Effective Test Generation and Adequacy Assessment for JavaScript-based Web Applications

by

Shabnam Mirshokraie

BSc. Computer Engineering, Ferdowsi University of Mashhad, Iran, 2006

MSc. Computing Science, Simon Fraser University, Canada, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF APPLIED SCIENCE

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

April 2015

Abstract

Today's modern Web applications rely heavily on JavaScript and client-side runtime manipulation of the DOM (Document Object Model) tree. One way to provide assurance about the correctness of such highly evolving and dynamic applications is through testing. However, JavaScript is loosely typed, dynamic, and notoriously challenging to analyze and test.

The work presented in this dissertation has focused on advancing the state-of-the-art in testing JavaScript-based web applications by proposing a new set of techniques and tools. We proposed (1) a new automated technique for JavaScript regression testing, which is based on inferring invariant assertions, (2) the first JavaScript mutation testing tool, capable of guiding the mutation generation towards behaviour-affecting mutants in error-prone portions of the code, (3) an automatic technique to generate test cases for JavaScript functions and events; Mutation analysis is used to generate test oracles, capable of detecting regression JavaScript and DOM-level faults, and (4) utilizing existing DOM-dependent assertions as well as useful execution information inferred from a DOM-based test suite to automatically generate assertions for unit-level testing of JavaScript functions.

To measure the effectiveness of the proposed approaches, we evaluated each method presented in this thesis by conducting various empirical studies and comparisons with existing testing techniques. The evaluation results point to the effectiveness of the proposed test generation and test assessment techniques in terms of accuracy and fault detection capability.

Table of Contents

[1] This work is under preparation.

List of Tables

List of Figures

Chapter 1

Introduction

In this section, we provide an introduction to modern web application testing, followed by some of the current techniques used for automating the testing process.

1.1 Test Automation

Software testing is an integral part of the software engineering. Testing helps to improve the quality and dependability of the applications. However, software systems have become more complex in last decades, with using different technologies and programming languages, that are even implemented by different developers. As a result, writing high-quality test cases for such applications that can assure their correct behaviour becomes more complicated, time consuming, and effort intensive for developers [15]. Automatic test case generation can significantly reduce the time and manual effort, while increasing the reliability of web applications.

Determining the desired behaviour of the application under test for a given input is called the Oracle Problem. Automating the oracle generation is an important part of the testing process since manual testing is expensive and time consuming, mainly because of the manual effort that should be spent in identifying the proper oracle. Our goal is to improve the dependability and reliability of the modern web applications by automating the test and oracle generation process with different techniques and tools proposed in this thesis.

1.1.1 Web Testing

One of the common engines of today's modern web applications is JavaScript. Developers employ JavaScript to add functionality, dynamically change the Document Object Model (DOM) structure, and communicate with web servers. Given the increasing reliance of web applications on this language, it is important to check its correct behaviour.

Automatically generating effective test suites for JavaScript applications is particularly challenging compared with traditional languages. The event-driven and highly dynamic nature of JavaScript, as well as its run-time interaction with the DOM make JavaScript applications error-prone [80] and difficult to test. The huge event space of such applications hinders model inferring techniques to cover the whole state space in a limited amount of time. Moreover, inferring test assertions with high fault finding capability requires a precise analysis of the interaction between the JavaScript and the DOM. Providing an accurate mapping between the two components becomes difficult as the JavaScript code and the DOM increase in size. It is also challenging to identify a proper set of test oracles from the large number of oracles that potentially can be selected.

To test JavaScript-based web applications, developers often write test cases using frameworks such as SELENIUM [10] to examine the correct interaction of the user with web application (GUI testing) and QUNIT [9] to test the proper functionality of the individual units (unit testing). Although such frameworks help to automate test execution, the test cases need to be written manually, which can be tedious and inefficient. Using SELENIUM to write DOM-based tests and assertions requires little knowledge about the internal operations performed at the client side code; The tester needs only basic knowledge of common event sequences to cover important DOM elements to assert. This makes it easier for the tester to write DOM-based test suites. However, DOM-based assertions can potentially miss some portion of the code, while more fine grained unit-level assertions might be capable of detecting such faults. Furthermore, since DOM-based tests are agnostic of the JavaScript code, finding the root cause of an error during DOM-based testing is more expensive than during unit testing. Figure 1.1 shows a sample DOM-based test case. The test case contains the sequence of clicking on different

```
1 @Test
2 public void testCase1(){
3   WebElement divElem=driver.findElements(By.id("divElem"));
4   divElem.click();
5   driver.findElements(By.id("endCell")).getSize().height;
6   WebElement startCell=driver.findElements(By.id("startCell"));
7   startCell.click();
8   driver.findElements(By.id("startCell"));
9   ...
10 }
```

Figure 1.1: SELENIUM test case.

DOM elements without an observable communication with the executed JavaScript code. On the other hand, writing unit test assertions for web applications that have rich interaction with the DOM through their JavaScript code is more tedious. To generate unit-level assertions, the technique needs to precisely interpret the full range of interaction between the code level operations of a unit and the DOM level operations of a system, otherwise it may not be able to assert the correctness of a particular behaviour when the unit is used as a part of a system. The inherent characteristics of unit and DOM-based tests, indicate that they are complementary and that there is a trade-off in individually using each to detect faults.

1.1.2 Current Test Generation Techniques

Different test and oracle generation techniques have been proposed to overcome the aforementioned problems. Sen et al. [99] 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. [58] 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.

There has been limited work on oracle generation for testing. Fraser et al. [43] 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 be-

haviour. They further identify [42] relevant pre-conditions on the test inputs and post-conditions on the outputs to ease human comprehension. Artzi et al. proposed Artemis [17], which supports automated testing of JavaScript applications. Artemis considers the event-driven execution model of a JavaScript application for feedback-directed testing.

Open Problems. Although, researchers have recently developed automated test generation techniques for JavaScript-based applications [17, 65, 66, 70, 96], current 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 [17, 70], or JavaScript runtime exceptions [17]. 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 the above mentioned shortcomings, we proposed an automated test case and assertion generation technique for JavaScript applications.

1.2 Adequacy Assessment

While automated testing can help the tester to assure the application's dependability and detect faults in the application code, it does not reveal the trustworthiness of the written tests. In order to understand how well the functionality and the data is being tested, we need to assess the quality of the tests. A large body of research

has been accomplished to assess the quality of test suites from different perspectives: (1) code coverage, and (2) mutation analysis. While code coverage tells how much of the source code is exercised by the test suite, it does not provide sufficient insight into the actual quality of the tests. Mutation testing has been proposed as a fault-based testing technique to assess and improve the quality of a test suite.

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

1.2.1 Mutation Testing Challenges

Despite being an effective test adequacy assessment method [16, 61], 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 [27]. Equivalent mutants have no observable effect on the application's behaviour, and as a result, cannot be killed by test cases. Empirical studies indicate that about 45% of all undetected mutants are equivalent [64, 97]. Establishing mutant equivalence is an undecidable problem [27]. According to a recent study [64], it takes on average 15 minutes to manually assess one single first-order mutant. While detecting equivalent mutants consumes considerable amount of time, there is still no fully automated technique that is capable of detecting all the equivalent mutants [64].

A large body of research has been conducted to turn mutation testing into a practical approach. To reduce the computational cost of mutation testing, researchers have proposed three main approaches to generate a smaller subset of all possible mutants: (1) *mutant clustering* [59], which is an approach that chooses a subset of mutants using clustering algorithms; (2) *selective mutation* [19, 78, 110],

which is based on a careful selection of more effective mutation operators to generate less mutants; and (3) *higher order mutation* (HOM) testing [60], which tries to find rare but valuable higher order mutants that denote subtle faults [61].

The problem of detecting equivalent mutants has been tackled by many researchers. The main goal of all equivalent mutant detection techniques is to help the tester identify the equivalent mutants after they are generated. According to the taxonomy suggested by Madeyski et al. [64], there are three main categories of approaches that address the problem of equivalent mutants: (1) detecting equivalent mutants [83], (2) avoiding equivalent mutant generation [46], and (3) suggesting equivalent mutants [97].

However, these solutions suffer from the following limitations:

1. They are involved with considerable amount of manual effort, and thus are error-prone;

2. The mutants need to be executed against the test suite, which limits the efficiency of the technique as the number of mutants increase.

3. The system only recommends testers to focus on those mutations that are more likely to be non-equivalent. These techniques are not fully automated and are used as a supporting system for the tester;

To tackle the above mentioned issues, we proposed a fully automated mutation generation technique that reduces the number of equivalent mutants and guides testers towards designing test cases for important portions of the code from the application's behaviour point of view.

1.3 Research Questions

To improve the dependability of JavaScript web applications, we designed two high-level research questions:

RQ 1.3.A. *How can we automatically generate effective test cases for JavaScript applications?*

In response to web testing challenges, we (1) designed an automated test case and oracle generator, which is capable of detecting faults in the JavaScript appli-

cations for both unit and DOM level, and (2) proposed an approach to exploit the existing DOM-based test suite in order to generate unit-level assertions.

RQ 1.3.B. *How can we effectively assess the quality of the existing JavaScript test cases?*

To assess the quality of test cases, we proposed a new JavaScript mutation testing approach, which helps to guide the mutation generation process towards parts of the code that are error-prone or likely to influence the program's output.

1.4 Contributions

In response to the first and second research questions as outlined in Section 1.3, the following papers have been published:

- Chapter 2: "JavaScript Assertion-based Regression Testing" [72], S. Mirshokraie and A. Mesbah, ICWE, 2012, 238-252.
- Chapter 3: "Efficient JavaScript Mutation Testing" [73], S. Mirshokraie, A. Mesbah and K. Pattabiraman, ICST, 2013, 74-83 (Best paper Runner-up award); "Guided Mutation Testing for JavaScript Web Applications" [76], S. Mirshokraie, A. Mesbah and K. Pattabiraman, in press (TSE 2015);
- Chapter 4: "JSEFT: Automated JavaScript Unit Test Generation" [75], S. Mirshokraie, A. Mesbah and K. Pattabiraman, To appear (ICST 2015); "PYTHIA: Generating Test Cases with Oracles for JavaScript Applications" [74], S. Mirshokraie, A. Mesbah and K. Pattabiraman, ASE, 2013, New Ideas Track, 610-615.
- Chapter 5: "Atrina: Inferring Unit Oracles from GUI Test Cases", under preparation.

I have also contributed to the following related publications:

- Automated Analysis of CSS Rules to Support Style Maintenance [68]: A. Mesbah and S. Mirshokraie, ICSE'12, 408-418;
- A Systematic Mapping Study of Web Application Testing [45]: V. Garousi, A. Mesbah, A. Betin Can and S. Mirshokraie, IST, vol. 55, no. 8, 1374-1396, 2013;

Chapter 2

JavaScript Invariant Detector

2.1 Abstract

One way to provide assurance about the correctness of highly evolving and dynamic applications is through regression testing. We propose [72] an automated technique for JavaScript regression testing, which is based on on-the-fly JavaScript source code instrumentation and dynamic analysis to infer invariant assertions. These obtained assertions are injected back into the JavaScript code to uncover regression faults in subsequent revisions of the web application under test. Our approach is implemented in a tool called JSART. We present our case study conducted on nine open source web applications to evaluate the proposed approach. The results show that our approach is able to effectively generate stable assertions and detect JavaScript regression faults with a high degree of accuracy and minimal performance overhead.

2.2 Introduction

Web applications usually evolve fast by going through rapid development cycles and are, therefore, susceptible to regressions, i.e., new faults in existing functionality after changes have been made to the system. One way of ensuring that such modifications (e.g., bug fixes, patches) have not introduced new faults in the modified system is through systematic regression testing. While regression testing of classical web applications has been difficult [103], dynamism and non-determinism pose an even greater challenge [93] for Web 2.0 applications.

```
1 function setDim(height, width) {
2   var h = 4*height, w = 2*width;
3   ...
4   return{h:h, w:w};
5 }

7 function play(){
8   $(#end).css("height", setDim($('body').width(), $('body').height()).h↩
        + 'px');
9   ...
10 }
```

Figure 2.1: Motivating JavaScript example.

In this work, we propose an automated technique for JavaScript regression testing, which is based on dynamic analysis to infer invariant assertions. These obtained assertions are injected back into the JavaScript code to uncover regression faults in subsequent revisions of the web application under test. Our technique automatically (1) intercepts and instruments JavaScript on-the-fly to add tracing code (2) navigates the web application to produce execution traces, (3) generates dynamic invariants from the trace data, (4) transforms the invariants into stable assertions and injects them back into the web application for regression testing.

Our approach is orthogonal to server-side technology, and it requires no manual modification of the source code. It is implemented in an open source tool called JSART (JavaScript Assertion-based Regression Testing). We have empirically evaluated the technique on nine open-source web applications. The results of our evaluation show that the approach generates stable invariant assertions, which are capable of spotting injected faults with a high rate of accuracy.

2.3 Motivation and Challenges

Figure 2.1 shows a simple JavaScript code snippet. Our motivating example consists of two functions, called `setDim` and `play`. The `setDim` function has two parameters, namely `height` and `width`, with a simple mathematical operation (Line 2). The function returns local variables, `h` and `w` (Line 4). `setDim` is called in the `play` function (Line 8) to set the `height` value of the CSS property of the DOM element with ID `end`. Any modification to the values of `height` or `width` would influence the returned values of `setDim` as well as the property

of the DOM element. Typical programmatic errors include swapping the order of `height` and `width` when they are respectively assigned to local variables `h` and `w` or calling `setDim` with wrong arguments, i.e., changing the order of function arguments.

Detecting such regression errors is a daunting task for web developers, especially in programming languages such as JavaScript, which are known to be challenging to test. One way to check for these regressions is to define invariant expressions of expected behaviour [71] over program variables and assert their correctness at runtime. This way any modification to `height`, `width`, `h`, or `w` that violates the invariant expression will be detected. However, manually expressing such assertions for web applications with thousands of lines of JavaScript code and several DOM states, is a challenging and time-consuming task. Our aim in this work is to provide a technique that automatically captures regression faults through generated JavaScript assertions.

2.4 Our Approach

Our regression testing approach is based on *dynamic analysis* of JavaScript code to infer invariants from a given web application. We use the thus obtained invariants as *runtime assertions* in the JavaScript code to automatically uncover regression errors that can be introduced after changes have been made to the web application in a subsequent reversion. Our approach is largely based on two assumptions (1) the current version of the web application, from which invariants are being generated, is bug-free (2) the inferred invariants capture program specifications that are unlikely to change frequently in the following revisions (we revisit these two assumptions in Section 5.5.4). Our regression testing technique is composed of the following four main steps: (1) JavaScript tracing, (2) Invariant generation, (3) Filtering unstable assertions, and (4) Regression testing through assertions. In the following subsections, we will describe each step in details.

2.4.1 JavaScript Tracing

In order to infer useful program invariants, we need to collect execution traces of the JavaScript code. The idea is to log as much program variable value changes at

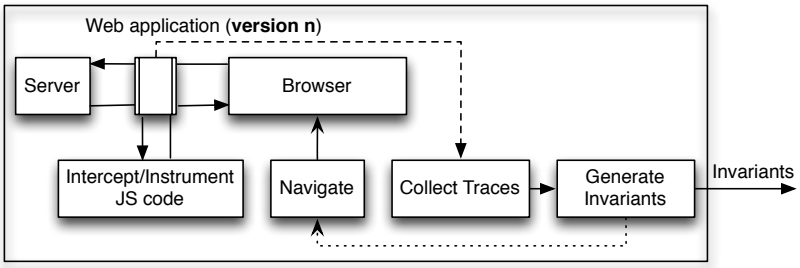


Figure 2.2: Overview of the JavaScript tracing and invariant generation steps (web application version *n*).

runtime as possible. Figure 2.2 depicts a block diagram of the tracing step. Our approach automatically generates trace data in three subsequent steps: (i) JavaScript interception and instrumentation, (ii) navigation, and (iii) trace collection. In the following, we explain each step in details.

JavaScript Interception and Instrumentation.

The approach we have chosen for logging variables is on-the-fly JavaScript source code transformation to add instrumentation code. We intercept all the JavaScript code of a given web application, both in JavaScript files and HTML pages, by setting up a proxy [21] between the server and the browser. We first parse the intercepted source code into an Abstract Syntax Tree (AST). We then traverse the AST in search of *program variables* as well as *DOM modifications* as described below.

