JSTERA: Generating Regression Tests for JavaScript Applications

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

Abstract—**Developers often test their web applications using frameworks such as Selenium. Although such frameworks help to automate test execution, the test cases need to be written manually, which is tedious and inefficient. Current web test generation techniques suffer from two shortcomings; they (1) merely target the generation of event sequences, (2) either ignore the oracle problem or simplify it through generic soft oracles such as HTML validation and runtime exceptions. We present a framework to automatically generate regression test cases at two complementary levels: (1) individual JavaScript functions, (2) DOM event sequences. These test cases are strengthened by automatically generated mutation-based oracles. We empirically evaluate the implementation of our approach, called JSTERA, to assess its efficacy. The results on thirteen JavaScript-based applications show that the generated test cases achieve a coverage of 68.4% and that JSTERA can detect injected JavaScript and DOM faults with a high accuracy (100% precision, 70% recall). We also find that JSTERA outperforms an existing JavaScript test automation framework both in terms of coverage and detected faults.**

Keywords—Test generation; oracles; JavaScript; DOM

I. INTRODUCTION

JavaScript plays a prominent role in JavaScript-based web applications. To test JavaScript applications, developers often write test cases using web testing frameworks such as SELENIUM (GUI tests) and QUNIT (JavaScript unit tests). Although such frameworks help to automate test execution, the test cases still need to be written manually, which is tedious and time-consuming. Further, the event-driven and highly dynamic nature of JavaScript, as well as its runtime interaction with the Document Object Model (DOM) make it challenging to effectively write stable test cases that achieve high coverage.

Researchers have recently developed automated test generation techniques for JavaScript-based applications [1], [2], [3], [4], [5]. However, current web test generation techniques suffer from two main shortcomings, namely, they:

1) Target the generation of *event sequences*, which operate at the event-level or DOM-level to cover the state space of the application. These techniques fail to capture faults that do not propagate to an observable DOM state. As such, they potentially miss this portion of code-level JavaScript faults. In order to capture such faults, effective test generation techniques need to target the code at the JavaScript unit-level, in addition to the event-level.
2) Either ignore the oracle problem altogether or simplify it through generic *soft oracles*, such as W3C HTML validation [1], [4], or JavaScript runtime exceptions [1]. A generated test case without assertions is not useful since coverage alone is not the goal of software testing. For such generated test cases, the tester still needs to manually write many assertions, which is time and effort intensive. On the other hand, soft oracles target generic fault types and are limited in their fault finding capabilities. However, to be practically useful, unit testing requires strong oracles to determine whether the application under test executes correctly.

To address these two shortcomings, we propose an automated regression test case and oracle generation technique for JavaScript applications. Our approach operates through a three step process. First, it dynamically explores and crawls the application, using a function coverage maximization greedy algorithm, to infer a test model. Then, it generates test cases at two complementary levels, namely, DOM event and JavaScript functions. Finally, it automatically generates test oracles for both levels, through a mutation-based algorithm. To the best of our knowledge, we are the first to automatically generate unit tests at the function-level, coupled with test oracles, for JavaScript-based applications.

A preliminary version of this work appeared in a short New Ideas paper [6]. In this current paper, we present the complete technique with conceptually significant improvements, including detailed new algorithms (Algorithms 1–2), a fully-functional tool implementation, and a thorough empirical analysis on 13 JavaScript applications, providing evidence of the efficacy of the approach.

Our main contributions in this work include:

- A novel algorithm for abstracting function states to reduce the state space in unit test generation;
- A generic browser-engine independent technique to generate client-side JavaScript-level unit test and DOM-level event-based test cases;
- A DOM mutation-based method to effectively generate DOM level test oracles;
- The implementation of our approach in an open-source tool called JSTERA;
- An empirical evaluation to assess the efficacy of JSTERA. The results of our evaluation show that on average (1) the generated test suite achieves a coverage of 68.4%; (2) the test oracles generated are able to detect injected faults with 100% precision and 70% recall; (3) compared to ARTEMIS, an existing JavaScript testing framework [1], JSTERA achieves 53% better coverage; and unlike ARTEMIS that detects generic faults such as runtime exceptions and HTML validation errors, JSTERA detects faults at the JavaScript code and DOM levels.

II. RELATED WORK

Web application testing. Marchetto and Tonella [2] propose a search-based algorithm for generating event-based sequences to test Ajax applications. Mesbah et al. [7] apply dynamic analysis to construct a model of the application's state space, from which event-based test cases are automatically generated. They propose [4] generic and application-specific invariants as a form of automated soft oracles for testing AJAX applications. Our earlier work, JSART [8], automatically infers program invariants from JavaScript execution traces and uses them as regression assertions in the code. Sen et al. [9] recently proposed a record and replay framework called Jalangi. It incorporates selective record-replay as well as shadow values and shadow execution to enable writing of heavy-weight dynamic analyses. The framework is able to track generic faults such as `null` and `undefined` values as well as type inconsistencies in JavaScript. Jensen et al. [10] propose a technique to test the correctness of communication patterns between client and server in AJAX applications by incorporating server interface descriptions. They construct server interface descriptions through an inference technique that can learn communication patterns from sample data. Saxena et al. [5] combine random test generation with the use of symbolic execution for systematically exploring a JavaScript application's event space as well as its value space, for security testing. Perhaps the most closely related work to ours is ARTEMIS [1], which supports automated testing of JavaScript applications. ARTEMIS considers the event-driven execution model of a JavaScript application for feedback-directed testing. In this paper, we quantitatively compare our approach with that of ARTEMIS (Section V).

Our work is different in two main aspects from these works: (1) they all target the generation of event sequences at the DOM level, while we also generate unit tests at the JavaScript code level, which enables us to cover more and find more faults, and (2) they do not address the problem of test oracle generation and only check against soft oracles (e.g., invalid HTML). In contrast, we generate strong oracles that capture application behaviours, and can detect a much wider range of faults.

