pp. 5–13.

[7] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification, and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[8] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *International Conference on Computer Assurance (COMPASS)*, 1996, pp. 224–236.

[9] 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 and Evol. Comp. Conf. (GECCO)*. ACM, 2004, pp. 1338–1349.

[10] J. Domínguez-Jiménez, A. Estero-Botaro, A. Garca-Domnguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and Software Technology*, vol. 53, no. 10, pp. 1108–1123, 2011.

[11] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proc. Intl. Symp. Softw. Testing and Analysis (ISSTA)*. ACM, 2009, pp. 69–79.

[12] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in *Proc. Intl. Conf. on Softw. Testing, Verification, and Validation (ICST)*. IEEE Computer Society, 2010, pp. 45–54.

[13] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[14] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proc. of the Intl. Symp. on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2011, pp. 100–109.

[15] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2012.

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

[17] V. Basili, L. Briand, and W. Melo, "A validation of object orient design metrics as quality indicators," *IEEE Transaction on Software Engineering (TSE)*, vol. 22, no. 10, pp. 751–761, 1996.

[18] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Intl. Conf. on Softw. Eng. (ICSE)*. IEEE Computer Society, 2006, pp. 452–461.

[19] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[20] T. McCabe, "A complexity measure," *IEEE Transaction on Software Engineering (TSE)*, vol. SE-2, no. 4, pp. 308–320, 1976.

[21] S. Mirshokraie and A. Mesbah, "JSART: JavaScript assertion-based regression testing," in *Proc. Intl. Conference on Web Engineering (ICWE)*. Springer, 2012, pp. 238–252.

[22] J. Hsu, "JavaScript anti-patterns," http://jaysoo.ca/2010/05/06/javascript-anti-patterns/, 2010.

[23] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, 2012.

[24] E. Weyl, "16 common JavaScript gotchas," http://www.standardista.com/javascript/15-common-javascript-gotchas/, 2010.

[25] "String replace JavaScript bad design," http://www.thespanner.co.uk/2010/09/27/string-replace-javascript-bad-design/, 2010.

[26] B. L. Roy, "Three common mistakes in JavaScript/EcmaScript," http://weblogs.asp.net/bleroy/archive/2005/02/15/Three-common-mistakes-in-JavaScript-_2F00_-EcmaScript.aspx, 2005.

[27] A. Burgess, "The 11 JavaScript mistakes you are making," http://net.tutsplus.com/tutorials/javascript-ajax/the-10-javascript-mistakes-youre-making/, 2011.

[28] T. Ho, "Premature invocation," http://tobyho.com/2011/10/26/js-premature-invocation/, 2011.

[29] P. Gurbani and S. Cinos, "Top 13 JavaScript mistakes," http://blog.tuenti.com/dev/top-13-javascript-mistakes/, 2010.

[30] C. Porteneuve, "Getting out of binding situations in JavaScript," http://www.alistapart.com/articles/getoutbindingsituations/, 2008.

[31] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[32] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. Intl. Conf. on Softw. Eng. (ICSE)*. ACM, 2012, pp. 419–429.

[33] M. Woodward, "Mutation testing - its origin and evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993.

[34] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," Yale University, Department of Computer Science, Tech. Rep. 276, 1979.

[35] W. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416–2430, 2010.

[36] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *Proc. 5th Intl. Workshop on Mutation Analysis*, 2010, pp. 126–131.