Tracing Program Variables. Our first interest is the range of values of JavaScript program variables. We probe function entry and function exit points, by identifying function definitions in the AST and injecting statements at the start, end, and before every `return` statement. We instrument the code to monitor value changes of *global variables*, *function arguments*, and *local variables*. Per program point, we yield information on *script name*, *function name*, and *line number*, used for debugging purposes. Going back to our running example (Figure 2.1), our technique adds instrumentation code to trace `width`, `height`, `h`, and `w`. For each variable, we collect information on *name*, *runtime type*, and *actual values*. The runtime type is stored because JavaScript is a loosely typed language, i.e., the types of variables cannot be determined syntactically, thus we log the variable types at runtime.

Tracing DOM Modifications. In modern web applications, JavaScript code frequently interacts with the DOM to update the client-side user interface state. Our recent study [81] of four bug-tracking systems indicated that DOM-related errors form 80% of all reported JavaScript errors. Therefore, we include in our execution trace how DOM elements and their attributes are modified by JavaScript at runtime. For instance, by tracing how the CSS property of the 'end' DOM element in Figure 2.1 is changed during various execution runs, we can infer the range of values for the `height` attribute.

Based on our observations, JavaScript DOM modifications usually follow a certain pattern. Once the pattern is reverse engineered, we can add proper instrumentation code around the pattern to trace the changes. In the patterns that we observed, first a JavaScript API is used to find the desired DOM element. Next, a function is called on the returned object responsible for the actual modification of the DOM-tree. After recognizing a pattern in the parsed AST, we add instrumentation code that records the value of the DOM attribute before and after the actual modification. Hence, we are able to trace DOM modifications that happen programmatically through JavaScript.

Navigation.

Once the AST is instrumented, we serialize it back to the corresponding JavaScript source code file and pass it to the browser. Next, we navigate the application in the browser to produce an execution trace. The application can be navigated in different ways including (1) manual clicking (2) test case execution (3) or using a web crawler. To automate the approach, our technique is based on automated dynamic crawling [69]. The execution needs to run as much of the JavaScript code as possible and execute it in various ways. This can be achieved through visiting as many DOM state changes as possible as well as providing different values for function arguments.

Trace Collection.

As the web application is navigated, the instrumented JavaScript code produces trace data, which needs to be collected for further analysis. Keeping the trace data

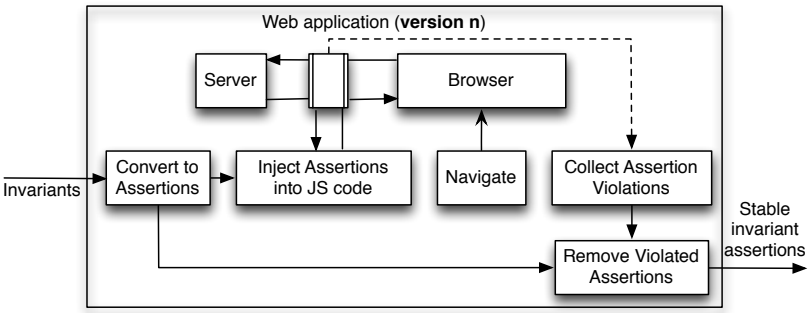


Figure 2.3: Overview of the filtering step to remove unstable invariant assertions, for web application version *n*.

in the browser's memory during the program execution can make the browser slow when a large amount of trace data is being produced. On the other hand, sending data items to the proxy as soon as the item is generated, can put a heavy load on the proxy, due to the frequency of HTTP requests. In order to tackle the aforementioned challenges, we buffer a certain amount of trace data in the memory in an array, post the data as an HTTP request to a proxy server when the buffer's size reaches a predefined threshold, and immediately clear the buffer in the browser's memory afterwards. Since the data arrives at the server in a synchronized manner, we concatenate the tracing data into a single trace file on the server side, which is then seeded into the next step (See Figure 2.2).

2.4.2 Invariant Generation

The assertion generation phase is involved with analyzing the collected execution traces to extract invariants. Substantial amount of research has been carried out on detecting dynamic program invariants [24, 32, 38, 51]. Our approach is based on Daikon [38] to infer likely invariants. As indicated with the dotted line in Figure 2.2, we cycle through the navigation and invariant generation phases until the size of generated invariant file remains unchanged, which is an indication that all possible invariants have been detected.

2.4.3 Filtering Unstable Invariant Assertions

The next step is to make sure that the generated invariants are truly invariants. An invariant assertion is called unstable when it is falsely reported as a violated assertion. Such assertions result in producing a number of false positive errors during the testing phase. To check the stability of the inferred invariants, we use them in the same version of the web application as assertions. Theoretically, no assertion violations should be reported because the web application has not changed. Hence, any assertion violation reported as such is a false positive and should be eliminated. Our filtering process, shown in Figure 2.3, consists of the following four processes:

- Converting the inferred invariants into checkable assertions;
- Injecting the invariant assertions into the same version of the web application;
- Navigating the web application;
- Collecting assertion violations and removing them;

From each of the inferred invariants, we generate an assertion in JavaScript format. We use on-the-fly transformation to inject the assertions directly into the source code of the same version of the web application. Since we have all the information about the program points and the location of the assertions, we can inject the assertions at the correct location in the JavaScript source code through the proxy, while the code is being sent to the client by the server. This way the assertions can run as part of the client-side code and gain access to the values of all program variables needed at runtime. Once the assertions are injected, we execute the web application in the browser and log the output. Next we collect and remove any violated assertions. The output of this step is a set of *stable* invariant assertions, used for automated regression testing in the next step.

2.4.4 Regression Testing through Assertions

Once a set of stable invariant assertions are derived from version `n` of a web application, they can be used for automatic regression testing a subsequent version

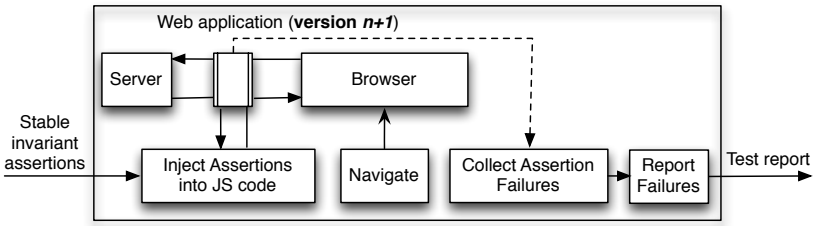


Figure 2.4: Overview of the JavaScript regression testing step through invariant assertions, for web application version *n+1*.

(n+1) of the web application. The regression testing phase is depicted in Figure 2.4.

We inject the inferred stable assertions to the JavaScript source code of the modified web application, in a similar fashion to the filtering step in Section 2.4.3. Once the assertions are injected, the new version of the web application is ready for regression testing. Any failed assertion during the testing phase generates an entry in the test report, which is presented to the tester at the end of the testing step. The generated test report provides precise information on the failed assertion, the file name, the line number, and the function name of the assertion.

```
function setDim(height, width) {
 2 assert((width < height), 'example.js:setDim:ENTER:POINT1');
 3 var h = 4*height, w = 2*width;
 4 ...
 5 assert((width < height), 'example.js:setDim:EXIT:POINT1');
 6 assert((w < h), 'example.js:setDim:EXIT:POINT1');
 7 return{h:h, w:w};
 8
 9
10 function play(){
11 $(#end).css("height", setDim($('body').width(), $('body').height()).h +↩
       'px');
12 assert(isIn($('#end').css('height'), {100, 200,
      300}),'example.js:play:POINT3');
13 ...
14
```

Figure 2.5: Invariant assertion code for JavaScript function parameters, local variables and DOM modifications. Injected assertions are shown in bold.

Figure 2.5 shows the automatically injected invariant assertions for our running example of Figure 2.1. Note that we do not show all the assertions as they clutter the figure. Each `assert` call has the invariant as the first parameter and the

corresponding debugging information in the second parameter, which includes information about script name, function name, and line number. In this example, the inferred invariants yield information about the inequality relation between function arguments, `width` and `height`, as well as local variables, `w` and `h`. The assertions in lines 2 and 5-6 check the corresponding inequalities, at entry and exit points of the `setDim` function at runtime. The example also shows the assertion that checks the `height` attribute of the DOM element, after the JavaScript DOM modification in the `play` function. The assertion that comes after the DOM manipulation (Line 12) checks the `height` value by calling the auxiliary `isIn` function. `isIn` checks the value of `height` to be in the given range, i.e., either 100, 200, or 300. Any values out of the specified range would violate the assertion.

2.5 Tool Implementation

We have implemented our JavaScript regression testing approach in a tool called JSART. JSART is written in Java and is available for download.[1]

JSART extends and builds on top of our InvarScope [47] tool. For JavaScript code interception, we use an enhanced version of Web-Scarab's proxy [21]. This enables us to automatically analyze and modify the content of HTTP responses before they reach the browser. To instrument the intercepted code, Mozilla Rhino[2] is used to parse JavaScript code to an AST, and back to the source code after instrumentation. The AST generated by Rhino's parser has traversal API's, which we use to search for program points where instrumentation code needs to be added. For the invariant generation step, we have extended Daikon [38] with support for accepting input and generating output in JavaScript syntax. The input files are created from the trace data and fed through the enhanced version of Daikon to derive dynamic invariants. The navigation step is automated by making JSART operate as a plugin on top of our dynamic AJAX crawler, CRAWLJAX [69].[3]

[1] http://salt.ece.ubc.ca/content/jsart/
[2] http://www.mozilla.org/rhino/
[3] http://www.crawljax.com

2.6 Empirical Evaluation

To quantitatively assess the accuracy and efficiency of our approach, we have conducted a case study following guidelines from Runeson and Höst [95]. In our evaluation, we address the following research questions:

RQ1 How successful is JSART in generating stable invariant assertions?

RQ2 How effective is our overall regression testing approach in terms of correctly detecting faults?

RQ3 What is the performance overhead of JSART?

The experimental data produced by JSART is available for download.[1]

2.6.1 Experimental Objects

Our study includes nine web-based systems in total. Six are game applications, namely, SameGame, Tunnel, TicTacToe, Symbol, ResizeMe, and GhostBusters. Two of the web applications are Jason and Sofa, which are a personal and a company homepage, respectively. We further include TuduList, which is a web-based task management application. All these applications are open source and use JavaScript on the client-side.

Table 2.1 presents each application's ID, name, and resource, as well as the characteristics of the custom JavaScript code, such as JavaScript lines of code (LOC), number of functions, number of local and global variables, as well as the cyclomatic complexity (CC). We use Eclipse IDE to count the JavaScript lines of code, number of functions, number of local as well as global variables. JSmeter [4] is used to compute the cyclomatic complexity. We compute the cyclomatic complexity across all JavaScript functions in the application.

2.6.2 Experimental Setup

To run the experiment, we provide the URL of each experimental object to JSART. In order to produce representative execution traces, we navigate each application

[4] http://jsmeter.info

Table 2.1: 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	TicTacToe	239	11	22	23	83	http://www.dynamicdrive.com/dynamicindex12/tictactoe.htm
4	Symbol	204	20	28	16	32	http://10k.aneventapart.com/2/Uploads/652
5	ResizeMe	45	5	4	7	2	http://10k.aneventapart.com/2/Uploads/594
6	GhostBusters	277	27	75	4	52	http://10k.aneventapart.com/2/Uploads/657
7	Jason	107	8	4	8	6	http://jasonjulien.com
8	Sofa	102	22	2	1	5	http://www.madebysofa.com/archive
9	TuduList	2767	229	199	31	28	http://tudu.ess.ch/tudu

several times with different crawling settings. Crawling settings differ in the number of visited states, depth of crawling, crawling time, and clickable element types. To obtain representative data traces, each of our experimental objects is navigated three times on average. Although JSART can easily instrument the source code of imported JavaScript libraries (e.g., jQuery, Prototype, etc), in our experiments we are merely interested in custom code written by developers, since we believe that is where most programming errors occur.

To evaluate our approach in terms of inferring stable invariant assertions (RQ1), we count the number of stable invariant assertions generated by JSART before and after performing the filtering step. As a last check, we execute the initial version of the application using the stable assertions to see whether our filtered invariant assertions are reporting any false positives.

Once the stable invariant assertions are obtained for each web application, we perform regression testing on modified versions of each application (RQ2). To that end, in order to mimic regression faults, we produce twenty different versions for each web application by injecting twenty faults into the original version, one at a time. We categorize our faults according to the following fault model:

1. **Modifying Conditional Statements**: This category is concerned with swapping consecutive conditional statements, changing the upper/lower bounds of loop statements, as well as modifying the condition itself;

Table 2.2: Properties of the invariant assertions generated by JSART.

App ID	Trace Data (MB)	# Total Assertions	# Entry Assertions	# Exit Assertions	# DOM Assertions	# Total Unstable Assertions	# Unstable Entry Assertions	# Unstable Exit Assertions	# Unstable DOM Assertions	# Total Stable Assertions	# Stable Entry Assertions	# Stable Exit Assertions	# Stable DOM Assertions
1	8.6	303	120	171	12	0	0	0	0	303	120	171	12
2	124	2147	1048	1085	14	14	9	5	0	2133	1039	1080	14
3	1.2	766	387	379	0	16	8	8	0	750	379	371	0
4	31.7	311	138	171	2	14	7	7	0	297	131	164	2
5	0.4	55	20	27	8	0	0	0	0	55	20	27	8
6	2.3	464	160	266	38	3	1	2	0	461	159	264	38
7	1.2	29	4	6	19	0	0	0	0	29	4	6	19
8	0.1	20	2	2	16	0	0	0	0	20	2	2	16
9	2.6	163	58	104	1	0	0	0	0	163	58	104	1

2. **Modifying Global/Local Variables**: In this category, global/local variables are changed by modifying their values at any point of the program, as well as removing or changing their names;

3. **Changing Function Parameters/Arguments**: This category is concerned with changing function parameters or function call arguments by swapping, removing, and renaming parameters/arguments. Changing the sequence of consecutive function calls is also included in this category;

4. **DOM modifications**: Another type of fault, which is introduced in our fault model is modifying DOM properties at both JavaScript code level and HTML code level.

For each fault injection step, we randomly pick a JavaScript function in the application code and seed a fault according to our fault model. We seed five faults from each category.

To evaluate the effectiveness of JSART (RQ2), 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.

To evaluate the performance of JSART (RQ3), we measure the extra time needed to execute the application while assertion checks are in place.

2.6.3 Results

In this section, we discuss the results of the case study with regard to our three research questions.

Generated Invariant Assertions.

Table 2.2 presents the data generated by our tool. For each web application, the table shows the total size of collected execution traces (MB), the total number of generated JavaScript assertions, the number of assertions at entry point of the functions, the number of assertions at exit point of the functions, and the number of DOM assertions. The unstable assertions before the filtering as well as the stable assertions after the filtering step are also presented. As shown in the table, for applications 1, 5, 7, 8, and 9, all the generated invariant assertions are stable and the filtering step does not remove any assertions. For the remaining four applications (2, 3, 4, 6), less than 5% of the total invariant assertions are seen as unstable and removed in the filtering process. Thus, for all the experimental objects, the resulting stable assertions found by the tool is more than 95% of the total assertions. Moreover, we do not observe any unstable DOM assertions. In order to assure the stability of the resulting assertions, we examine the obtained assertions from the filtering step across multiple executions of the original application. The results show that all the resulting invariant assertions are truly stable since we do not observe any false positives.

As far as RQ1 is concerned, our findings indicate that (1) our tool is capable of automatically generating a high rate of JavaScript invariant assertions, (2) the unstable assertions are less than 5% of the total generated assertions, (3) the filtering technique is able to remove the few unstable assertions, and (4) all the remaining

Table 2.3: Precision and Recall for JSART fault detection.

App ID	# FN	# FP	# TP	Precision (%)	Recall (%)
1	2	0	18	100	90
2	4	0	16	100	80
3	1	0	19	100	95
4	2	0	18	100	90
5	0	0	20	100	100
6	1	0	19	100	95
7	0	0	20	100	100
8	0	0	20	100	100
9	1	0	19	100	95

invariant assertions that JSART outputs are stable, i.e., they do not produce any false positives on the same version of the web application.