Oracle generation. There has been limited work on oracle generation for testing. Fraser et al. [11] propose μTEST, which employs a mutant-based oracle generation technique. It automatically generates unit tests for Java object-oriented classes by using a genetic algorithm to target mutations with high impact on the application's behaviour. They further identify [12] relevant pre-conditions on the test inputs and post-conditions on the outputs to ease human comprehension. Differential test case generation approaches [13], [14] are similar to mutation-based techniques in that they aim to generate test cases that show the difference between two versions of a program. However, mutation-based techniques such as ours, do not require two different versions of the application. Rather, the generated differences are in the form of controllable mutations that can be used to generate test cases capable of detecting regression faults in future versions of the program. Staats et al. [15] address the problem of selecting oracle data, which is formed as a subset of internal state variables as well as outputs for which the expected values are determined. They apply mutation testing to produce oracles and rank the inferred oracles in terms of their fault finding capability. This work is different from ours in that they merely focus on supporting the creation of test oracles by the programmer, rather than fully automating the process of test case generation. Further, (1) they do not target JavaScript; (2) in addition to the code-level mutation analysis, we propose DOM-related mutations to capture error-prone [16] dynamic interactions of JavaScript with the DOM.

```
 1 var currentDim=20;
 2 function cellClicked() {
 3   var divTag = '<div id='divElem' />';
 4   if($(this).attr('id') == 'cell0'){
 5     $('#cell0').after(divTag);
 6     $('div #divElem').click(setup);
 7   }
 8   else if($(this).attr('id') == 'cell1'){
 9     $('#cell1').after(divTag);
10     $('div #divElem').click(function(){setDim(20)});
11   }
12 }

14 function setup() {
15   setDim(10);
16   $('#startCell').click(start);
17 }

19 function setDim(dimension) {
20   var dim=($('#endCell').width() + $('#endCell').height↩
          ()))/dimension;
21   currentDim += dim;
22   $('#endCell').css('height', dim+'px');
23   return dim;
24 }

26 function start() {
27   if(currentDim > 40)
28     $(this).css('height', currentDim+'px');
29   else $(this).remove();
30 }

32 $document.ready(function() {
33   ...
34   $('#cell0').click(cellClicked);
35   $('#cell1').click(cellClicked);
36 });
```

Fig. 1. JavaScript code of the running example.

III. CHALLENGES AND MOTIVATION

In this section, we illustrate some of the challenges associated with test generation for JavaScript applications.

Figure 1 presents a snippet of a JavaScript game application that we use as a running example throughout the paper. This simple example uses the popular jQuery library [17] and contains four main JavaScript functions:

1) `cellClicked` is bound to the event-handlers of DOM elements with IDs `cell0` and `cell1` (Lines 34–35). These two DOM elements become available when the DOM is fully loaded (Line 32). Depending on the element clicked, `cellClicked` inserts a `div` element with ID `divElem` (Line 3) after the clicked element and makes it clickable by attaching either `setup` or `setDim` as its event-handler function (Lines 5–6, 9–10).
2) `setup` calls `setDim` (Line 15) to change the value of the global variable `currentDim`. It further makes an element with ID `startCell` clickable by setting its event- handler to `start` (Line 16).
3) `setDim` receives an input variable. It performs some computations to set the `height` value of the `css` property of a DOM element with ID `endCell` and the value of `currentDim` (Lines 20–22). It also returns the computed dimension.
4) `start` is called at runtime when the element with ID `startCell` is clicked (Line 16), which either updates the width dimension of the element on which it was called, or removes the element (Lines 27-29).

There are four main challenges in testing JavaScript applications.

The first challenge is that a fault may not immediately propagate into a DOM-level observable failure. For example, if the '+' sign in Line 21 is mistakenly replaced by '−', the affected result does not immediately propagate to the observable DOM state after the function exits. While this mistakenly changes the value of a global variable, `currentDim`, which is later used in `start` (Line 27), it neither affects the returned value of the `setDim` function nor the `css` value of element `endCell`. Therefore, a GUI-level event-based testing approach does not help to detect the fault in this case.

The second challenge is related to fault localization; even if the fault propagates to a future DOM state and a DOM-level test case detects it, finding the actual location of the fault is still challenging for the tester as the DOM-level test case is agnostic of the JavaScript code. However, a unit test case that targets individual functions, e.g., `setDim` in this running example, helps a tester to spot the fault, and thus easily resolve it.

The third challenge pertains to the event-driven dynamic nature of JavaScript, and its extensive interaction with the DOM resulting in many state permutations and execution paths. In the initial state of the example, clicking on `cell0` or `cell1` takes the browser to two different states as a result of the `if-else` statement in Lines 4 and 8 of the function `cellClicked`. Even in this simple example, expanding either of the resulting states has different consequences due to different functions that can be potentially triggered. Executing either `setup` or `setDim` in Lines 6 and 10 results in different execution paths, DOM states, and code coverage. It is this dynamic interaction of the JavaScript code with the DOM (and indirectly CSS) at runtime that makes it challenging to generate test cases for JavaScript applications.

The fourth important challenge in unit testing JavaScript functions that have DOM interactions, such as `setDim`, is that the DOM tree in the state expected by the function, has to be present during unit test execution. Otherwise the test will fail due to a `null` or `undefined` exception. This situation arises often in modern web applications that have many DOM interactions.

IV. Approach

Our main goal in this work is to generate client-side test cases coupled with effective test oracles, capable of detecting regression JavaScript and DOM-level faults. Further, we aim to achieve this goal as efficiently as possible. Hence, we make two design decisions. First, we assume that there is a finite amount of time available to generate test cases. Consequently we guide the test generation to maximize coverage under a given time constraint. The second decision is to minimize the number of test cases and oracles generated to only include those that are essential in detecting potential faults. Consequently, to examine the correctness of the test suite generated, the tester would only need to examine a small set of assertions, which minimizes their effort.

Our approach generates test cases and oracles at two complimentary levels:

DOM-level event-based tests consist of DOM-level event sequences and assertions to check the application's behaviour from an end-user's perspective.

Function-level unit tests consist of unit tests with assertions that verify the functionality of JavaScript code at the function level.

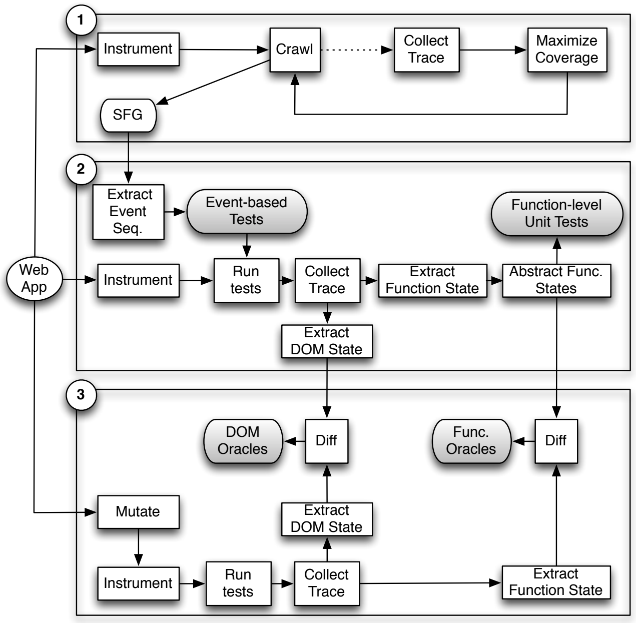

Fig. 2. Overview of our test generation approach.