Effectiveness.

Since applications 3, 4, and 9 do not contain many DOM assertions, we were not able to inject 5 faults from the DOM modification category. Therefore, we randomly chose faults from the other fault model categories.

In Table 2.3, we present the accuracy of JSART in terms of its fault finding capability. The table shows the number of false negatives, false positives, true positives, as well as the percentages for precision and recall. As far as RQ2 is concerned, our results show that JSART is very accurate in detecting faults. The precision is 100%, meaning that all the injected faults, which are reported by the tool, are correct. This also implies that our filtering mechanism successfully eliminates unstable assertions as we do not observe any false positives. The recall oscillates between 80-100%, which is caused by a low rate of missed faults (discussed in Section 5.5.4 under Limitations). Therefore, as far as RQ2 is concerned, JSART is able to successfully spot the injected faults with a high accuracy rate.

Performance.

Figure 2.6 depicts the total running time needed for executing each web application with and without the assertion code. Checking a fairly large number of assertions at runtime can be time consuming. Thus, to capture the effect of the added assertions on the execution time, we exploit a 2-scale diagram. As shown in Figure 2.6,

each experimental object is associated with two types of data. The left-hand Y-axis represents the running time (seconds), whereas the right-hand Y-axis shows the number of assertions. This way we can observe how the number of assertions relates to the running time. As expected, the figure shows that by increasing the number of assertions, the running time increases to some degree. While the time overhead of around 20 seconds is more evident for the experimental object 2 (i.e., Tunnel with 2147 number of assertions), it is negligible for the rest of the experimental objects. Considering that Tunnel has 260 statements in total, the number of assertions instrumented in the code is eight times more than the number of statements in the original version. Therefore, it is reasonable to observe a small amount of overhead. Though assertions introduce some amount of overhead, it is worth mentioning that we have not experienced a noticeable change (i.e., freezing or slowed down execution) while running the application in the browser.

Thus, as far as RQ3 is concerned, the amount of overhead introduced by our approach is 6 seconds on average for our experimental objects, which is negligible during testing. Furthermore, based on our observations, the assertions do not negatively affect the observable behaviour of the web applications in the browser.

2.7 Discussion

Unstable Assertions.

As mentioned in Section 2.4.3, we observe a few number of unstable invariant assertions initially, which are removed by our filtering mechanism. By analyzing our trace data, we observe that such unstable assertions arise mainly because of the multiple runtime types of JavaScript variables. This is based on the fact that in JavaScript it is possible to change the type of a variable at runtime. However, Daikon treats variables as single type, selects the first observed type, and ignores the subsequent types in the trace data. This results in producing a few number of unstable invariant assertions for JavaScript. We remove such unstable assertions in our filtering step. A drawback of removing these assertions, is that our tool might miss a fault during the regression testing phase. However, according to our observations, such unstable assertions form only around 5% of the total generated

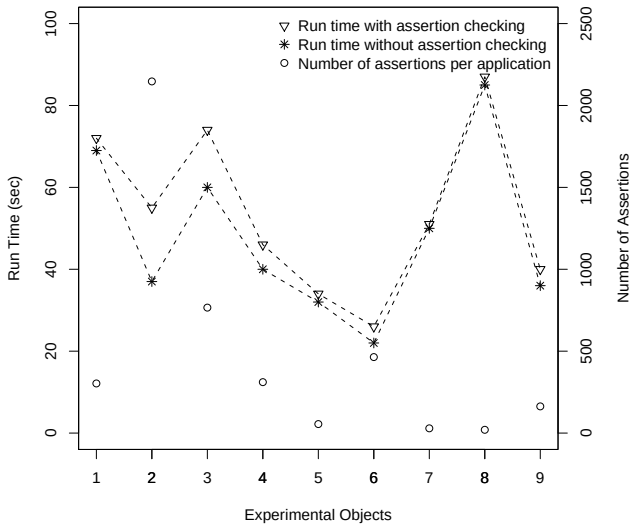


Figure 2.6: Performance plot of JSART.

assertions. Thus, we are still able to achieve high accuracy as presented in the previous section.

Limitations.

Our approach is not able to detect syntax errors that are present in the JavaScript code. Furthermore, tracing DOM manipulations using APIs other than the standard DOM API or jQuery is currently not supported by JSART. Further, a regression fault either directly violates an invariant assertion, or it can violate closely related assertions, which have been affected by the fault. However, if the tool is not able to infer any invariants in the affected scope of the error, it fails to detect the fault. This results in observing a low rate of false negatives as illustrated in Section 5.5.

Revisiting the Assumptions.

As we mentioned in Section 5.4, we assume that the current version of the web application is bug-free. This is based on the fact that in regression testing a gold standard is always needed as a trusted version for comparing the test results against

Table 2.4: Manual effort imposed by our approach for deriving stable invariant assertions.

App ID	Total Time (min)	Manual Effort (min)
1	13	4
2	11.5	3
3	15.5	5
4	11	3
5	6.5	2.5
6	9	4.5
7	7.5	3.5
8	6.5	2
9	18	13

[23] to detect regression faults. However, if the original version of the application does contain an error, the generated assertions might reflect the error as well, and as such they are not able to detect the fault. Our second assumption states that the program specifications are unlikely to change frequently in revisions. Here we assume that software programs evolve gradually and regression faults are mostly due to small changes. However, if major upgrades occur in subsequent revisions such that the core specification of the application is affected, the inferred invariants from the original version may not be valid any longer and new invariant assertions need to be generated.

Automation Level.

While the testing phase of JSART is fully automated, the navigation part requires some manual effort. Although the crawling is performed automatically, we do need to manually setup the tool with different crawling configurations per application execution. Moreover, for each application run, we manually look at the size of the invariant output to decide whether more execution traces (and thus more crawling sessions) are needed. We present the manual effort involved with detecting stable invariant assertions in Table 2.4. The table shows the total time, which is the duration time of deriving stable assertions including both automatic and manual parts. The reported manual effort contains the amount of time required for setting up the tool as well as the manual tasks involved with the navigation part. The results show the average manual effort is less than 5 minutes.

2.8 Related Work

Automated testing of modern web applications is becoming an active area of research [17, 66, 70, 87]. Most of the existing work on JavaScript analysis is, however, focused on spotting errors and security vulnerabilities through static analysis [48, 49, 112]. We classify related work into two broad categories: web application regression testing and program invariants.

Web Application Regression Testing.

Regression testing of web applications has received relatively limited attention from the research community [103, 108]. Alshahwan and Harman [14] discuss an algorithm for regression testing of web applications that is based on session data [36, 100] repair. Roest et al. [93] propose a technique to cope with the dynamism in Ajax web interfaces while conducting automated regression testing. None of these works, however, target regression testing of JavaScript in particular.

Program Invariants.

The concept of using invariants to assert program behaviour at runtime is as old as programming itself [29]. A more recent development is the automatic detection of program invariants through dynamic analysis. Ernst et al. have developed Daikon [38], a tool capable of inferring likely invariants from program execution traces. Other related tools for detecting invariants include Agitator [24], DIDUCE [51], and DySy [32]. Recently, Ratcliff et al. [90] have proposed a technique to reuse the trace generation of Daikon and integrate it with genetic programming to produce useful invariants. Conceptually related to our work, Rodríguez-Carbonell and Kapur [92] use inferred invariant assertions for program verification.

Mesbah et al. [70] proposed a framework called ATUSA for manually specifying generic and application-specific invariants on the DOM-tree and JavaScript code. These invariants were subsequently used as test oracles to detect erroneous behaviours in modern web applications. Pattabiraman and Zorn proposed DoDOM [87], a tool for inferring invariants from the DOM tree of web applications for reliability testing.

To the best of our knowledge, our work in this paper is the first to propose an

automated regression testing approach for JavaScript, which is based on JavaScript invariant assertion generation and runtime checking.

Chapter 3

JavaScript Mutation Testing

3.1 Abstract

Mutation testing is an effective test adequacy assessment technique. However, there is a high computational cost in executing the test suite against a potentially large pool of generated mutants. Moreover, there is much effort involved in filtering out equivalent mutants. Prior work has mainly focused on detecting equivalent mutants after the mutation generation phase, which is computationally expensive and has limited efficiency. We propose [73, 76] an algorithm to select variables and branches for mutation as well as a metric, called *FunctionRank*, to rank functions according to their relative importance from the application's behaviour point of view. We present a technique that leverages static and dynamic analysis to guide the mutation generation process towards parts of the code that are more likely to influence the program's output. Further, we focus on the JavaScript language, and propose a set of mutation operators that are specific to web applications. We implement our approach in a tool called MUTANDIS. The results of our empirical evaluation show that (1) more than 93% of generated mutants are non-equivalent, and (2) more than 75% of the surviving non-equivalent mutants are in the top 30% of the ranked functions.

3.2 Introduction

Mutation testing is a fault-based testing technique to assess and improve the quality of a test suite. The technique first generates a set of mutants, i.e., modified versions

of the program, by applying a set of well-defined mutation operators on the original version of the system under test. These mutation operators typically represent subtle mistakes, such as typos, commonly made by programmers. A test suite's adequacy is then measured by its ability to detect (or 'kill') the mutants, which is known as the adequacy score (or mutation score).

Despite being an effective test adequacy assessment method [16, 61], 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 [27]. Equivalent mutants have no observable effect on the application's behaviour, and as a result, cannot be killed by test cases. Empirical studies indicate that about 45% of all undetected mutants are equivalent [64, 97]. Establishing mutant equivalence is an undecidable problem [27]. According to a recent study [64], it takes on average 15 minutes to manually assess one single first-order mutant. While detecting equivalent mutants consumes considerable amount of time, there is still no fully automated technique that is capable of detecting all the equivalent mutants [64].

There has been significant work on reducing the cost of detecting equivalent mutants. According to the taxonomy suggested by Madeyski et al. [64], three main categories of approaches deal with the problem of equivalent mutants: (1) detecting equivalent mutants [82, 83], (2) avoiding equivalent mutant generation [13], and (3) suggesting equivalent mutants [97]. Our proposed technique falls in the second category (these categories are further described in Section 5.6).

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

Our mutation testing approach is generic and can be applied to any programming language. However, in this work, we implement our technique for JavaScript,

a loosely-typed dynamic language that is known to be error-prone [31, 79] and difficult to test [17, 70]. 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 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 in this work.

The key contributions of this work are:

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

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

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

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

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

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

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

Figure 3.1: JavaScript code of the running example.

3.3 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 5.1 presents a snippet of a JavaScript-based game that we will use as a running example throughout this thesis. The application contains four main func-

tions as follows:

1. `startPlay` function calls `setup` to set the dimension of all `div` DOM elements;

2. `setup` function is responsible for setting the `height` value of the `css` property of all the DOM elements with the given class name. The actual dimension computation is performed by calling the `getDim` function;

3. `getDim` receives two parameters `width` and `height` based on which it returns the calculated dimension;

4. Finally, `endGame` sets the `height` value of the `css` property of a DOM element with id `startCell`, to indicate a game termination.

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

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

3.4 Overview of Approach

An overview of our mutation testing technique is depicted in Figure 3.2. 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 3.2.

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

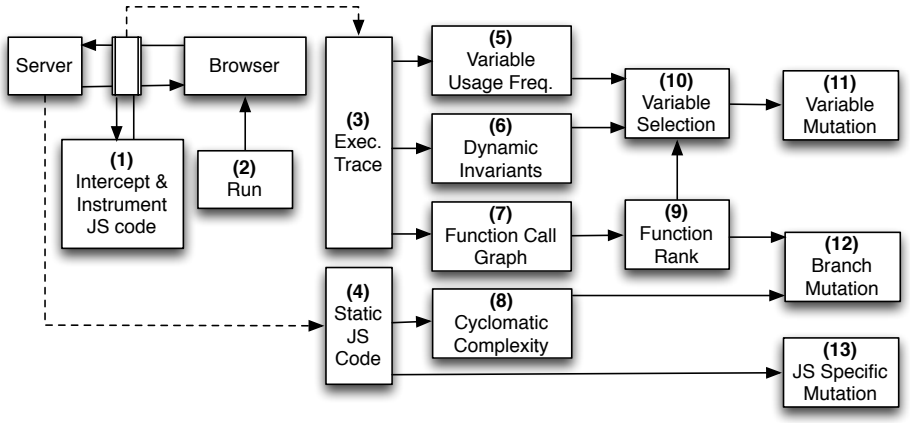


Figure 3.2: Overview of our mutation testing approach.

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 functions' call frequency. In addition to dynamically inferred information from the execution traces, we also construct the function call graph of the application by incorporating both static and dynamic information.

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

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

In addition to variables, our technique mutates branch conditions, including loops. Functions with high cyclomatic complexity are known to be more error-prone and challenging to test [20, 77], as the tester needs to detect and exercise all the different paths of the function. We therefore (4) statically analyze the JavaScript

code of the web application, and (8) measure its cyclomatic complexity. To perform branch mutation (12), we target the highly ranked functions (selected in 9) that also exhibit high cyclomatic complexity.

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

Our overall guided mutation testing algorithm is presented in Algorithm 1. In the following three sections, we describe in detail our technique for ranking functions (Section 3.5), ranking and selecting variables (Section 3.6), and performing the actual mutations, including the mutation operators (Section 3.7).

3.5 Ranking Functions

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

3.5.1 Ranking Functions for Variable Mutation

In order to rank and select functions for variable mutation generation, we propose a new metric called *FunctionRank*, which is based on *PageRank* [26], but takes dynamic function calls into account. As such, *FunctionRank* measures the relative importance of each function at runtime. To calculate this metric, we use a function call graph inferred from a combination of static and dynamic analysis (line 6 in Algorithm 1). Our insight is that the more a function is used, the higher its impact will be on the application's behaviour. As such, we assign functions that are highly ranked, a higher selection probability for mutation.

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

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

We enhance our dynamically inferred call graph of the executed functions by merging the graph with the statically obtained call graph containing uncovered functions. Note that constructing function call graph for the JavaScript applications using static analysis is often unsound due to highly dynamic nature of the JavaScript language. In JavaScript functions can be called through dynamic property access (e.g., `array[func]`). They can be stored in object properties with different names, and properties can be dynamically added or removed. Moreover, JavaScript functions are first class meaning that they can be passed as parameters. While static program analysis cannot reason about such dynamic function calls in JavaScript applications, relying on pure dynamic analysis can also lead to an incomplete call graph because of the unexecuted functions that are part of the uncovered code at run-time. Therefore, in our approach we choose to first construct our function call graph based on dynamic information obtained during the execution, and then make use of static analysis for those functions that are remained uncovered during the execution.

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

Algorithm 1: Guided Mutation Algorithm.

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

$App \leftarrow$ INSTRUMENT(App)
begin
  $trace \leftarrow$ COLLECTTRACE(App)
  $\{callFrq_{f_{i,j}}, varUsgFrq_{f_i}, invars_{f_i}\} \leftarrow$ GETREQUIREDINFO($trace$)
  $l = m = 0$
  while $l < MaxVarMut$ **do**
    $\{FR(f_i)_{i=0}^{n}\} \leftarrow$ FUNCTIONRANK($callGraph, callFrq_{f_{i,j}}$)
    $mutF \leftarrow$ SELECTFUNC($(FR(f_i))_{i=0}^{n}$)
    $\alpha \leftarrow \frac{1}{1-ReadVar_{f_i}}$
    $candidVars_{mutF} \leftarrow invars_{mutF} \cup \{v_i | varUsgFrq_{mutF}(v_i) > \alpha\}$
    $\{pr(v_i \in candidVars_{mutF})\} \leftarrow \frac{1}{|candidVars_{mutF}|}$
    $mutVar \leftarrow$ SELECTVAR($candidVars_{mutF}, pr(v_i)$)
    $mutant_l \leftarrow$ VARIABLEMUTATION($mutF, mutVar, varMutOps$)
    $l{+}{+}$
  end
  $varMutants \leftarrow \bigcup mutant_{l=1}^{MaxVarMut}$
  while $m < MaxBrnMut$ **do**
    $\{pr(f_i)_{i=0}^{n}\} \leftarrow \frac{fcc(f_i) \times FR(f_i)}{\sum_{j=1}^{n} fcc(f_i) \times FR(f_i)}$
    $mutF \leftarrow$ SELECTFUNC($(pr(f_i))_{i=0}^{n}$)
    $mutBrn \leftarrow$ SELECTRANDOMBRN($mutF$)
    $mutant_m \leftarrow$ BRANCHMUTATION($mutBrn, brnMutOps$)
    $m{+}{+}$
  end
  $brnMutants \leftarrow \bigcup mutant_{m=1}^{MaxBrnMut}$
  $Mutants \leftarrow varMutants \cup brnMutants$
  return $Mutants$
end

The FunctionRank Metric. The original *PageRank* algorithm [26] assumes that for a given vertex, the probability of following all outgoing edges is identical, and hence all edges have the same weight. For $FunctionRank$, we instead apply edge weights proportional to the dynamic call frequencies of the functions.

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

$$FR(f_i) = \sum_{j \in M(f_i)} FR(f_j) \times l(f_j, f_i), \tag{3.1}$$

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

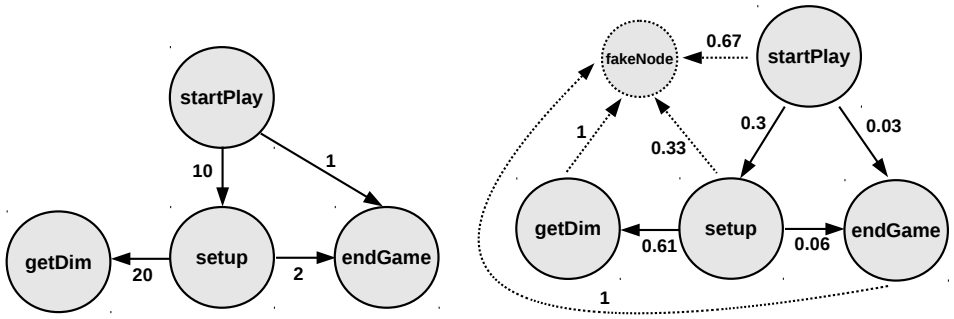
The initial *PageRank* metric requires the sum of weights on the outgoing edges to be 1. Therefore, to solve equation 3.1, we need to normalize the edge weights from each function in our formula such that for each i, $\sum_{j=1}^{n} l(f_i, f_j) = 1$. To preserve the impact value of call frequencies on edges when compared globally in the graph, we normalize $l(f_i, f_j)$ over the sum of weights on all edges. Since outgoing edges from function f_i should sum to 1, an extra node called $fakeNode$ is added to the graph. Note that the extra $fakeNode$ is different from the mock *main* node added earlier. $fakeNode$ contains an incoming edge from f_i, where:

$$l(f_i, fakeNode) = 1 - \sum_{j=1}^{n} l(f_i, f_j) \tag{3.2}$$

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

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

We initially assign equal $FunctionRank$ values to all nodes in 3.1.The calculation of $FunctionRank$ is performed recursively, until the values converge. Thus,



(a) Call graph with call numbers. **(b)** Call graph with call frequencies.

Figure 3.3: Call graph of the running example.

the $FunctionRank$ of a function i depends on:

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

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

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

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

Figure 3.3 depicts the function call graph obtained from our running example (Figure 5.1). The labels on the edges of Figure 3.3a show the number of calls to each function in the graph. Figure 3.3b shows the modified graph with the extra node $fakeNode$ added to compute the normalized function call frequency values. In our example, assuming that the number of `div` elements is 20 (line

3 in Figure 5.1), `setup` will be called 20 times and `endGame` will be called once (lines 4 and 6). Now, assume that the number of DOM elements with the class name specified as the input to function `setup` varies each time `setup` is called (line 10) such that two elements have a length of zero and the total length of the rest is 20. Then, function `endGame` is called twice (in line 12) when the length of such elements is zero, and `getDim` is called 20 times in total (line 14). Therefore, the call frequencies of functions `setup` and `endGame` become 0.3 and 0.03 respectively when they are called by `startPlay` in lines 4 and 6. Similarly, the call frequencies of `getDim` and `endGame` become 0.61 and 0.06, respectively, when called by `setup`.

Note that if the weight on an outgoing edge of a given function is simply normalized over the sum of the weights on all the outgoing edges of that function, then the call frequencies for both `setup` and `getDim` become 0.91 when they are called by `startPlay` and `setup`, respectively. However, as shown in Figure 3.3a the number of times that function `getDim` is called is twice that of `setup`. To obtain a realistic normalization, we have introduced the $fakeNode$, as shown in Figure 3.3b.

Table 3.1 shows the computed $FunctionRank$ values, using equation 3.1, for each function of the running example. The values are presented as percentages. `getDim` achieves the highest $FunctionRank$ because of the relatively high values of both the incoming edge weight (where `getDim` is called by `setup` in line 14 in Figure 5.1), and the $FunctionRank$ of its caller node, `setup`. These ranking values are used as probability values for selecting a function for mutation.

To illustrate the advantage of $FunctionRank$, we show the same calculation using the traditional *PageRank* metric, i.e., without considering dynamic edge weights. As shown in Table 3.1, `endGame` obtains the highest ranking using *PageRank*. However, this function has not been used extensively during the application execution, and hence has only limited impact on the behaviour of the application. In contrast, when $FunctionRank$ is used, `endGame` falls to the third place, and is hence less likely to be chosen for mutation.

Table 3.1: Computed *FunctionRank* and *PageRank* for the running example.

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

3.5.2 Ranking Functions for Branch Mutation

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

The cyclomatic complexity measures the number of linearly independent paths through a program's source code [67]. By using this metric, we aim to concentrate the branch mutation testing effort on the functions that are error-prone and harder to test.

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

We compute the probability of choosing a function f_i for branch mutation using the previously measured *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:

$$p(f_i) = \frac{fcc(f_i) \times FR(f_i)}{\sum_{j=1}^{n} fcc(f_j) \times FR(f_j)}, \tag{3.3}$$

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

Table 3.2 shows the cyclomatic complexity, the frequency, and the function selection probability measured for each function in our example (Figure 5.1). The probabilities are obtained using equation 3.3. As shown in the table, `getDim` achieves the highest selection probability as both its *FunctionRank* and cyclomatic

Table 3.2: Ranking functions for branch mutation (running example).

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

complexity are high.

3.6 Ranking Variables

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

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

3.6.1 Variables Involved in Dynamic Invariants

A recent study [98] showed that if a mutation violates dynamic invariants, it is very likely to be non-equivalent. This suggests that mutating variables that are involved in forming invariants affects the expected behaviour of the application with a high probability. Inspired by this finding, we infer invariants from the execution traces, as depicted in Figure 3.2. 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 [72]. 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 5.1), 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.

3.6.2 Variables with High Usage Frequency

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

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

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

$$\alpha = \frac{1}{ReadVariables_{f(i)}}, \tag{3.4}$$

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

Going back to the `getDim` function in our running example of Figure 5.1, the values of function parameters `width` and `height`, as well as the local variables `w` and `h` are read just once in lines 19 and 20, when they are involved in a number of simple computations. The result of the calculation is assigned to the local variable

`v`, which then is checked as a condition for the `if-else` statement. `v` is returned from the function in either line 22 or 24, depending on the outcome of the `if` statement. In this example, variable `v` has the highest usage frequency since it has been used as a condition in a conditional statement as well as the returned value of the `getDim` function.

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

3.7 Mutation Operators

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

3.7.1 Generic Mutation Operators

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 overall guided mutation process includes:

- Selecting a function as described in Section 3.5.1 and mutating a *variable* randomly selected from the list of potential candidates obtained from the variable ranking phase (Section 3.6),
- Selecting a function as described in Section 3.5.2 and mutating a *branch statement* selected randomly (lines 16-19 in Algorithm 1).

Table 3.3 shows the generic mutation operators we use for mutating global variables, local variables as well as function parameters/arguments. Table 3.4

Table 3.3: Generic mutation operators for variables and function parameters.

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

Table 3.4: Generic mutation operators for branch statements.

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

presents the operators we use for changing `for` loops, `while` loops, `if` and `switch-case` statements, as well as `return` statements.

3.7.2 JavaScript-Specific Mutation Operators

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

Adding/Removing the `var` keyword. Using `var` inside a function declares the variable in local scope, thus preventing overwriting of global variables ([31, 56, 84]). A common mistake is to forget to add `var`, or to add a redundant `var`, both of which we consider.

Removing the global search flag from `replace`. A common mistake is assuming that the string `replace` method affects all possible matches. The `replace` method only changes the first occurrence. To replace all occurrences, the global modifier needs to be set ([12, 94, 105]).

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 ([28, 105]).

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`) ([50, 55, 84]).

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 ([31, 94, 105]). Otherwise, an error will result.

Table 3.5: DOM, jQuery, and XmlHttpRequest (XHR) operators.

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

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 ([31, 89, 105]), 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())` ([94]).

In addition, we propose a list of DOM specific mutation operators within the JavaScript code. Table 3.5 shows a list of DOM operators that match DOM modification patterns in either pure JavaScript language or the JQUERY library. We further include two mutation operators that target the `XmlHttpRequest` type and state as shown in Table 3.5.

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

3.8 Tool Implementation: MUTANDIS

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

To infer JavaScript dynamic invariants, we use our recently developed tool, JSART [72]. For JavaScript code interception, we employ a proxy between the client and the server. This enables us to automatically analyze the content of HTTP responses before they reach the browser. To instrument or mutate the intercepted code, Mozilla Rhino[3] is used to parse JavaScript code to an AST, and back to the source code after the instrumentation or mutation is performed. The execution trace profiler is able to collect trace data from the instrumented application code by exercising the web application under test through one of the following methods: (1) exhaustive automatic crawling using CRAWLJAX [69], (2) the execution of existing test cases, or (3) a combination of crawling and test suite execution.

3.9 Empirical Evaluation

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

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

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

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

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

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

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

Table 3.6: Characteristics of the experimental objects.

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

Table 3.7: Bug severity description.

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

3.9.1 Experimental Objects

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

Table 4.1 presents each application's ID, name, and resource, as well as the static characteristics of the JavaScript code, such as JavaScript lines of code (LOC) excluding libraries, number of functions, and the cyclomatic complexity (CC) across all JavaScript functions in each application.

3.9.2 Experimental Setup

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

We evaluate the efficiency of MUTANDIS in generating non-equivalent mutants (**RQ1**) for the first five applications in Table 4.1. We collect execution traces by instrumenting the custom JavaScript code of each application and executing the instrumented code through automated dynamic crawling. We navigate each application several times with different crawling settings. Crawling settings differ in the number of visited states, depth of crawling, and clickable element types. We inject a single fault at a time in each of these five applications using MUTANDIS. The number of injected faults for each application is 40; in total, we inject 200 faults for the five objects. We automatically generate these mutants from the following mutation categories: (1) variables, (2) branch statements, and (3) JavaScript-specific operators. We then examine each application's behaviour to determine whether the generated mutants are equivalent.

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

To make sure that changes in the applications' behaviour, from which the non-equivalency is determined, are not cosmetic changes we use the bug severity ranks used by Bugzilla, a popular bug tracking system. The description and the rank

associated with each type of bug severity is shown in Table 3.7. We choose non-equivalent mutants from our previously generated mutants (for RQ1). We then analyze the output of the mutated version of the application and assign a bug score according to the ranks in Table 3.7.

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

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

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

Unfortunately, no test suites are available for the first five applications. Thus, to address **RQ3**, we run our tool on the SimpleCart, JQUERY, and WymEditor that come with Qunit[4] test cases. We gather the required execution traces of the SimpleCart library by running its test cases, as this library has not been deployed on a publicly available application. However, to collect dynamic traces of the JQUERY library, we use one of our experimental objects (SameGame), which uses JQUERY as one of its JavaScript libraries. Unlike the earlier case, we include the JQUERY library in the instrumentation step. We then analyze how the application uses different functionalities of the JQUERY library using our approach. The execution traces of the WymEditor are collected by crawling the application. We generate 120 mutants for each of the three experimental objects. Mutated statements, which are not executed by the test suite are excluded. After injecting a fault using MUTANDIS, we run the test cases on the mutated version of each application. We determine the usefulness of our approach based on (1) the number of non-equivalent generated mutants, and (2) the number of non-equivalent *surviving* mutants. A non-equivalent surviving mutant is one that is neither killed nor equivalent, and is an indication of the incompleteness of the test cases. The presence of such mutants can help testers to improve the quality of their test suite. For mature test suites, we expect the number of non-equivalent surviving mutants to be low [57]. We further compare MUTANDIS against random mutation testing to evaluate the effect of

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

Table 3.8: Mutants generated by MUTANDIS.

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

our approach on the stubbornness of the generated mutants. Stubborn mutants are non-equivalent mutants that remain undetected by a high quality test suite [109].

3.9.3 Results

Generated Non-Equivalent Mutants (RQ1)

Table 3.8 presents our results for the number of non-equivalent mutants and the severity of the injected faults using MUTANDIS. For each web application, the table shows the number of mutants, number of equivalent mutants, the number of non-equivalent mutants, the percentage of equivalent mutants, and the average bug severity as well as the percentage of the severity in terms of the maximum severity level.

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

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 these equivalent mutants generated by MUTANDIS originate from the branch mutation category. The reason is that in our current approach, branch expressions are essentially ranked according to the variables used in their expressions without considering whether mutating the expression

changes the actual boolean outcome of the whole expression (e.g.; `if(trueVar || var){...}` where the value of `trueVar` is always `true`, and thus mutating `var` to `!var` does not affect the boolean outcome of the expression). We further notice cases in our experimental objects where the programmer writes essentially unused hard-coded branch expressions. For instance, in Tunnel, we observed a couple of `return true/false` statements at exit point of the functions that have high *FunctionRank* and cyclomatic complexity value. However, the returned value is never used by the caller function and hence, mutating the return boolean value as part of branch mutation generates an equivalent mutant. This is the main reason that we observe 10% of equivalent mutants (the highest in Table 3.8) for the Tunnel application. Moreover, we notice that certain types of mutation operators affect the number of equivalent mutants. For example for a number of mutations we observe that replacing $>=$ ($<=$) sign with $>$ ($<$) keeps the program's behaviour unchanged since either the lower/upper bound is never reached or the programmer specify extra bounds checking before returning the final value.

Fault Severity of the Generated Mutants. The fault severity of the injected faults is also presented in Table 3.8. We computed the percentage of the bug severity as the ratio of the average severity rank to the maximum severity rank (which is 5). As shown in the table, the average bug severity rank across all applications is 3.72 (bug severity percentage is 74.4% on average). We observed only a few faults with trivial severity (e.g; cosmetic changes). 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 [79]. Therefore, it is very rare for web applications to have type 5 errors, and hence the maximum severity rank is often 4.

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

Moreover, we noticed that the careful choice of a variable for mutation is also as important as the function selection. For example, in the SameGame applica-

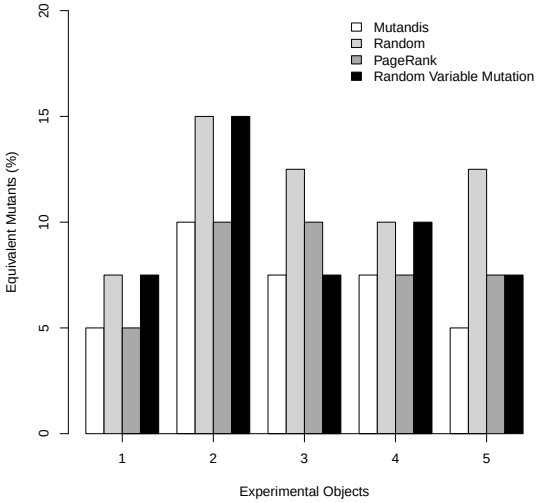


Figure 3.4: Equivalent Mutants (%) generated by MUTANDIS, random, PageRank, and random variable selection.

tion, the `updateBoard` function is responsible for redrawing the game board each time a cell is clicked. Although `updateBoard` is ranked as an important function according to its *FunctionRank*, there are two variables within this function that have high usage frequency compared to other variables. While mutating either of these variables causes major loss of functionality, selecting the remaining ones for mutation either has no effect or only marginal effect on the application's behaviour. Furthermore, we observed that the impact of mutating variables that are part of the invariants as well as the variables with high usage frequency can severely affect the application's behaviour. This indicates that both invariants and usage frequency play a prominent role in generating faults that cause major loss of functionality, thereby justifying our choice of these two metrics for variable selection (Section 3.6).