An overview of the technique is depicted in Figure 2. At a high level, our approach is composed of three main steps:

1) In the first step (Section IV-A), we dynamically explore various states of a given web application, in such a way as to maximize the number of functions that are covered throughout the program execution. The output of this initial step is a state-flow graph (SFG) [4], capturing the explored dynamic DOM states and event-based transitions between them.
2) In the second step (Section IV-B), we use the inferred SFG to generate event-based test cases. We run the generated tests against an instrumented version of the application. From the execution trace obtained, we extract DOM element states as well as JavaScript function states at the entry and exit points, from which we generate function-level unit tests. To reduce the number of generated test cases to only those that are constructive, we devise a *state abstraction* algorithm that minimizes the number of states by selecting representative function states.
3) To create effective test oracles for the two test case levels, we automatically generate mutated versions of the application (Section IV-C). Assuming that the original version of the application is fault-free, the test oracles are then generated at the DOM and JavaScript code levels by comparing the states traced from the original and the mutated versions.

A. Maximizing Function Coverage

In this step, our goal is to maximize the number of functions that can be covered, while exercising the program's event space. To that end, our approach combines static and dynamic analysis to decide which state and event(s) should be selected for expansion to maximize the probability of covering uncovered JavaScript functions. While exploring the web application under test, our function coverage maximization algorithm selects a next state for exploration, which has the maximum value of the sum of the following two metrics:

1. Potential Uncovered Functions. This pertains to the total number of uncovered functions that can potentially be visited through the execution of DOM events in a given DOM state s_i. When a given function f_i is set as the event-handler of a DOM element $d \in s_i$, it makes the element a potential clickable element in s_i. This can be achieved through various patterns in web applications depending on which DOM event model level is adopted. To calculate this metric, our algorithm identifies all JavaScript functions that are directly or indirectly attached to DOM elements as event handlers, in s_i through code instrumentation and execution trace monitoring.

2. Potential Clickable Elements. The second metric, used to select a state for expansion, pertains to the number of DOM elements that can potentially become clickable elements, i.e., if the event-handlers bound to those clickables are triggered, new (uncovered) functions will be executed. To obtain this number, we statically analyze the previously obtained *potential uncovered functions* within a given state in search of such elements.

While exploring the application, the next state for expansion is selected by adding the two metrics and choosing the state with the highest sum. The procedure repeats the aforementioned steps until the designated time limit, or state space size is reached.

In the running example of Figure 1, in the initial state, clicking on elements with IDs `cell0` and `cell1` results in two different states due to an `if-else` statement in Lines 4 and 8 of `cellClicked`. Let's call the state in which a `DIV` element is located after the element with ID `cell0` as s_0, and the state in which a `DIV` element is placed after the element with ID `cell1` as s_1. If state s_0, with the clickable `cell0`, is chosen for expansion, function `setup` is called. As shown in Line 15, `setup` indirectly calls `setDim`, and thus, by expanding s_0 both of the aforementioned functions get called by a single click. Moreover, a potential clickable element is also created in Line 16, with `start` as the event-handler. Therefore, expanding s_1 results only in the execution of `setDim`, while expanding s_0 results in the execution of functions `setup`, `setDim`, and a potential execution of `start` in future states. At the end of this step, we obtain a state-flow graph of the application that can be used in the next test generation step.

B. Generating Test Cases

In the second step, our technique first extracts sequences of events from the inferred state-flow graph. These sequences of events are used in our test case generation process. We generate test cases at two complementary levels, as described below.

DOM-level event-based testing. To verify the behaviour of the application at the user interface level, each event path, taken from the initial state (`Index`) to a leaf node in the state-flow graph, is used to generate DOM event-based test cases. Each extracted path is converted into a JUNIT SELENIUM-based test case, which executes the sequence of events, starting from the initial DOM state. Going back to our running example, one possible event sequence to generate is: `$('#cell0').click`→`$('div #divElem').click`→`$('#startCell').click`.

To collect the required trace data, we capture all DOM elements and their attributes after each event in the test path is fired. This trace is later used in our DOM oracle comparison, as explained in Section IV-C.

JavaScript function-level unit testing. To generate unit tests that target JavaScript functions directly (as opposed to event-triggered function executions), we log the state of each function at their entry and exit point, during execution. To that end, we instrument the code to trace various entities. At the entry point of a given JavaScript function we collect (1) function parameters including passed variables, objects, functions, and DOM elements, (2) global variables used in the function, and (3) the current DOM structure just before the function is executed. At the exit point of the JavaScript function and before every `return` statement, we log the state of the (1) return value of the function, (2) global variables that have been accessed in that function, and (3) DOM elements accessed (read/written) in the function. At each of the above points, our instrumentation records the name, runtime type, and actual values. The dynamic type is stored because JavaScript is a dynamically typed language, meaning that the variable types cannot be determined statically. Note that complex JavaScript objects can contain circular or multiple references (e.g., in JSON format). To handle such cases, we perform a de-serialization process in which we replace such references by an object in the form of $\$ref : Path$, where $Path$ denotes a $JSONPath$ string[1] that indicates the target path of the reference.

In addition to function entry and exit points, we log information required for calling the function from the generated test cases. JavaScript functions that are accessible in the public scope are mainly defined in (1) the global scope directly (e.g., `function f(){...}`), (2) variable assignments in the global scope (e.g., `var f = function(){...}`), (3) constructor functions (e.g, `function constructor() {this. member= function(){...}}`), and (4) prototyping (e.g., `Constructor.prototype.f= function() {...}`). Functions in the first and second case are easy to call from test cases. For the third case, the constructor function is called via the `new` operator to create an object type, which can be used to access the object's properties (e.g., `container=new Constructor(); container.member();`). This allows us to access the inner function, which is a member of the `constructor` function in the above example. For the prototype case, the function can be invoked through `container.f()` from a test case.

Going back to our running example in Figure 1, at the entry point of `setDim`, we yield the value and type of both the input parameter `dimension` and global variable `currentDim`, which is accessed in the function. Similarly, at the exit point, we log the values and types of the returned variable `dim` and `currentDim`.

In addition to the values logged above, we need to capture the DOM state for functions that interact with the DOM. This is to address the fourth challenge outlined in Section III. To mitigate this problem, we capture the state of the DOM just before the function starts its execution, and include that as a *test fixture* in the generated unit test case.