Effectiveness of *FunctionRank* and selective variable mutation (RQ2)

The results obtained from MUTANDIS, random mutation, *PageRank*, and random variable mutation in terms of the percentage of equivalent mutants and bug severity rank are shown in Figure 3.4 and Figure 3.5, respectively.

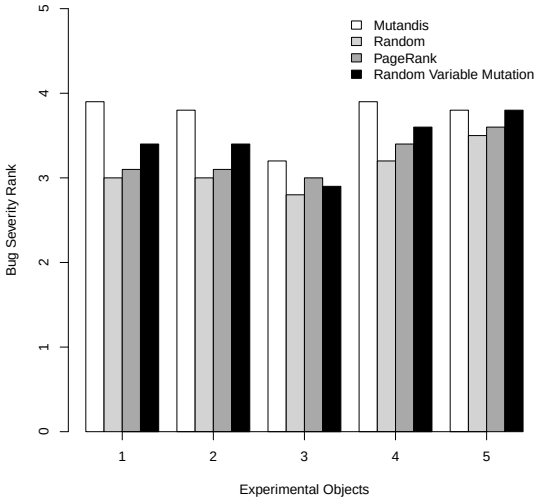


Figure 3.5: Bug Severity Rankd (Avg) achieved by MUTANDIS, random, PageRank, and random variable selection.

As shown in Figure 3.4, the percentage of equivalent mutants generated by MUTANDIS is always less than or equal to the ones generated by the other three approaches. Not surprisingly, random mutation obtains the largest percentage of equivalent mutants (ranges from 7.5–15%). This indicates that our selective variable mutation plays a more prominent role in reducing the percentage of equivalent mutants generated by MUTANDIS.

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

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

We observed that for three applications (ID=1, 2, 4) the main reason behind the reduction in the number of equivalent mutants is the use of selective variable mutation, as by replacing selective variable mutation with random mutation, the percentage of equivalent mutants significantly increases (ranges from 33–50% increment) For these applications, we observed that although high rank functions are selected for mutation, modifying a non-behavioural affecting part of the selected

function's code (i.e., a useless branch or variable) results in generating an equivalent mutant. Therefore, the choice of the variable or branch to mutate is very important.

However, for application with ID 3 (GhostBusters), *FunctionRank* plays a prominent role in reducing the number of equivalent mutants. Figure 3.4 shows that for this application the percentage of equivalent mutants becomes the same as MUTANDIS, when we use random variable mutation coupled with *FunctionRank*. We observed that in the aforementioned application, major variables in the program have high usage frequency. Moreover, these variables are shared among detected invariants, thus making the selection of a specific variable for mutation less effective compared to other applications. For the last application (ID 5), we observed that *FunctionRank* and selective variable mutation are both effective in terms of generating non-equivalent mutants.

Figure 3.5 compares the severity of the injected faults according to the ranks provided in Table 3.7. The results show that MUTANDIS achieves the highest rank among the other approaches. Our mutation generation technique increases the criticality of the injected faults by 20% in comparison with random mutation approach.

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

We further noticed that using the proposed selective variable mutation increases the bug severity by 9% on average. While this indicates the importance of using the proposed variable mutation technique, it reveals that our rank-based function selection technique plays a more prominent role in increasing the severity degree of the injected faults compared to our variable selection strategy. For example, in application with ID 2 (Tunnel), function `updateTunnel` contains the main logic of the application, and it is among the top-ranked functions. Since `updateTunnel` is significantly used throughout the application' execution as its high rank indicates, modifications to the variables of the function affects the expected behaviour of the application, and cause the application to show more severe bugs. Our function ranking technique is able to guide the mutation process towards selecting `updateTunnel` function, and thus increasing the overall bug

Table 3.9: Mutation score computed for SimpleCart, JQUERY, and WymEditor.

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

severity degree. On the other hand, more than 90% of the local and global variables used in function `updateTunnel` are involved with crucial reading and writing of properties. While mutating such important variables generates non-equivalent mutants, it will not significantly improve the criticality of the injected faults *among the non-equivalent mutants* compared to random selection of variables. This implies that our variable selection strategy plays a more prominent role in generating non-equivalent mutants rather than increasing the severity degree of the mutation.

Assessing Existing Test Cases (RQ3)

The results obtained from analyzing the mutants generated by MUTANDIS on the test cases of SimpleCart, JQUERY library, and WymEditor are presented in Table 3.9. The columns under "MUTANDIS", and "Random" present the results obtained by using our approach and random mutation generation respectively. The table shows the number of test cases, branch coverage achieved by the test suite, number of mutants, number of equivalent mutants, number of non-equivalent mutants, number of mutants detected by the test suite (killed mutants), the percentage of non-equivalent mutants and the equivalent mutants, the percentage of non-equivalent surviving mutants, and the mutation score. To compute the percentage of equivalent mutants in presence of the test suite, we follow the guidance suggested by [97], where, $Equiv(\%) = \frac{\#Equiv}{\#TotalMutants - \#Killed} \times 100$. Similarly, the percentage of non-equivalent mutants is: $Non\text{-}Equiv(\%) = \frac{\#Non\text{-}Equiv}{\#TotalMutants - \#Killed} \times 100$

The percentage of non-equivalent surviving mutants is: $\frac{\#NonEquivSurvivingMutants}{\#TotalNonEquivMutants} \times 100$.

Mutation score is used to measure the effectiveness of a test suite in terms of its ability to detect faults [106]. 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.

Quality of test suites. The test suites of both JQuery and WymEditor are frequently updated in response to issues raised by the users. Both JQUery and WymEditor have around 71% branch coverage. This points to the overall high quality of the test cases considering how difficult it is to write unit-level test cases for JavaScript code. Note that despite the low branch coverage of SimpleCart, we gather execution traces of this application based on the available test suite. Therefore, the process of mutation generation is performed according to the executed part of the application from the test suite point of view. We also observed that for the three applications in Table 3.9, a substantial percentage of uncovered branches are related to check for different browser settings (i.e., running the application under IE, FireFox, etc).

Surviving mutants. As shown in the table, less than 30% of the mutants generated by MUTANDIS are equivalent. SimpleCart achieves a mutation score of 67, which means there is much room for test case improvement in this application. For SimpleCart, we noticed that the number of non-equivalent, surviving mutants in the branch mutation category is more than twice the number in the variable mutation category. This shows that the test suite was not able to adequately examine several different branches in the SimpleCart library, possibly because it has a high cyclomatic complexity (Table 4.1). 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:

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

This again points to the importance of *FunctionRank* in test case adequacy

assessment.

As far as RQ3 is concerned:

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

Stubbornness of the generated mutants. Comparing the percentage of equivalent mutants as well as surviving non-equivalent mutants generated by MUTANDIS to those generated by random mutation in Table 3.9, reveals that while our approach decreases the percentage of equivalent mutants (55% on average), it *does not negatively affect the stubbornness of the mutants*. To better show the effectiveness of MUTANDIS in decreasing the number of equivalent mutants, we compute odds ratio, which is a useful measure of effect size for categorical data [64]; the odds of non-equivalent mutants generated by approach M is computed as $odds_{Non\text{-}Equiv\,in\,M} = \frac{\#Non\text{-}Equiv_M - \#killed_M}{\#Equiv_M}$.

Regarding our results, $odds\,ratio_{Non\text{-}Equiv} = \frac{odds_{Non\text{-}Equiv\,in\,Mutandis}}{odds_{Non\text{-}Equiv\,in\,Random}} = 2.6$, which is the odds of non-equivalent mutants generated by MUTANDIS divided by the odds of non-equivalent mutants using random mutation generation. This indicates that the odds of non-equivalent mutants generated by MUTANDIS is 2.6 times higher than the random mutation strategy. We similarly measure the $odds\,ratio_{killed}$ for the number of killed mutants. The $odds\,ratio_{killed}$ of 0.98 indicates that compared with random mutation generation, our approach does not sacrifice stubbornness of the mutants. We further discuss the stubbornness of the mutants in Section 5.5.4.

3.10 Discussion

3.10.1 Stubborn Mutants

The aim of our mutation testing approach is to guide testers towards potentially error-prone parts of the code while easing the burden of handling equivalent mutants by reducing the number of such mutations. However, reducing the number of equivalent mutants might imply a decrease in the number of generated stubborn (or hard-to-kill) mutants, which are particularly useful for test adequacy assessment. Our initial results indicate that while the proposed guided approach reduces the number of equivalent mutants, it does not negatively affect the number of stub-

born mutants generated. This finding is in line with a recent empirical study [109], in which no precise correlation was found between the number of equivalent mutants and stubborn mutants. However, we acknowledge that our finding is based on preliminary results and more research in this direction is needed.

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

3.10.2 Type and Location of Operators

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

- Variable mutations that happen in the body of conditional statements with more than one nested statement, where the conditions are involved with both variable as well as DOM related expressions. To satisfy such conditions, not only the variables should hold proper values, but also the proper structure as well as the properties of the involved DOM elements are required to be in place. This intertwined interaction limits the input space to only a few and challenging ones that are capable of satisfying the condition.

- Replacing the prefix unary operators with postfix unary operators, e.g., `++variable` to `variable++`.

- Replacing the logical operators in conditional statements when the statement contains more than one different logical operator (e.g., `if(A && B || C){...}` to `if(A && B && C){...}`).

- Swapping `true/false` in conditional statements when the statement contains more than two conditions (e.g., `if(A && B && C){...}` to `if(A && !B && C){...}`).

- Removing a parameter from a function call where the function contains more than three parameters.

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

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

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

3.10.3 Characteristics of JavaScript Functions

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

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

usage scenarios. Therefore, the mutation that occurs in these functions remains undetected unless the tester (1) correctly identifies all possible scopes from which the function can be invoked, and (2) associates each invocation with proper test oracles that are relevant to the value of `this`.

Second feature is function variadicity, meaning that a JavaScript function can be invoked with an arbitrary number of arguments compared to the function's static signature, which is common in web applications [91]. For example, if a function is called without passing all the expected parameters, the remaining parameters are set to `undefined`, and thus the function exhibits a different behaviour. Note that in cases where the programmer uses the same implementation of a given function for the purpose of different functionalities, the function achieves a high rank value according to our ranking mechanism since the function is executed several times from different scopes of the application. Testing the expected behaviour of all the possible functionalities is quite challenging, since invoking a particular functionality is often involved with triggering only a specific sequence of events capable of taking the application to the proper state. While the code coverage of the function is the same among different usage scenarios, the mutated statement remains unkilled unless a proper combination of test input and oracle is used. We believe that if our guided approach takes into account such particular usages of a function, which are barely exposed, it can reduce the number of equivalent mutants while increasing the number of hard-to-kill mutants, which forms part of our future work.

3.10.4 Threats to Validity

An external threat to the validity of our results is the limited number of web applications we use to evaluate the usefulness of our approach in assessing existing test cases (RQ4). Unfortunately, few JavaScript applications with up-to-date test suites are publicly available. Another external threat to validity is that we do not perform a quantitative comparison of our technique with other mutation techniques. However, to the best of our knowledge, there is no mutation testing tool available for JavaScript, which limits our ability to perform such comparisons. A relatively low number of generated mutants in our experiments is also a threat to validity. However, detecting equivalent mutants is a labour intensive task. For example it

took us more than 4 hours to distinguish the equivalent mutants for JQUERY in our study. In terms of internal threat to validity, we had to manually inspect the application's code to detect equivalent mutants. This is a time intensive task, which may be error-prone and biased towards our judgment. However, this threat is shared by other studies that attempt to detect equivalent mutants. As for the replicability of our study, MUTANDIS and all the experimental objects used are publicly available, making our results reproducible.

3.11 Related Work

A large body of research has been conducted to turn mutation testing into a practical approach. To reduce the computational cost of mutation testing, researchers have proposed three main approaches to generate a smaller subset of all possible mutants: (1) *mutant clustering* [59], which is an approach that chooses a subset of mutants using clustering algorithms; (2) *selective mutation* [19, 78, 110], which is based on a careful selection of more effective mutation operators to generate less mutants; and (3) *higher order mutation* (HOM) testing [60], which tries to find rare but valuable higher order mutants that denote subtle faults [61].

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

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

According to the taxonomy suggested by Madeyski et al. [64], there are three main categories of approaches that address the problem of equivalent mutants: (1) detecting equivalent mutants, (2) avoiding equivalent mutant generation, and (3) suggesting equivalent mutants. As far as equivalent mutant detection techniques are concerned, the most effective approach is proposed by Offutt and Pan [82, 83],

which uses constraint solving and path analysis. The results of their evaluation showed that the approach is able to detect on average the 45% of the equivalent mutants. However, these solutions are involved with considerable amount of manual effort and are error-prone. Among equivalent detection methods, program slicing has also been used in equivalent mutants detection [54]. The goal there is to guide a tester in detecting the locations that are affected by a mutant. Such equivalent mutant detection techniques are orthogonal to our approach. If a mutation has been statically proven to be equivalent, we do not need to measure its impact on the application's expected behaviour and we focus only on those that cannot be handled using static techniques. Moreover, static techniques are not able to fully address unpredictable and highly dynamic aspects of programming languages such as JavaScript.

Among avoiding equivalent mutant generation techniques, Adamopoulos et al. [13] present a co-evolutionary approach by designing a fitness function to detect and avoid possible equivalent mutants. Domínguez-Jiménez et al. [35] 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 [63]. An important limitation of these approaches is that the generated mutant needs to be executed against the test suite to compute its fitness function. In contrast, our approach avoids generating equivalent mutants in the first place, thereby achieving greater efficiency. Bottaci [25] presents a mutation analysis technique based on the available type information at run-time to avoid generating incompetent mutants. This approach is applicable for dynamically typed programs such as JavaScript. However, the efficiency of the technique is unclear as they do not provide any empirical evaluation of their approach. Gligoric et al. [46] conduct the first study on performing selective mutation to avoid generating equivalent mutants in concurrent code. The results show that there are important differences between the concurrent mutation and sequential mutation operators. The authors show that sequential and concurrent mutation operators are independent, and thus they pro-

pose sets of operators that can be used for mutation testing of concurrent codes. While we also make use of a small set of mutation operators, we aim to support sequential programs.

Among the equivalent mutant suggestion techniques, Schuler et al. [98] suggest possible equivalent mutants by checking invariant violations. They generate multiple mutated versions and then classify the versions based on the number of violated invariants. The system recommends testers to focus on those mutations that violate the most invariants. In a follow-up paper [97], they extend their work to assess the role of code coverage changes in detecting equivalent mutants. Kintis et al. [62] present a technique called I-EQM to dynamically isolate first order equivalent mutants. They further extend the coverage impact approach [97] to classify more killable mutants. In addition to coverage impact, the classification scheme utilizes second order mutation to assess first order mutants as killable. To generate mutants, they utilize Javalanche [97]. Our work is again different from these approaches in the sense that instead of classifying mutants, we avoid generating equivalent mutants a priori by identifying behaviour-affecting portions of the code.

Deng et al. [34] implement a version of statement deletion (SDL) mutation operator for Java within the muJava mutation system. The design of SDL operator is based on a theory that performing mutation testing using only one mutation operator can result in generating effective tests. However, the authors cannot conclude that SDL-based mutation is as effective as selective mutation, which contains a sufficient set of mutation operators from all possible operators. Therefore, they only recommend for future mutation systems to include SDL as a choice, which we have already taken into account in this paper.

Ayari et al. [18] and Fraser et al. [43] apply search based techniques to automatically generate test cases using mutation testing for Java applications. Harman et al. [52] propose SHOM approach which combines dynamic symbolic execution and Search based software testing to generate strongly adequate test data to kill first and higher order mutants for C programs. However, all these approaches make use of mutation testing for the purpose of test case generation, and thus to generate mutants they rely on the available mutation testing frameworks.

Zhang et al. [111] present FaMT approach which incorporates a family of techniques for prioritizing and reducing tests to reduce the time required for mutation

testing. FaMT is designed based on regression test prioritization and reduction. Our approach is orthogonal to this work as our goal is to optimize the mutant generation to produce useful mutants, which can later be executed against the test suite. Our mutation generation approach can be combined with this technique to further speed up mutation testing.