In the running example, at the entry point of `setDim`, we log the `innerHTML` of the current DOM as the function contains several calls to the DOM, e.g., retrieving the element with

[1] http://goessner.net/articles/JsonPath/

Algorithm 1: Function State Abstraction

```
   input : The set of function states $st_i \in ST_f$ for a given function $f$
   output: The obtained abstracted states set $AbsStates$
   begin
1      for $st_i \in ST_f$ do
2          $L = 1$; $StSet_L \leftarrow \emptyset$
3          if $\textsc{BrnCovLns}[st_i] \neq \textsc{BrnCovLns}[StSet]_{l=1}^{L}$ then
4              $StSet_{L+1} \leftarrow st_i$
5              $L++$
6          else
7              $StSet_l \leftarrow st_i \cup StSet_l$
8          $K = L + 1$; $StSet_K \leftarrow \emptyset$
9          if $\text{DOM}\textsc{Props}[st_i] \neq \text{DOM}\textsc{Props}[StSet]_{k=L+1}^{K} \,||$
           $\textit{RetType}[st_i] \neq \textsc{RetType}[StSet]_{k=L+1}^{K}$ then
10             $StSet_{K+1} \leftarrow st_i$
11             $K++$
12         else
13             $StSet_k \leftarrow st_k \cup StSet_k$
14     while $StSet_{K+L} \neq \emptyset$ do
15         $SelectedSt \leftarrow \textsc{SelectMaxSt}(st_i | st_i \cap StateSet_{j=1}^{K+L})$
16         $AbsStates.\textsc{add}(SelectedState)$
17         $StSet_{K+L} \leftarrow StSet_{K+L} - StSet_j$
18     return $AbsStates$
```

ID `endCell` in Line 22. We further include in our execution trace the way DOM elements and their attributes are modified by the JavaScript function at runtime. The information that we log for accessed DOM elements includes the ID attribute, the XPath position of the element on the DOM tree, and all the modified attributes. Collecting this information is essential for oracle generation in the next step. We use a set collection to keep the information about DOM modifications, so that we can record the latest changes to a DOM element without any duplication within the function. For instance, we record ID as well as both `width` and `height` properties of the `endCell` element.

Once our instrumentation is carried out, we run the generated event sequences obtained from the state-flow graph. This way, we produce an execution trace that contains:

- Information required for preparing the environment for each function to be executed in a test case, including its input parameters, used global variables, and the DOM tree in a state that is expected by the function;
- Necessary entities that need to be assessed after the function is executed, including the function's output as well as the touched DOM elements and their attributes (The actual assessment process is explained in Section IV-C).

Function State Abstraction. As mentioned in Section III, the highly dynamic nature of JavaScript applications can result in a huge number of function states. Capturing all these different states can potentially hinder the technique's scalability for large applications. In addition, generating too many test cases can inversely affect test suite comprehension. We apply a function state abstraction method to minimize the number of function-level states needed for test generation.

Our abstraction method is based on classification of function (entry/exit) states according to their impact on the function's behaviour, in terms of covered branches within the function, the function's return value type, and characteristics of the accessed DOM elements.

Branch coverage: Taking different branches in a given function can change its behaviour. Thus, function entry states that result in a different covered branch should be taken into account while generating test cases. Going back to our example in Figure 1, executing either of two branches in lines 27 and 29 clearly takes the application into a different DOM state. In this example, we need to include the states of the `start` function that result in different covered branches, e.g., two different function states where the value of the global variable `currentDim` at the entry point falls into different boundaries.

Return value type: A variable's type can change in JavaScript at runtime. This can result in changes in the expected outcome of the function. Going back to our example, if `dim` is mistakenly assigned a `string` value before adding it to `currentDim` (Line 21) in function `setDim`, the returned value of the function becomes the `string` concatenation of the two values rather than the expected numerical addition.

Accessed DOM properties: DOM elements and their properties accessed in a function can be seen as entry states. Changes in such DOM entry states can affect the behaviour of the function. For example, in line 29 `this` keyword refers to the clicked DOM element of which function `start` is an event-handler. Assuming that `currentDim` ≤ 40, depending on which DOM element is clicked, by removing the element in line 29 the resulting state of the function `start` differs. Therefore, we take into consideration the DOM elements accessed by the function as well as the type of accessed DOM properties.

Algorithm 1 shows our function state abstraction algorithm. The algorithm first collects covered branches of individual functions per entry state ($\textsc{BrnCovLns}[st_i]$ in Line 3). Each function's states exhibiting same covered branches are categorized under the same set of states (Lines 4 and 7). $StSet_l$ corresponds to the set of function states, which are classified according to their covered branches, where $l = 1, ..., L$ and L is the number of current classified sets in covered branch category. Similarly, function states with the same accessed DOM characteristics as well as return value type, are put into the same set of states (Lines 10 and 13). $StSet_k$ corresponds to the set of function states, which are classified according to their DOM/return value type, where $k = 1, ..., K$ and K is the number of current classified sets in that category. After classifying each function's states into several sets, we cover each set by selecting one of its common states. The state selection step is a *set cover problem* [18], i.e., given a universe U and a family S of subsets of U, a cover is a subfamily $C \subseteq S$ of sets whose union is U. Sets to be covered in our algorithm are $StSet_{K+L}$, where $st_i \in StSet_{K+L}$. We use a common greedy algorithm for obtaining the minimum number of states that can cover all the possible sets (Lines 15-17). Finally, the abstracted list of states is returned in Line 18.

C. Generating Test Oracles

In the third step, our approach automatically generates test oracles for the two levels of test cases generated in the previous step, as depicted in the third step of Figure 2. Instead of randomly generating assertions, our oracle generation uses a mutation-based process.

Mutation testing is typically used to evaluate the quality of a test suite [19], or to generate test cases that kill mutants [11]. In our approach, we adopt mutation testing to (1) reduce the number of assertions automatically generated, (2) target critical and error-prone portions of the application. Hence, the tester