Bhattacharya et al. [22] propose *NodeRank* to identify parts of code that are prone to bugs of high severity. Similar to our work, *NodeRank* uses the *PageRank* algorithm to assign a value to each node in a graph, indicating the relative importance of that node in the whole program according to the program's static call graph. The authors empirically show that such important portions of the code require more maintenance and testing effort as the program evolves. In our approach we propose a new metric, $FunctionRank$, which takes advantage of dynamic information collected at execution time for measuring the importance of a function in terms of the program's behaviour. Weighting the ranking metric with call frequencies as we do makes it more practical in web application testing, as the likelihood of exercising different parts of the application can be different. Further, to the best our knowledge, we are the first to apply such a metric to mutation testing.

Chapter 4

Generating Test Cases with Oracles for JavaScript Applications

4.1 Abstract

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 test JavaScript-based applications. Current web test automation techniques target the generation of event sequences, but they ignore testing the JavaScript code at the unit level. Further they either ignore the oracle problem completely or simplify it through generic soft oracles such as HTML validation and runtime exceptions. We present a framework [74, 75] to automatically generate test cases for JavaScript applications at two complementary levels, namely events and individual JavaScript functions. Our approach employs a combination of function coverage maximization and function state abstraction algorithms to efficiently generate test cases. In addition, these test cases are strengthened by automatically generated mutation-based oracles. We empirically evaluate the implementation of our approach, called JSEFT, to assess its efficacy. The results, on 13 JavaScript-based applications, show that the generated test cases achieve a coverage of 68% and that JSEFT can detect injected JavaScript and DOM faults with a high accuracy (100% precision, 70% recall). We also find that JSEFT outperforms an existing JavaScript test automation framework both in terms of coverage and detected faults.

4.2 Introduction

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 time-consuming.

Researchers have recently developed automated test generation techniques for JavaScript-based applications [17, 65, 66, 70, 96]. However, current web test generation techniques suffer from two main shortcomings:

1. Target the generation of *event sequences*, that can potentially miss the portion of code-level JavaScript faults.
2. Either ignore the oracle problem altogether or simplify it through generic *soft oracles*.

To address these two shortcomings, we propose an automated test case generation technique for JavaScript applications.

Our approach, called JSEFT (JavaScript Event and Function Testing) operates through a three step process. First, it dynamically explores the event-space of the application using a *function coverage maximization* method, to infer a test model. Then, it generates test cases at two complementary levels, namely, DOM event and JavaScript functions. Our technique employs a novel *function state abstraction* algorithm to minimize the number of function-level states needed for test generation. Finally, it automatically generates test oracles, through a mutation-based algorithm.

This work makes the following main contributions:

- An automatic technique to generate test cases for JavaScript functions and events.
- A combination of function converge maximization and function state abstraction algorithms to efficiently generate unit test cases;
- A mutation-based algorithm to effectively generate test oracles, capable of detecting regression JavaScript and DOM-level faults;
- The implementation of our technique in a tool called JSEFT, which is publicly available [6];

```
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 });
```

Figure 4.1: JavaScript code of the running example.

- An empirical evaluation to assess the efficacy of JSEFT using 13 JavaScript applications.

The results of our evaluation show that on average (1) the generated test suite by JSEFT achieves a 68% JavaScript code coverage, (2) compared to Artemis, a feedback-directed JavaScript testing framework [17], JSEFT achieves 53% better coverage, and (3) the test oracles generated by JSEFT are able to detect injected faults with 100% precision and 70% recall.

4.3 Challenges and Motivation

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

Figure 4.1 presents a snippet of a JavaScript game application that we use as a running example throughout the thesis. This simple example uses the popular jQuery library [4] 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 may 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 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.

4.4 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 complementary 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.

An overview of the technique is depicted in Figure 4.2. At a high level, our approach is composed of three main steps:

1. In the first step (Section 4.4.1), 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) [70], capturing the explored dynamic DOM states and event-based transitions between them.

2. In the second step (Section 4.4.2), 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 4.6). 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.

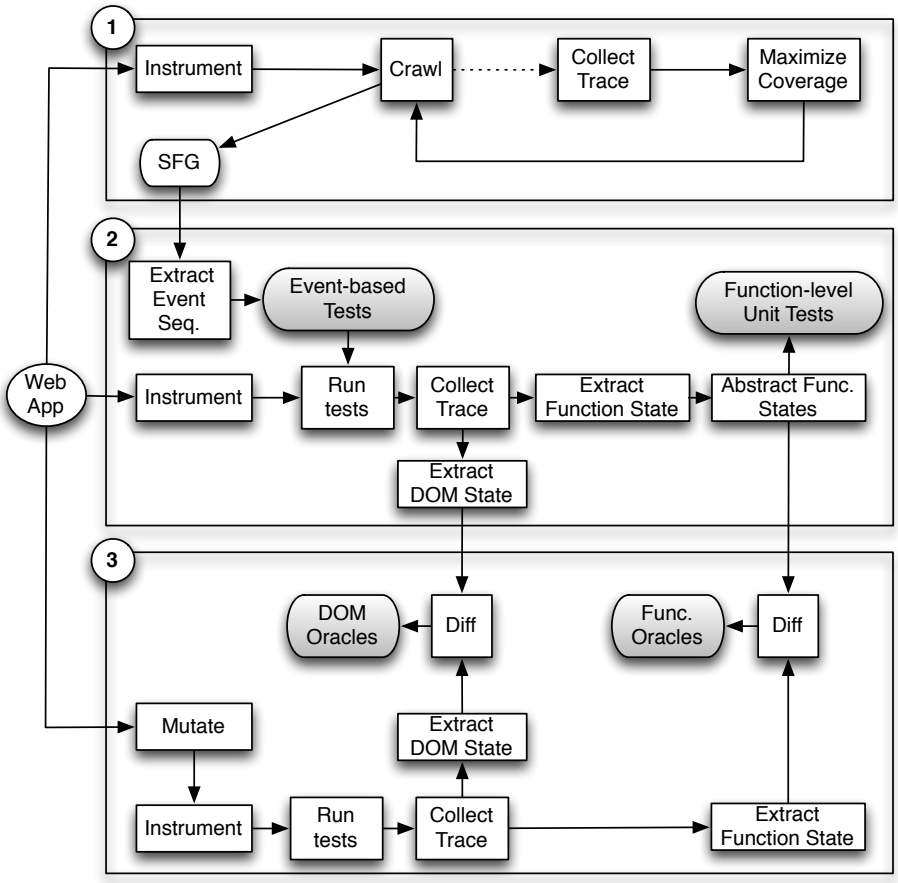


Figure 4.2: Overview of our test generation approach.

4.4.1 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 unexecuted 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. 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 4.1, 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` 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.

4.4.2 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 4.6.

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(){...}}`);
4. Prototypes (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 4.1, at the entry point of `setDim`, we log 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 5.3. 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* [9] 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 ID `endCell` in Line 22. We further include in our execution

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

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 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 4.6).

Function State Abstraction. As mentioned in Section 5.3, 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 negatively 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

Algorithm 2: 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** BRNCOVLNS$[st_i] \neq$ BRNCOVLNS$[StSet]_{l=1}^{L}$ **then**
4 $StSet_{L+1} \leftarrow st_i$
5 $L++$
end
6 **else**
7 $StSet_l \leftarrow st_i \cup StSet_l$
end
8 $K = L+1$; $StSet_K \leftarrow \emptyset$
9 **if** DOMPROPS$[st_i] \neq$ DOMPROPS$[StSet]_{k=L+1}^{K}$ || $RetType[st_i] \neq$ RETTYPE$[StSet]_{k=L+1}^{K}$ **then**
10 $StSet_{K+1} \leftarrow st_i$
11 $K++$
end
12 **else**
13 $StSet_k \leftarrow st_k \cup StSet_k$
end
end
14 **while** $StSet_{K+L} \neq \emptyset$ **do**
15 $SelectedSt \leftarrow$ SELECTMAXST$(st_i | st_i \cap StSet_{j=1}^{K+L})$
16 $AbsStates$.ADD$(SelectedSt)$
17 $StSet_{K+L} \leftarrow StSet_{K+L} - SelectedSt$
end
18 return *AbsStates*
end

example in Figure 4.1, executing either of the 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 2 shows our function state abstraction algorithm. The algorithm first collects covered branches of individual functions per entry state (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* [30], 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.

4.4.3 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 4.2. 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 [33], or

to generate test cases that kill mutants [43]. 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 would only need to examine a small set of effective assertions to verify the correctness of the generated oracles. Algorithm 3 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 3 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

Algorithm 3: 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 ← INSTRUMENT(App)
   begin
2      while GenMuts < n do
3          foreach EvSeq ∈ SFG do
4              OnEvDomSt ← Trace.GETONEVDOMST(Ev ∈ EvSeq)
5              AfterEvDomSt ← Trace.GETAFTEREVDOMST(Ev ∈ EvSeq)
6              AccdDomProps ← GETACCDDOMNDS(OnEvDomSt)
7              EquivalentDomMut ← true
8              while EquivalentDomMut do
9                  MutDom ← MUTATEDOM(AccdDomProps, OnEvDomSt)
10                 ChangedSt ← EvSeq.EXECEVENT(MutDom)
11                 Diff_{ChangedSt,AfterEvDomSt} ← DIFF(ChangedSt, AfterEvDomSt)
12                 if Diff_{ChangedSt,AfterEvDomSt} ≠ ∅ then
13                     EquivalentDomMut ← false
14                     DomAssert_i = Diff_{ChangedSt,AfterEvDomSt}
15                     DomAsserts_{Ev,AfterEvDomSt} = ⋂DomAssert_i
                   end
               end
16             AbsFcSts ← Trace.GETABSFCSTS()
17             EquivalentCodeMut ← true
18             while EquivalentCodeMut do
19                 MutApp ← MUTATEJSCODE(App)
20                 MutFcSts ← EvSeq.EXECEVENT(MutApp)
21                 foreach FcEntry ∈ AbsFcSts.GETFCENTRIES do
22                     FcExit ← AbsFcSts.GETFCEXIT(FcEntry)
23                     MutFcExit ← MutFcSts.GETMUTFCEXIT(FcEntry)
24                     Diff_{FcExit,MutFcExit} ← DIFF(FcExit, MutFcExit)
25                     if Diff_{FcExit,MutFcExit} ≠ ∅ then
26                         EquivalentCodeMut ← false
27                         FcAssert_i = ⋂Diff_{FcExit,MutFcExit}
28                         FcAsserts_{FcEntry} = ⋃FcAssert_i
                       end
                   end
               end
           end
       end
29     return {FcAsserts, DOMAsserts}
   end
```

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 3). 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 4.3 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 4.3 shows the corresponding assertion. Furthermore, Assuming that the DOM mutation makes `currentDim`≤ 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 4.3.

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 3), we identify all changed DOM elements and generate assertions for these elements. Note that any changes detected by the DIFF operator (line 12 in Algorithm 3) is an indication that the corresponding DOM mutation is *not equivalent* (line 13); if no change is