Algorithm 2: Oracle Generation

```
input : A Web application ($App$), list of event sequences obtained from SFG ($EvSeq$), maximum number of mutations ($n$)
output: Assertions for function-level ($FcAsserts$) and DOM event-level tests ($DomAsserts$)
1  $App \leftarrow$ INSTRUMENT($App$)
   begin
2      while $GenMuts < n$ do
3          foreach $EvSeq \in SFG$ do
4              $OnEvDomSt \leftarrow Trace.$GETONEVDOMST($Ev \in EvSeq$)
5              $AfterEvDomSt \leftarrow Trace.$GETAFTEREVDOMST($Ev \in EvSeq$)
6              $AccdDomProps \leftarrow$ GETACCDDOMNDS($OnEvDomSt$)
7              $EquivalentDomMut \leftarrow true$
8              while $EquivalentDomMut$ do
9                  $MutDom \leftarrow$ MUTATEDOM($AccdDomProps, OnEvDomSt$)
10                 $ChangedSt \leftarrow EvSeq.$EXECEVENT($MutDom$)
11                 $Diff_{ChangedSt,AfterEvDomSt} \leftarrow$ DIFF($ChangedSt$, $AfterEvDomSt$)
12                 if $Diff_{ChangedSt,AfterEvDomSt} \neq \emptyset$ then
13                     $EquivalentDomMut \leftarrow false$
14                     $DomAssert_i = Diff_{ChangedSt,AfterEvDomSt}$
15                     $DomAsserts_{Ev,AfterEvDomSt} = \bigcap DomAssert_i$
16             $AbsFcSts \leftarrow Trace.$GETABSFCSTS()
17             $EquivalentCodeMut \leftarrow true$
18             while $EquivalentCodeMut$ do
19                 $MutApp \leftarrow$ MUTATEJSCODE($App$)
20                 $MutFcSts \leftarrow EvSeq.$EXECEVENT($MutApp$)
21                 foreach $FcEntry \in AbsFcSts.$GETFCENTRIES do
22                     $FcExit \leftarrow AbsFcSts.$GETFCEXIT($FcEntry$)
23                     $MutFcExit \leftarrow MutFcSts.$GETMUTFCEXIT($FcEntry$)
24                     $Diff_{FcExit,MutFcExit} \leftarrow$ DIFF($FcExit$, $MutFcExit$)
25                     if $Diff_{FcExit,MutFcExit} \neq \emptyset$ then
26                         $EquivalentCodeMut \leftarrow false$
27                         $FcAssert_i = \bigcap Diff_{FcExit,MutFcExit}$
28                         $FcAsserts_{FcEntry} = \bigcup FcAssert_i$
29     return $\{FcAsserts, DOMAsserts\}$
```

would only need to examine a small set of effective assertions to verify the correctness of the generated oracles. Algorithm 2 shows our algorithm for generating test oracles. At a high level, the technique iteratively executes the following steps:

1) A mutant is created by injecting a single fault into the original version of the web application (Line 9 and 19 in Algorithm 2 for DOM mutation and code-level mutation, respectively),
2) Related entry/exit program states at the DOM and JavaScript function levels of the mutant and the original version are captured. $OnEvDomSt$ in Line 4 is the original DOM state on which the event Ev is triggered, $AfterEvDomSt$ in line 5 is the observed DOM state after the event is triggered, $MutDom$ in line 9 is the mutated DOM, and $ChangedSt$ in line 10 is the corresponding affected state for DOM mutations. $FcExit$ in Line 22 is the exit state of the function in the original application and $MutFcExit$ in line 23 is the corresponding exit state for that function after the application is mutated for function-level mutations.
3) Relevant observed state differences at each level are detected and abstracted into test oracles (DIFF in Line 11 and 24 for DOM and function-level oracles, respectively),
4) The generated assertions (Lines 15 and 28) are injected into the corresponding test cases.

DOM-level event-based test oracles. After an event is triggered in the generated SELENIUM test case, the resulting DOM state needs to be compared against the expected structure. One naive approach would be to compare the DOM tree in its entirety, after the event execution. Not only is this approach inefficient, it results in brittle test-cases, i.e., the smallest update on the user interface can break the test case. We propose an alternative approach that utilizes *DOM mutation testing* to detect and selectively compare only those DOM elements and attributes that are affected by an injected fault at the DOM-level of the application. Our DOM mutations target only the elements that have been accessed (read/written) during execution, and thus have a larger impact on the application's behaviour. To select proper DOM elements for mutation, we instrument JavaScript functions that interact with the DOM, i.e., code that either accesses or modifies DOM elements.

We execute the instrumented application by running the generated SELENIUM test cases and record each accessed DOM element, its attributes, the triggered event on the DOM state, and the DOM state after the event is triggered (GETONEVDOMST in line 4, GETAFTEREVDOMST in line 5, and GETACCDDOMNDS in line 6 to retrieve the original DOM state, DOM state after event Ev is triggered, and the accessed DOM properties as event Ev is triggered, respectively, in Algorithm 2). To perform the actual mutation, as the application is re-executed using the same sequence of events, we mutate the recorded DOM elements, one at a time, before the corresponding event is fired. MUTATEDOM in line 9 mutates the DOM elements, and $EvSeq$.EXECEVENT in line 10 executes the event sequence on the mutated DOM. The mutation operators include (1) deleting a DOM element, and (2) changing the attribute, accessed during the original execution. As we mutate the DOM, we collect the current state of DOM elements and attributes.

Figure 3 shows part of a DOM-level test case generated for the running example. Going back to our running example, as a result of clicking on `$('div #divElem')` in our previously obtained event sequence (`$('#cell0').click`→`$('div #divElem').click`→`$('#startCell')`), the `height` and `width` properties of DOM element with ID `endCell`, and the DOM element with ID `startCell` are accessed. One possible DOM mutation is altering the `width` value of the `endCell` element before click on `$('div #divElem')` happens. We log the consequences of this modification after the click event on `$('div #divElem')` as well as the remaining events. This mutation affects the `height` property of DOM element with ID `endCell` in the resulting DOM state from clicking on `$('div #divElem')`. Line 6 in Figure 3 shows the corresponding assertion. Furthermore, Assuming that the DOM mutation makes `currentDim`$\leq$ 40 in line 27, after click on element `#startCell` happens, the element is removed and no longer exists in the resulting DOM state. The generated assertion is shown in line 10 of Figure 3.

Hence, we obtain two sets of execution traces that contain information about the state of DOM elements for each fired event in the original and mutated application. By comparing these two traces (DIFF in line 11 in Algorithm 2), we identify all changed DOM elements and generate assertions for these elements. Note that any changes detected by the

```
@Test
public void testCase1(){
  WebElement divElem=driver.findElements(By.id("divElem"));
  divElem.click();
  int endCellHeight=driver.findElements(By.id("endCell")).getSize().hight;
  assertEquals(endCellHeight, 30);
  WebElement startCell=driver.findElements(By.id("startCell"));
  startCell.click();
  boolean exists=driver.findElements(By.id("startCell")).size!=0;
  assertTrue(exists);
  int startCellHeight=driver.findElements(By.id("startCell")).getSize().height;
  assertEquals(startCellHeight, 50);
}
```

Fig. 3. Generated SELENIUM test case.

DIFF operator (line 12 in Algorithm 2) is an indication that the corresponding DOM mutation is *not equivalent* (line 13); if no change is detected, another DOM mutation is generated. We automatically place the generated assertion immediately after the corresponding line of code that executed the event, in the generated event-based (SELENIUM) test case. $DomAsserts_{Ev,AfterEvDomSt}$ in line 15 contains all DOM assertions for the state $AfterEvDOMSt$ and the triggered event Ev.

Function-level test oracles. To seed code level faults, we use our recently developed JavaScript mutation testing tool, MUTANDIS [20]. Mutations generated by MUTANDIS are selected through a *function rank* metric, which ranks functions in terms of their relative importance from the application's behaviour point of view. The mutation operators are chosen from a list of common operators, such as changing the value of a variable or modifying a conditional statement. Once a mutant is produced (MUTATEJSCODE in line 19), it is automatically instrumented. We collect a new execution trace from the mutated program by executing the same sequence of events that was used on the original version of the application. This way, the state of each JavaScript function is extracted at its entry and exit points. $AbsFcSts$.GETFCENTRIES in line 21 retrieves the function's entries from the abstracted function's states. GETFCEXIT in line 22 , and GETMUTFCEXIT in line 23 retrieve the corresponding function's exit state in the original and mutated application. This process is similar to the function state extraction algorithm explained in Section IV-B.

After the execution traces are collected for all the generated mutants, we generate function-level test oracles by comparing the execution trace of the original application with the traces we obtained from the modified versions (DIFF in line 24 in Algorithm 2). If the DIFF operator detects no changes (line 25 of the algorithm), an equivalent mutant is detected, and thus another mutant will be generated.

Our function-level oracle generation targets *postcondition assertions*. Such postcondition assertions can be used to examine the expected behaviour of a given function after it is executed in a unit test case. Our technique generates postcondition assertions for all functions that exhibit a *different exit-point state* but the *same entry-point state*, in the mutated execution traces. $FcAssert_i$ in line 27 contains all such post condition assertions. Due to the dynamic and asynchronous behaviour of JavaScript applications, a function with the same entry state can exhibit different outputs when executed multiple times. In this case, we need to combine assertions to make sure that the generated test cases do not mistakenly fail. $FcAsserts_{FcEntry}$

```
test("Testing setDim",4,function(){
  var fixture = $("#qunit-fixture");
  fixture.append("<button id=\"cell0\"> <div id=\"divElem\"/> </button> <div id=\"endCell\" style=\"height:200px;width:100px;\"/>");
  var currentDim=20;
  var result= setDim(10);
  equal(result, 30);
  equal(currentDim, 50);
  ok($(#endCell).length > 0));
  equal($(#endCell).css('height'), 30); });
```

Fig. 4. Generated QUNIT test case.

in line 28 contains the union of function assertions generated for the same entry but different outputs during multiple executions. Let's consider a function f with an entry state $entry$ in the original version of the application (A), with two different exit states $exit_1$ and $exit_2$. If in the mutated version of the application (A_m), f exhibits an exit state $exit_m$ that is different from both $exit_1$ and $exit_2$, then we combine the resulting assertions as follows: assert1($exit_1$,$expRes_1$)$\|$assert2($exit_2$,$expRes_2$), where the expected values $expRes_1$ and $expRes_2$ are obtained from the execution trace of A.

Each assertion for a function contains (1) the function's returned value, (2) the used global variables in that function, and/or (3) the accessed DOM element in that function. Each assertion is coupled with the expected value obtained from the execution trace of the original version.

The generated assertions that target variables, compare the value as well as the runtime type against the expected ones. An oracle that targets a DOM element, first checks the existence of that DOM element. If the element exists, it checks the attributes of the element by comparing them against the observed values in the original execution trace. Assuming that `width` and `height` are 100 and 200 accordingly in Figure 1, and '+' sign is mutated to '-' in line 20 of the running example in Figure 1, the mutation affects the global variable `currentDim`, `height` property of element with ID `endCell`, and the returned value of the function `setDim`. Figure 4 shows a QUNIT test case for `setDim` function according to this mutation with the generated assertions.

D. Tool Implementation

We have implemented our JavaScript test and oracle generation approach in an automated tool called JSTERA. The tool is written in Java and is publicly available for download [21]. Our implementation requires no browser modifications, and is hence portable. For JavaScript code interception, we use a web proxy, which enables us to automatically instrument JavaScript code before it reaches the browser. The crawler for JSTERA extends and builds on top of the event-based crawler, CRAWLJAX [7], with random input generation enabled for form inputs. As mentioned before, to mutate JavaScript code, we integrate our recently developed mutation testing tool, MUTANDIS [20]. The upper-bound for the number of mutations can be specified by the user. However, the default is 50 for code-level and 20 for DOM-level mutations. We observed that these default numbers provide a balanced trade-off between oracle generation time, and the fault finding capability of the tool. DOM-level test cases are generated in a JUNIT format that uses SELENIUM (WebDriver) APIs to fire events on the application's DOM inside the browser. JavaScript function-level tests are generated in the QUNIT unit testing framework [22], capable of testing any generic JavaScript code.

TABLE I. CHARACTERISTICS OF THE EXPERIMENTAL OBJECTS.

ID	Name	LOC	Resource
1	SameGame	206	crawljax.com/same-game/
2	Tunnel	334	arcade.christianmontoya.com/tunnel/
3	GhostBusters	282	10k.aneventapart.com/2/Uploads/657/
4	Peg	509	www.cccontheweb.org/peggame.htm
5	BunnyHunt	580	http://www.themaninblue.com/experiment/BunnyHunt/
6	AjaxTabs	592	https://github.com/amazingSurge/jquery-tabs/
7	NarrowDesign	1005	http://www.narrowdesign.com
8	JointLondon	1211	http://www.jointlondon.com
9	FractalViewer	1245	onecm.com/projects/canopy/
10	SimpleCart	1900	https://github.com/wojodesign/simplecart-js/
11	WymEditor	3035	http://www.wymeditor.org
12	TuduList	1963	http://tudu.ess.ch/tudu
13	TinyMCE	26908	http://www.tinymce.com

V. EMPIRICAL EVALUATION

To quantitatively assess the efficacy of our test generation approach, we have conducted an empirical study, in which we address the following research questions:

RQ1 How effective is JSTERA in generating test cases with high coverage?
RQ2 How capable is JSTERA of generating test oracles that detect regression faults?
RQ3 How does JSTERA compare to existing automated JavaScript testing frameworks?

JSTERA and all our experimental data produced are available for download [21].

A. Objects

Our study includes thirteen JavaScript-based applications in total. Table I presents each application's ID, name, lines of custom JavaScript code (LOC, excluding JavaScript libraries) and resource. The first five are web-based games. AjaxTabs is a JQUERY plugin for creating tabs. NarrowDesign and JointLondon are websites. FractalViewer is a fractal tree zoom application. SimpleCart is a shopping cart library, WymEditor is a web-based HTML editor, TuduList is a web-based task management application, and TinyMCE is a JavaScript based WYSIWYG editor control. The applications range from 206 to 27K lines of JavaScript code.

The experimental objects are open-source and cover different application types. All the applications are interactive in nature and extensively use JavaScript on the client-side.

B. Setup

To address our research questions, we provide the URL of each experimental object to JSTERA. Test cases are then automatically generated by JSTERA. We give JSTERA 10 minutes in total for each application. 5 minutes of the total time is designated for the dynamic exploration step.