```
1 @Test
2 public void testCase1(){
3   WebElement divElem=driver.findElements(By.id("divElem"));
4   divElem.click();
5   int endCellHeight=driver.findElements(By.id("endCell")).getSize().↩
        height;
6   assertEquals(endCellHeight, 30);
7   WebElement startCell=driver.findElements(By.id("startCell"));
8   startCell.click();
9   boolean exists=driver.findElements(By.id("startCell")).size!=0;
10  assertTrue(exists);
11  int startCellHeight=driver.findElements(By.id("startCell")).getSize()↩
        .height;
12  assertEquals(startCellHeight, 50);
13 }
```

Figure 4.3: Generated SELENIUM test case.

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 [73]. 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 4.4.2.

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

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

Figure 4.4: Generated QUNIT test case.

application with the traces we obtained from the modified versions (DIFF in line 24 in Algorithm 3). 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 called multiple times. In this case, we need to combine assertions to make sure that the generated test cases do not mistakenly fail. $FcAsserts_{FcEntry}$ 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 4.1, and '+' sign is mutated to '-' in line 20 of the running example in Figure 4.1, the mutation affects the global variable `currentDim`, `height` property of element with ID `endCell`, and the returned value of the function `setDim`. Figure 4.4 shows a QUNIT test case for `setDim` function according to this mutation with the generated assertions.

4.4.4 Tool Implementation

We have implemented our JavaScript test and oracle generation approach in an automated tool called JSEFT. The tool is written in Java and is publicly available for download [6]. 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 JSEFT extends and builds on top of the event-based crawler, CRAWLJAX [69], with random input generation enabled for form inputs. As mentioned before, to mutate JavaScript code, we use our recently developed mutation testing tool, MUTANDIS [73]. 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 [9], capable of testing any generic JavaScript code.

4.5 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 JSEFT in generating test cases with high coverage?

Table 4.1: Characteristics of the experimental objects.

ID	Name	LOC	URL
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.ccconthewebb.org/peggame.htm
5	BunnyHunt	580	http://www.themaninblue.com/experiment/BunnyHunt/
6	AjaxTabs	592	https://github.com/amazingSurge/jquery-tabs/
7	NarrowDesign	1,005	http://www.narrowdesign.com
8	JointLondon	1,211	http://www.jointlondon.com
9	FractalViewer	1,245	onecm.com/projects/canopy/
10	SimpleCart	1,900	https://github.com/wojodesign/simplecart-js/
11	WymEditor	3,035	http://www.wymeditor.org
12	TuduList	1,963	http://tudu.ess.ch/tudu
13	TinyMCE	26,908	http://www.tinymce.com

RQ2 How capable is JSEFT of generating test oracles that detect regression faults?

RQ3 How does JSEFT compare to existing automated JavaScript testing frameworks?

JSEFT and all our experimental data are available for download [6].

4.5.1 Objects

Our study includes thirteen JavaScript-based applications in total. Table 4.1 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.

4.5.2 Setup

To address our research questions, we provide the URL of each experimental object to JSEFT. Test cases are then automatically generated by JSEFT. We give JSEFT 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 [5], an open-source tool for measuring JavaScript code coverage. We report the average results over five runs to account for the non-determinism behaviour that stems from crawling the application. 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 2), and (3) evaluate the effectiveness of applying mutation techniques (Algorithm 3) to reduce the number of assertions generated.

Test Oracles (RQ2). To evaluate the fault finding capability of JSEFT (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 4.6), 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 JSEFT 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 actual faults: $\frac{TP}{TP+FP}$

Recall is the rate of actual injected faults that the tool finds: $\frac{TP}{TP+FN}$

Table 4.2: Results showing the effects of our **function coverage maximization**, **function state abstraction**, and **mutation-based oracle generation** algorithms.

	St. 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	-	-

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 JSEFT performs with respect to existing JavaScript test automation tools, we compare its coverage and fault finding capability to that of Artemis [17]. Similar to JSEFT, 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 [17]. 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 JSEFT.

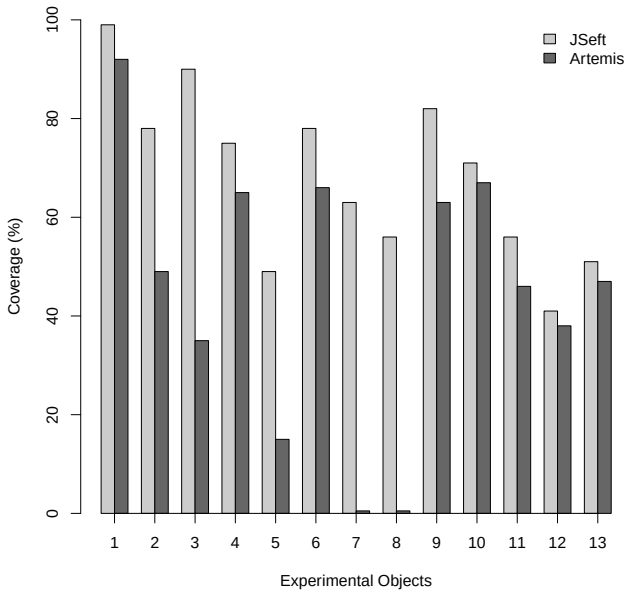


Figure 4.5: Statement coverage achieved.

4.5.3 Results

Test Case Generation (RQ1). Figure 4.5 depicts the statement coverage achieved by JSEFT for each application. The results show that the test cases generated by JSEFT achieve a coverage of 68.4% on average, ranging from 41% (ID 12) up to 99% (ID 1). We investigated why JSEFT 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 JSEFT. 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 JSEFT is only able to cover a portion of these applications in the limited time it had available.

Table 4.2 columns under "St. 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

Table 4.3: Fault detection.

App ID	# Injected Faults	JSEFT						Artemis	
		#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

all 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 4.2 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 4.4.2.

The last two columns of Table 4.2, under "Oracles", present 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 4.3 presents the results on the fault finding

capabilities of JSEFT. The table shows the total number of injected faults, the number of false negatives, false positives, true positives, and the precision and recall of JSEFT.

JSEFT achieves 100% precision, meaning that all the detected faults reported by JSEFT are real faults. *In other words, there are no false-positives.* This is because the assertions generated by JSEFT are all stable i.e., they do not change from one run to another. However, the recall of JSEFT is 70% on average, and ranges from 48 to 100%. This is due to false negatives, i.e., missed faults by JSEFT, 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 the 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 4.5 shows the code coverage achieved by both JSEFT and Artemis on the experimental objects running for the same amount of time, i.e., 10 minutes. The test cases generated by JSEFT achieve 68.4% coverage on average (ranging from 41–99%), while those generated by Artemis achieve only 44.8% coverage on average (ranging from 0–92%). Overall, the test cases generated by JSEFT achieve 53% more coverage than Artemis, which points to the effectiveness of JSEFT in generating high coverage test cases. Further, as can be seen in the bar plot of Figure 4.5, for all the applications, the test cases generated by JSEFT 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 additional 10 minutes for these applications (i.e., 20 minutes in total). Even then, neither application completes under Artemis.

Table 4.3 shows the precision and recall achieved by JSEFT and Artemis. With respect to fault finding capability, unlike Artemis that detects only generic faults such as runtime exceptions and W3C HTML validation errors, JSEFT 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, JSEFT achieves five-fold higher recall (70% on average) compared with Artemis (12.8% recall on average).

4.5.4 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 failed to make sure that the injected fault was indeed responsible for the assertion failure. Regarding the reproducibility of our results, JSEFT and all the applications used in this study are publicly available, thus making the study replicable.

4.6 Related Work

Web application testing. Marchetto and Tonella [65] propose a search-based algorithm for generating event-based sequences to test Ajax applications. Mesbah et al. [69] apply dynamic analysis to construct a model of the application's state space, from which event-based test cases are automatically generated. In subsequent work [70], they propose generic and application-specific invariants as a form of automated soft oracles for testing AJAX applications. Our earlier work, JSART [72], automatically infers program invariants from JavaScript execution traces and

uses them as regression assertions in the code. Sen et al. [99] 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. [58] 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. [96] 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. Our work is different in two main aspects from these: (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.

Perhaps the most closely related work to ours is Artemis [17], 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 5.5).

Oracle generation. There has been limited work on oracle generation for testing. Fraser et al. [43] 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 [42] relevant pre-conditions on the test inputs and post-conditions on the outputs to ease human comprehension. Differential test case generation approaches [37, 102] 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. [101] 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 [80] dynamic interactions of JavaScript with the DOM.

Chapter 5

Inferring Unit Oracles from GUI Test Cases

5.1 Abstract[1]

Testing JavaScript web applications is challenging due to its complex runtime interaction with the Document Object Model (DOM). Writing unit-level assertions for JavaScript applications is even more tedious as the tester needs to precisely understand the interaction between the DOM and the JavaScript code, which is responsible for updating the DOM. In this work, we propose to leverage existing DOM-dependent assertions in a human-written DOM-based test cases as well as useful execution information inferred from the DOM-based test suite to automatically generate assertions used for unit-level testing of the JavaScript code of the application. Our approach is implemented in a tool called Atrina. We evaluate our approach to assess its effectiveness. The results indicate that Atrina maps DOM-based assertions to the corresponding JavaScript code with high accuracy (99% precision, 90% recall). In terms of fault finding capability, the assertions generated by Atrina outperform human-written DOM-based assertions by 37% on average. It also surpasses the state-of-the-art mutation-based assertion generation technique by 29% on average in detecting faults.

5.2 Introduction

The close relation between the Document Object Model (DOM) and the underlying JavaScript code creates an interactive web application. To check the application's

[1]This work is under preparation.

behaviour from an end-user's perspective, testers often use popular frameworks such as Selenium. Using these frameworks to write DOM-based tests and assertions requires little knowledge about the internal operations performed at the client side code. Rather, the tester needs only basic knowledge of common event sequences to cover important DOM elements to assert. This makes it easier for the tester to write DOM-based test suites. On the other hand, writing unit test assertions at code-level for web applications that have rich interaction with the DOM through their JavaScript code is more tedious. To write unit-level assertions, the tester needs to precisely understand the full range of interaction between the code level operations of a unit and the DOM level operations of a system, and thus may fail to assert the correctness of a particular behaviour when the unit is used as a part of a system.

Our previous findings [75] indicate that DOM-based assertions can potentially miss the related portion of code-level failure, while more fine grained unit-level assertions are capable of detecting such faults. Furthermore, finding the root cause of an error during DOM-based testing can be more expensive than during unit testing. The inherent characteristics of unit and DOM-based tests, indicate that they are complementary and that there is a trade-off in individually using each to detect faults.

Current test generation approaches either produce unit test oracles based on mutation testing techniques [43, 75], or they rely on soft oracles [17]. Mutation-based approaches suffer from high computational cost and equivalent mutants which are syntactically different but semantically are the same as the original application. Soft oracles such as HTML validation and runtime exceptions are also limited in that they fail to capture logical and computational errors. Recently, Milani Fard et al. [39] proposed using the DOM-based test suite of a web application to re-generate assertions for newly detected states through exploring alternative paths of the application. However, the new assertions generated by this technique remain at the DOM-level without considering the relation between the JavaScript code and the DOM. In this work, we propose to exploit an existing DOM-based test suite to generate unit-level assertions at the code-level for applications that highly interact with the DOM through the underlying JavaScript code. We utilize existing DOM-dependent assertions as well as useful execution information inferred from

a DOM-based test suite to automatically generate assertions used for testing individual JavaScript functions.

To the best of our knowledge this work is the first to propose an approach for generating unit-level assertions by using the existing DOM-based test suite of the application. The main contributions of our work include:

- A slicing-based technique to generate unit-level assertions capable of testing JavaScript functions by utilizing existing DOM-based test assertions;
- A technique for selecting effective DOM elements in detecting code-level faults, which remain unchecked in the existing DOM-based test suite;
- An implementation of our approach in a tool, called Atrina;
- An empirical evaluation to assess the efficacy of the approach on four open-source web applications; The results show that the assertions generated by Atrina surpass the fault finding capability of (1) the human-written DOM-based assertions by 37% on average, and (2) the state-of-the-art mutation-based assertion generation technique by 29% on average.

5.3 Motivation

Unlike DOM-based testing, asserting the behaviour of a JavaScript application through unit-level tests requires a tester to check the correctness of several intermediate code-level variables and object properties. The code-level operations are mainly responsible for updating the DOM throughout the application execution. Therefore, a tester needs to analyze the relation between the JavaScript code and the DOM's evolution. We believe DOM-based assertions can be utilized as a guideline to generate unit test assertions at JavaScript code level.

Figure 5.1 presents a snippet of a JavaScript-based shopping cart application as well as a sample DOM-based SELENIUM test case that we will use as a running example. The application's code (a) contains two main functions as follows:

1. `addToCart` is bound to the event handler of DOM elements with class `merchandise`. When the element is clicked, `addToCart` gets the information of the selected merchandise, and sets the quantity of the current available items by updating the `availItems` object. If a valid discount

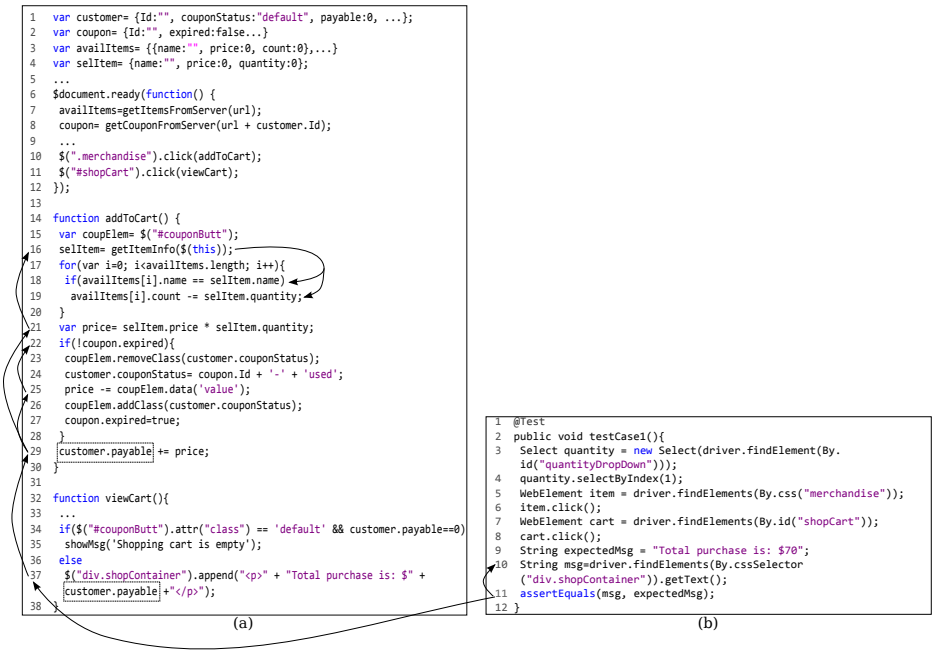


Figure 5.1: Running example (a) JavaScript code, and (b) DOM-based test case. The line from (b) to (a) shows the point of contact between the DOM assertion and the code. The arrow lines in (a) show the backward as well as forward slices between JavaScript statements.

coupon exists, `addToCart` calculates the discount value, and disables the selected coupon button with ID `couponButt` by removing the corresponding class. Finally, `addToCart` updates the payable amount by setting the `payable` property of the `customer` object.

2. `viewCart` is invoked by clicking on a DOM element with ID `shopCart`. The function appends a message to a `div` element with class `shopContainer` including the final payable amount of the customer. If the coupon button with ID `couponButt` is not selected and the payable amount is equal to zero, then the empty cart message is shown.

Let's assume that in line 21 of Figure 5.1(a) `selItem.price`, which is assigned by the original price of the merchandise is 100, and `selItem.quantity` is 1. In line 25, the discount, which is calculated based on the `data` value of the `couponButt` element is 30. The DOM-based assertion in Figure 5.1(b)

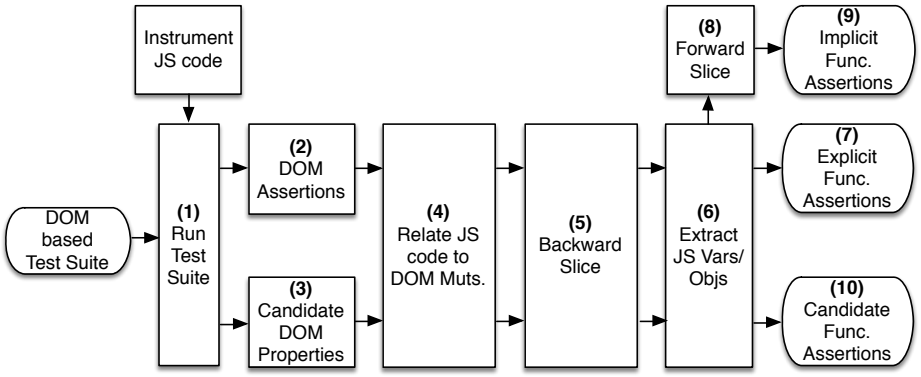


Figure 5.2: Overview of our assertion generation approach.

(line 11) checks the correctness of a text appended to a `div` element with class `shopContainer` containing the final amount payable by the customer, which is equal to 70 in this example. Analyzing the assertion in line 11 of Figure 5.1(b) indicates that the expected value of the assertion is directly influenced by the `payable` property of `customer` object as well as the object's property `coupon.expired` in function `addToCart`. We also infer that a variable in line 16 of Figure 5.1(a) which directly influences the value of `customer.payable`, is also used in updating the value of `availItems.count` in line 19.

Further, by leveraging the execution information obtained from running the DOM-based test case, we can identify DOM's evolution, which are not checked by the existing assertions, but potentially influence the fault finding capability of the test suite. For instance, DOM element with ID `couponButt` is accessed several times in function `addToCart` as well as `viewCart` as the test case in Figure 5.1(b) runs, however it remains unchecked. Since evolution of the `couponButt` DOM element pertains to the underlying JavaScript code, it is important to assert on code statements responsible for changing the aforementioned DOM element.

5.4 Overview of Approach

An overview of our unit-level assertion generation technique is depicted in Figure 5.2. At a high level, our approach generates unit-level assertions by utilizing

Algorithm 4: Oracle Generation

input : Test suite T; The set of test cases $tc_i \in T$
output: The ordered set of oracles *oracles*

begin
1 $trace \leftarrow \text{EXEC}(T)$
2 $domAccss \leftarrow \text{GETDOMACC}(trace)$
3 $freqAccdDOM \leftarrow \emptyset$
4 $\alpha = \frac{1}{\text{READPROPERTIES}(T)}$
5 **for** $dom \in domAccss$ **do**
6 **if** $\text{ACC}(prop_{dom}) \geq \alpha$ **then**
7 $freqAccdDOM \leftarrow dom \cup freqAccdDOM$
end
end
8 **for** $tc_i \in T$ **do**
9 $trace \leftarrow \text{EXEC}(tc_i)$
10 $domAccss \leftarrow \text{GETDOMACC}(trace)$
11 **for** $asstn \in assertions_{tc_i}$ **do**
12 $asserDOMAcc \leftarrow \text{GETDOMACC}(asstn)$
13 $asserDOMMuts \leftarrow \text{GETDOMMUTS}(asserDOMAcc)$
14 **for** $domMut \in asserDOMMuts$ **do**
15 $bwSts \leftarrow \text{GETBWSLICE}(domMut, trace)$
16 $expAsstnRel \leftarrow \text{GETWRVARS}(bwSts)$
17 $fwSts \leftarrow \text{GETFWSLICE}(bwSts, trace)$
18 $impAsstnRel \leftarrow \text{GETWRVARS}(fwSts)$
end
end