Test Case Generation (RQ1). To measure client-side code coverage, we use JSCOVER [23], an open-source tool for measuring JavaScript code coverage. We report the average results over five runs. In addition, we assess each step in our approach separately as follows: (1) compare the statement coverage achieved by our function coverage maximization with a method that chooses the next state/event for the expansion uniformly at random, (2) assess the efficacy of our function state abstraction method (Algorithm 1), and (3) evaluate the effectiveness of applying mutation techniques (Algorithm 2) to reduce the number of assertions generated.

Test Oracles (RQ2). To evaluate the fault finding capability of JSTERA (RQ2), we simulate web application faults by automatically seeding each application with 50 random faults. We automatically pick a random program point and seed a fault at that point according to our fault category. While mutations used for oracle generation have been selectively generated (as discussed in Section IV-C), mutations used for the purpose of evaluation are randomly generated from the entire application. Note that if the mutation used for the purpose of evaluation and the mutation used for generating oracles happen to be the same, we remove the mutant from the evaluation set. Next we run the whole generated test suite (including both function-level and event-based test cases) on the faulty version of the application. The fault is considered detected if an assertion generated by JSTERA fails and our manual examination confirms that the failed assertion is detecting the seeded fault. We measure the precision and recall as follows:

Precision is the rate of injected faults found by the tool that are correct: $\frac{TP}{TP+FP}$
Recall is the rate of correct injected faults that the tool finds: $\frac{TP}{TP+FN}$

where *TP* (true positives), *FP* (false positives), and *FN* (false negatives) respectively represent the number of faults that are correctly detected, falsely reported, and missed.

Comparison (RQ3). To assess how JSTERA performs with respect to existing JavaScript test automation tools, we compare its coverage and fault finding capability to that of ARTEMIS [1]. Similar to JSTERA, we give ARTEMIS 10 minutes in total for each application; we observed no improvements in the results obtained from running ARTEMIS for longer periods of time. We run ARTEMIS from the command line by setting the iteration option to 100 and enabling the coverage priority strategy, as described in [1]. Similarly, JSCover is used to measure the coverage of ARTEMIS (over 5 runs). We use the output provided by ARTEMIS to determine if the seeded mutations are detected by the tool, by following the same procedure as described above for JSTERA.

C. Results

Test Case Generation (RQ1). Figure 5 depicts the statement coverage achieved by JSTERA for each application. The results show that the test cases generated by JSTERA achieve a coverage of 68.4% on average, ranging from 41% (ID 12) up to 99% (ID 1). We investigated why JSTERA has low coverage for some of the applications. For instance, we observed that in JointLondon (ID 7), the application contains JavaScript functions that are browser/device specific, i.e., they are exclusively executed in Internet Explorer, or iDevices. As a result, we are unable to cover them using JSTERA. We also noticed that some applications required more time to achieve higher statement coverage (e.g., in NarrowDesign ID 8), or they have a large DOM state space (e.g., BunnyHunt ID 5) and hence JSTERA is only able to cover a portion of these applications in the limited time it had available.

Table II columns under "Coverage" present JavaScript statement coverage achieved by our function coverage maximization algorithm versus a random strategy. The results show a 9% improvement on average, for our algorithm, across all

TABLE II. RESULTS OF OUR FUNCTION COVERAGE MAXIMIZATION, FUNCTION STATE ABSTRACTION, AND MUTATION-BASED ORACLE GENERATION ALGORITHMS.

	Coverage		**State Abstraction**			**Oracles**	
App ID	**Fun. cov. maximize (%)**	**Random exploration (%)**	**#Func.States w/o abstraction**	**#Func.States with abstraction**	**Func.State Reduction (%)**	**#Assertions w/o mutation**	**#Assertions with mutation**
1	99	80	447	33	93	5101	136
2	78	78	828	21	97	23212	81
3	90	66	422	14	96	3520	45
4	75	75	43	19	56	1232	109
5	49	45	534	23	95	150	79
6	78	75	797	30	96	1648	125
7	63	58	1653	54	97	198202	342
8	56	50	32	18	43	78	51
9	82	82	1509	49	97	65403	253
10	71	69	71	23	67	6584	96
11	56	54	1383	131	90	2530	318
12	41	38	1530	62	96	3521	184
13	51	47	1401	152	89	2481	335
AVG	68.4	62.8	-	-	85.5	-	-

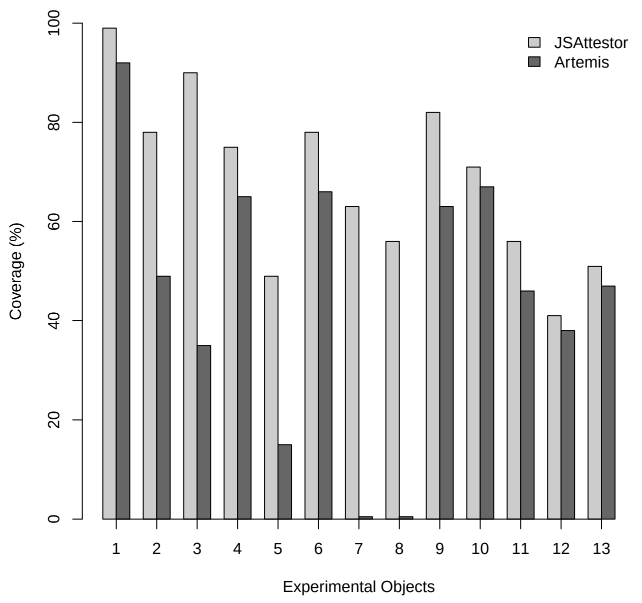

Fig. 5. Statement coverage achieved.

the applications. We observed that our technique achieves the highest improvement when there are many dynamically generated clickable DOM elements in the application, for example, GhostBusters (ID 3).

The columns under "State Abstract" in Table II present the number of function states before and after applying our function state abstraction algorithm. The results show that the abstraction strategy reduces function states by 85.5% on average. NarrowDesign (ID 7) and FractalViewer (ID 9) benefit the most by a 97% state reduction rate. Note that despite this huge reduction, our state abstraction does not adversely influence the coverage as we include at least one function state from each of the covered branch sets as described in Section IV-B.

The last two columns of Table II, under "Oracles", present

TABLE III. FAULT DETECTION.

		JSTERA						**ARTEMIS**	
App ID	**# Injected Faults**	**#FN**	**#FP**	**#TP**	**Precision (%)**	**Recall (%)**	**By func-level tests (%)**	**Precision (%)**	**Recall (%)**
1	50	0	0	50	100	100	30	100	20
2	50	9	0	41	100	82	73	100	12
3	50	4	0	46	100	92	17	100	8
4	50	15	0	35	100	70	28	100	22
5	50	26	0	24	100	48	25	100	0
6	50	9	0	41	100	82	15	100	16
7	50	17	0	33	100	66	24	100	0
8	50	23	0	27	100	54	26	100	0
9	50	6	0	44	100	88	41	100	24
10	50	16	0	34	100	68	65	100	8
11	50	21	0	29	100	58	27	100	6
12	50	26	0	24	100	48	17	100	22
13	50	23	0	27	100	54	26	100	28
AVG	-	15	0	35	100	70	32	100	12.8

the number of assertions obtained by capturing the whole application's state, without any mutations, and with our mutation-based oracle generation algorithm respectively. The results show that the number of assertions is decreased by 86.5% on average due to our algorithm. We observe the most significant reduction of assertions for JointLondon (ID 7) from more than 198000 to 342.

Fault finding capability (RQ2). Table III presents the results on the fault finding capabilities of JSTERA. The table shows the total number of injected faults, the number of false negatives, false positives, true positives, and the precision and recall of JSTERA.

JSTERA achieves 100% precision, meaning that all the detected faults reported by JSTERA are real faults. *In other words, there are no false-positives.* This is because the assertions generated by JSTERA are all stable i.e., they do not change from one run to another. However, the recall of JSTERA is 70% on average, and ranges from 48 to 100%. This is due to false negatives, i.e., missed faults by JSTERA, which occur when the injected fault falls is either in the uncovered region of the application, or is not properly captured by the generated oracles.

The table also shows that on average 32% percent of the injected faults (ranges from 15–73%) are detected by function-level test cases, but not by our DOM event-based test cases. This shows that a considerable number of faults do not propagate to observable DOM states, and thus cannot be captured by DOM-level event-based tests. For example in SimpleCart application (ID 10), if we mutate the mathematical operation that is responsible for computing the total amount of purchased items, the resulting error is not captured by event-based tests as the fault involves internal computations only. However, the fault is detected by a function-level test that directly checks the returned value of the function. This points to the importance of incorporating function-level tests in addition to event-based tests for JavaScript web applications. We also observed that even when an event-based test case detects a JavaScript fault, localizing the error to the corresponding JavaScript code can be quite challenging. However, function-level tests pinpoint the corresponding function when an assertion fails, making it easier to localize the fault.

Comparison (RQ3). Figure 5 shows the code coverage achieved by both JSTERA and ARTEMIS on the experimental objects running for the same amount of time, i.e., 10 minutes. The test cases generated by JSTERA achieve 68.4% coverage on average (ranging from 41–99%), while ARTEMIS achieves only 44.8% coverage on average (ranging from 0–92%). Overall, the test cases generated by JSTERA achieve 53% coverage than ARTEMIS, which points to the effectiveness of JSTERA in generating high coverage test cases. Further, as can be seen in the bar plot of Figure 5, for all the applications, the test cases generated by JSTERA achieve higher coverage than those generated by ARTEMIS. This increase was more than 226% in the case of Bunnyhunt (ID 5). For two of the applications, NarrowDesign (ID 7) and JointLondon (id 8), ARTEMIS was not able to complete the testing task within the allocated time of ten minutes. Thus we let ARTEMIS run for an extra 10 minutes for these applications (i.e., 20 minutes in total). Even then, neither application completes under ARTEMIS.

Table III shows the precision and recall achieved by JSTERA and ARTEMIS. With respect to fault finding capability, unlike ARTEMIS that detects only generic faults such as runtime exceptions and W3C HTML validation errors, JSTERA is able to accurately distinguish faults at the code-level and DOM-level through the test oracles it generates. Both tools achieve 100% precision, however, JSTERA achieves five-fold higher recall (70% on average) compared with ARTEMIS, which achieves 12.8% recall on average.

D. Threats to Validity

An external threat to the validity of our results is the limited number of web applications that we use to evaluate our approach. We mitigated this threat by using JavaScript applications that cover various application types. Another threat is that we validate the failed assertions through manual inspection, which can be error-prone. To mitigate this threat, we carefully inspected the code in which the assertion was failed to make sure that the injected fault was indeed responsible for the assertion failure. Regarding the reproducibility of our results, JSTERA and all the applications used in this study are publicly available, thus making the study replicable.

VI. CONCLUSIONS

In this paper, we presented a technique to automatically generate regression test cases for JavaScript applications at two complementary levels: (1) individual JavaScript functions, (2) DOM event sequences. We also proposed a method for effectively generating test oracles along with the test cases, for detecting faults in JavaScript code as well as the DOM tree. We implemented our approach in an open-source tool called JSTERA. We empirically evaluated JSTERA on 13 web applications. The results show that the generated tests by JSTERA achieve high coverage (68.4% on average), and that the injected faults can be detected with a high accuracy rate (recall 70%, precision 100%).

REFERENCES

[1] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 571–580.

[2] A. Marchetto and P. Tonella, "Using search-based algorithms for Ajax event sequence generation during testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.

[3] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *Proc. 1st Int. Conference on Sw. Testing Verification and Validation (ICST'08)*. IEEE Computer Society, 2008, pp. 121–130.

[4] 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.

[5] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Proc. Symp. on Security and Privacy (SP'10)*. IEEE Computer Society, 2010, pp. 513–528.

[6] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Pythia: Generating test cases with oracles for JavaScript applications," in *Proc. International Conference on Automated Software Engineering (ASE), New Ideas Track*. IEEE Computer Society, 2013.

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

[8] S. Mirshokraie and A. Mesbah, "JSART: JavaScript assertion-based regression testing," in *Proc. International Conference on Web Engineering (ICWE'12)*. Springer, 2012, pp. 238–252.

[9] K. Sen, S. Kalasapur, T., and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'013)*. ACM, 2013.

[10] C. Jensen, A. Møller, and Z. Su, "Server interface descriptions for automated testing of JavaScript web applications," in *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'013)*. ACM, 2013.

[11] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 2, pp. 278–292, 2012.

[12] ——, "Generating parameterized unit tests," in *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011, pp. 364–374.

[13] K. Taneja and T. Xie, "DiffGen: Automated regression unit-test generation," in *Proc. International Conference on Automated Software Engineering (ASE'08)*. IEEE Computer Society, 2008, pp. 407–410.

[14] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 1, pp. 29–45, 2009.

[15] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proc. International Conference on Software Engineering (ICSE'11)*, 2011, pp. 870–880.

[16] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*. IEEE Computer Society, 2013, pp. 55–64.

[17] "jQuery API," http://api.jquery.com.

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

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

[20] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. 6th International Conference on Software Testing Verification and Validation (ICST'13)*. IEEE Computer Society, 2013, pp. 74–83.

[21] "Pythia," http://salt.ece.ubc.ca/software/pythia/.

[22] "Qunit," http://qunitjs.com.

[23] "JSCover," http://tntim96.github.io/JSCover/.