19 $cndDOMMuts \leftarrow \text{GETDOMMUTS}(freqAccdDOM)$
20 **for** $domMut \in cndDOMMuts$ **do**
21 $bwSts \leftarrow \text{GETBWSLICE}(domMut, trace)$
22 $cndAsstn \leftarrow \text{GETWRVARS}(bwSts)$
end
23 $explicitAsstn[func]_{f=1}^{n} \leftarrow \text{ACCESSIBLES}([func]_{f=1}^{n}, [expAsstnRel])$
24 $implicitAsstn[func]_{f=1}^{n} \leftarrow \text{ACCESSIBLES}([func]_{f=1}^{n}, [impAsstnRel])$
25 $candidateAsstn[func]_{f=1}^{n} \leftarrow \text{ACCESSIBLES}([func]_{f=1}^{n}, [cndAsstn])$
26 $oracles[func]_{f=1}^{n} \leftarrow \{explicitAsstn \cup implicitAsstn \cup candidiateAsstn\}$
end
27 return $(oracles[func]_{f=1}^{n})$
end

human written DOM-based tests and assertions. Our code level assertions fall in the following three categories: (1) explicit assertions, which are directly inferred from analyzing the manually written DOM-based assertions, (2) implicit assertions, which are indirectly affected by the human written DOM-based assertions, and (3) candidate assertions, which are not considered in the written DOM-based assertions, yet are potentially useful to be checked by the function level test suite. We describe our approach below. The numbers below in parentheses correspond to

those in the boxes of Figure 5.2.

In the first part of our approach we (1) execute the instrumented application by running the existing DOM-based test suite. In this step we gather a detailed execution trace of the application. We then extract (2) DOM-based assertions, and (3) candidate DOM element properties, which are useful DOM properties that can potentially be utilized for the purpose of assertion generation. We (4) identify the initial point of connection between the application's source code and checked DOM element. We (5) calculate the backward slice of the DOM mutating statements to find the entire code blocks that update the checked DOM element. We then (6) extract accessible entities from the obtained statements. Accessible entities form our explicit assertions (7). We further (8) perform a forward slice on the extracted entities to identify statements, that are implicitly affected by such entities. The accessible entities associated with the collected statements form our implicit assertions (9). In addition to explicit and implicit assertions, we also generate candidate assertions (10). Candidate assertions are involved with updating potentially useful DOM element properties, which are not checked in the existing DOM-based assertions. To obtain candidate assertions, we perform step (4), (5), and (6) on the inferred candidate DOM element properties (3).

Our overall unit-level assertion generation is presented in Algorithm 4. In the following sections we describe our technique for extracting DOM related information from the execution (Section 5.4.1), relating DOM mutations to the JavaScript code (Section 5.4.2), and generating unit test assertions (Section 5.4.3).

5.4.1 Extracting DOM-Related Characteristics

The DOM connects a test case to the web application's code. Therefore, we first need to analyze the DOM-based test suite and extract the following pieces of information: (1) DOM-related operations of the existing test suite that may have tight connection with the JavaScript code, and (2) frequently accessed DOM properties, which are potentially influential in improving the fault finding capability of the test suite, but left unchecked in the manually-written test suite.

DOM-Related Operations. Any written test case needs to check the correctness of the application's behaviour. In a DOM-based test case the expected behaviour

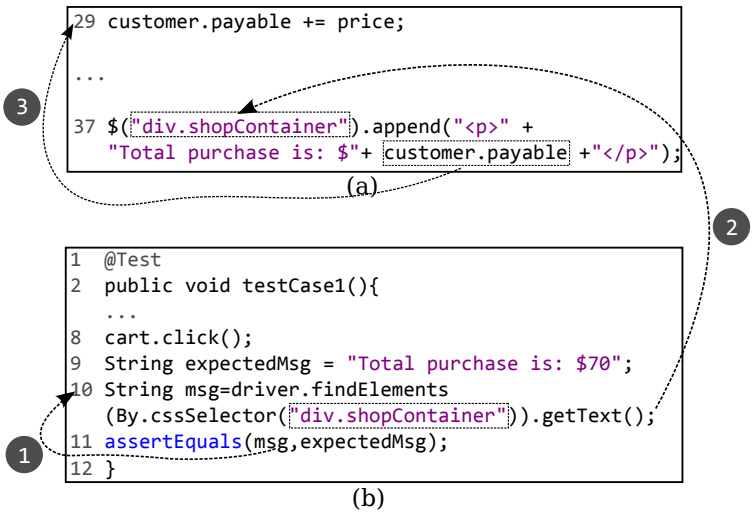


Figure 5.3: Finding (1) intra DOM assertion dependency within the test case (b), (2) inter DOM assertion dependency between (b) DOM-based assertion and (a) the JavaScript code, and (3) the initial point of contact between (b) DOM-based assertion and (a) the JavaScript code.

is checked through DOM-based assertions. DOM-based assertion is defined as $< domProps, expVal >$, where *domProps* consists of one or more DOM element features (e.g. attribute, and/or textual value), and *expVal* is the correct value expected by the assertion. Through the rest of this chapter, we call the DOM element feature as a DOM property. DOM-based assertions play a significant role in our approach as they can guide us towards important portions of the underlying JavaScript code that need to be checked in unit-level assertions. For each DOM-based assertion we find *intra DOM assertion dependency* within the test case.

Definition 1 (Intra DOM Assertion Dependency) *An intra DOM assertion dependency is defined as a three tuple of* $< assert, domElems, domProps >$*, where assert is the intended DOM-based assertion, domElems is the accessed DOM elements in the test case pertaining to the assertion, and domProps is the accessed DOM properties within the assertion.*

GETDOMACC in line 10 of Algorithm 4 retrieves DOM dependencies of the assertion in the test case. Going back to our example in Figure 5.3(b), tracking the assertion in line 11 shows that it has a DOM dependency to a `div` element with class `shopContainer`, which is accessed in line 10. The intra DOM assertion

dependency of the example further shows that the `text` value of the DOM element is compared with the `expectedMsg` in line 11.

We further need to correlate the inferred intra DOM assertion dependency with the application's code. We call the correlation between the DOM-based assertion and the application's code as *inter DOM assertion dependency*.

Definition 2 (Inter DOM Assertion Dependency) *An inter DOM assertion dependency is defined as*
$< assert, initPoint >$*, where assert is the intended DOM-based assertion, and initPoint is the initial line of code in the application that is responsible for mutating the property of a DOM element extracted from the intra DOM assertion dependency.*

In order to find the initial point of contact between the application's code and a mutated DOM property in the DOM-based test case, we track evolution of the accessed DOM elements (GETDOMMUTS in line 13 of the algorithm) as well as invoked event handlers as the test case runs. We consider additions and removals of child nodes, changes to attributes, and updates to child text nodes as DOM mutations. For instance, running the sample test case in Figure 5.1(b) results in mutating (1) the textual value of `div` element with class `shopContainer`, and (2) the `class` attribute of DOM element with ID `couponButt`.

In Section 5.4.2, we explain inferring the initial point of contact between the source code and a mutated DOM element in a DOM-based test suite in details.

Frequently Accessed DOM Properties. In addition to DOM-based assertions, we further consider DOM element properties, that are frequently accessed within the application as the test case runs (lines 1 to 7 of Algorithm 4). ACC in line 6 of the algorithm computes the access frequency of a DOM property, $freqAccdDOM$ in line 7 contains the inferred candidate DOM properties, and GETDOMMUTS in line 19 records DOM mutations occur on candidate DOM properties. The intuition is that frequent use of a given DOM property can point to the extent of application's behaviour dependency on the DOM property. Thus, if changes happen to that property through the JavaScript code, it is important to assert the correctness of such mutations. We define the access frequency of a DOM element

property as the number of times that the element's property has been read during the execution of a test case. DOM properties include attributes as well as textual value of the elements. In order to record DOM property accesses within the application, we rewrite native function calls used by programmers to access DOM element such as `getElementById`, `getElementsByClassName`, and `getElementsByTagName`. The returned object from these functions is later used to access attributes or textual values of the element. Thus, we apply a forward slice on the returned object to find instances of element's property access in the code. For example in function `addToCart` of Figure 5.1(a), DOM element with ID `couponButt` is assigned to `coupElem` variable. The assigned variable is later used to access the `class` attribute as well as the `value` of the DOM element in lines 23, 25, and 26.

Let $Acc(prop_{el})$ be the access frequency computed for property *prop* of DOM element *el*, then:

$Acc(prop_{el}) = \frac{Read(prop_{el})}{\sum_{e=1}^{n} Read(domElem_e)}$, where $Read(domElem_e)$ is the number of times that DOM element *domElem* is read, given that the total number of DOM elements during the execution of a test case is *n*. Note that reading a DOM element refers to accessing the element to read the corresponding property. In Figure 5.1(a), the `class` attribute of DOM element `couponButt` is read in lines 23 and 34, and thus the access frequency computed for the `class` attribute of the element is equal to $\frac{2}{3}$.

We choose element's property with access frequencies above a threshold α as potential candidates, which are later used for the purpose of unit-level assertion generation. We automatically compute this threshold for each test case as:

$\alpha = \frac{1}{ReadProperties(T)}$, where $ReadProperties(T)$ is the total number of properties which have been read during the execution of test suite T.

Going back to our running example and the sample DOM-based test case in Figure 5.1, `class` attribute of the `couponButt` is selected as a potential candidate since its access frequency ($\frac{2}{3}$) is greater than the computed threshold, which is equal to $\frac{1}{2}$ in this example.

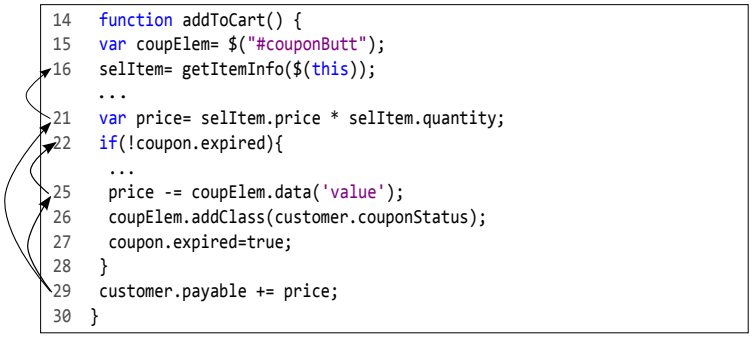


Figure 5.4: Intra code dependency through backward slicing.

5.4.2 Relating DOM Changes to the Application's Code

To determine the initial point of contact between DOM and the underlying application's code, we first cross reference the DOM element as well as the property we are interested in with a set of DOM mutations obtained from the execution trace. The desired DOM element and its property are inferred from either the intra DOM assertion dependency or the candidate DOM properties as described in Section 5.4.1. Recall that our execution trace contains information about triggered events, event handlers, and DOM mutations caused by the events. Therefore, we can identify relevant events and invoked functions corresponding to a given DOM mutation. For example, the collected execution trace in Figure 5.3 contains information about the mutations of a `div` element with class `shopContainer`, which pertains to the DOM-based assertion.

To figure out where the mutation originated in our execution trace, we keep record of DOM accesses within the invoked functions. For each DOM access, we track JavaScript lines of code that are responsible for updating the corresponding DOM element. Going back to our example in Figure 5.3, given that the textual property of the `div` element is extracted from the intra DOM assertion dependency, we identify line 37 in function `viewCart` as the initial point of contact responsible for changing the `text` of DOM element.

After inferring DOM mutant statements, we identify the *intra code dependency* within the application's code.

Definition 3 (Intra Code Dependency) *An intra code dependency is defined as*

$< criterion, codeSts >$, where criterion is a variable at the initial point of contact, and codeSts is the set of statements that are either affected by or influence the criterion.

To find the intra code dependency, we perform backward as well as forward slicing techniques by using *criterion* as the slicing criterion. GETBWSLICE in lines 15 and 21 of Algorithm 4 computes a backward slice with respect to assertion related DOM mutations, and candidate DOM property mutations respectively. We use dynamic slicing to capture run-time dependencies accurately. Note that instrumenting the entire application's code to perform dynamic slicing incurs high performance overheads. To avoid this, we first intercept the code sent from the server to the client, and then statically instrument only those statements that may affect a given DOM element. To extract the subset of the code statements, we first find the JavaScript closure scope which contains the definition of the variable in the initial slicing criteria. Then all references to the variable within the closure scope are found. Therefore, we can identify all locations in the code where the variable is updated, read, or a new alias is created. For each variable update/read related to the variable of the slicing criteria, we track the data dependencies for such an operation. The aforementioned steps are performed iteratively for each dependencies to collect the subset of code statements, which are instrumented for a given initial slicing criteria. The instrumented code keeps track of all updates and accesses to all relevant data and control dependencies. Once the test case runs, we collect traces from the instrumented code. This trace is used to dynamically extract backward slicing as well as forward slicing statements. Note that in addition to backwards slicing which is later used to generate explicit assertions, we also use forwards slicing to generate our implicit assertions (Section 5.4.3).

The backwards slicing technique starts by extracting instances of the initial slicing criteria from the trace. For each *read* operations, the trace is traversed backwards to find the nearest related *write* operation. Once found, the *write* operation is added to the slice under construction. This process is repeated for all the data dependencies related to that write operation. A similar approach is taken for including control dependencies in the slice. Our slicing technique supports inter procedural slicing. For example, if a variable is assigned by the return value of a

called function, the slicer recursively tracks the function and performs a backward slice on the statement returned by the called function.

To address aliasing when computing the slice of a variable that has been set by a non-primitive value, we need to consider possible aliases that may refer to the same object. Specifically in JavaScript *dot notation* and *bracket notation* are frequently used to modify objects at run time. Since static analysis techniques often ignore this issue [40], we use dynamic slicing. If a reference to an object of interest is saved to a second object's property, e.g. through the use of the *dot notation*, the object of interest may also be altered via aliases of the second object. For example, after executing statement `a.b.c = objOfInterest;`, updates to `objOfInterest` may be possible through `a`, `a.b`, or `a.b.c`. To deal with such scenarios, our slicing technique searches through the collected trace and adds the forward slice for each detected alias to the current slice for our variable of interest (e.g. `objOfInterest`).

Given `customer.payable` as the initial slicing criteria in our example, Figure 5.4 shows the relevant backward slice statements (lines 29, 25, 22, 21, and 16), where `customer.payable`, variable `price`, as well as properties of the object `selItem` are assigned, and the value of `coupon.expired` is checked in the conditional statement. By the end of backward slicing step, we collect all the relevant statements to a given DOM element. These are later used to derive test assertions.

5.4.3 Generating Unit-Level Assertions

Our approach targets postcondition assertions which are used to examine the expected behaviour of a given function after it is executed in a unit test case. By analyzing a given DOM-based test case, we generate unit-level assertions in the following three categories: (1) explicit assertions, (2) implicit assertions, and (3) candidate assertions.

Explicit Assertions

After collecting all the statements, that are relevant to a given DOM-based assertion, we extract accessible entities from these statements (ACCESSIBLES in line 23

of the algorithm). Types of accessible entities include (1) the function's returned value, (2) the used global variables in that function, (3) the object's property where the object is accessible in the outer scope of the function, and/or (4) the accessed DOM element in that function. Dynamic backward slice of a DOM-based assertion helps to (1) track all statements that contribute to the checked result and as such identify those entities that might have influenced the checked property value of the DOM element, and (2) eliminate unrelated entities that are not involved in the computation that leads to the update performed on the checked DOM element.

Since our dynamic slice is extracted from the program run, we can track all concrete values associated with accessible entities. During the run of a test case, there might be different instances where a given statement is executed. Different execution instances can lead to different behaviour. Since we are using dynamic slicing, an instance that leads to the required behaviour, which is checked through the DOM-based assertion, is on the backward slice. Given that the manually-written expected value, that is checked against the DOM's property is valid, the concrete values of related entities in the backward slice are potentially correct. Therefore, concrete value of an entity in the backward slice can be used as the expected value of the entity in unit-level assertions to test the current version of the application (discussed in Section 5.5.4). *explicitAsstn* in line 23 of Algorithm 4 contains the inferred explicit assertions.

In our running example (Figure 5.4), explicit assertions check the correctness of `customer.payable`, `coupon.expired`, as well as `price` and `quantity` properties which belong to `selItem` object. Assuming that the original price of the item is 100, the number of selected item is 1, and the calculated discount according to the `value` attribute of a DOM element with ID `couponButt` is 30, then the expected values included in the assertions for each of the entities are 70, boolean value `true`, 100, and 1, respectively.

Implicit Assertions

We gather all the statements that explicitly affect the computations relevant to a given DOM-based assertion. While assertions inferred from such statements are inherently important, we further need to consider entities that are implicitly in-

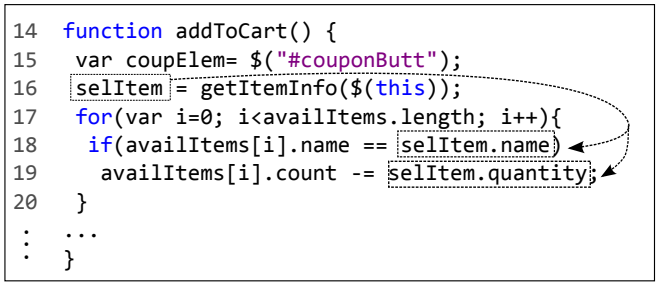


Figure 5.5: Intra code dependency through forward slicing.

fluenced by the checked DOM element in the manually-written test suite. For this purpose we apply a dynamic forward slice on the statements collected from a backward slice of a DOM-based assertion. A forward slice with respect to a statement *st*, indicates how subsequently an operand at *st* is being used. This can help the tester to ensure that *st* establishes the expected outcome of the computations assumed by later statements.

GETFWSLICE in line 17 of the algorithm computes forward slice on the variable operands of a statement in the backward slice. The process of forward slicing is similar to the backward slicing technique discussed earlier (Section 5.4.2). The slicing criterion of the forward slice module is either a variable, object's property, or an accessed DOM property extracted from the statements in a backward slice segments of the code. The accessible entities (ACCESSIBLES in line 24), which have been set within the collected forward slice statements establish the implicit assertions. *implicitAsstn* in line 24 of Algorithm 4 contains the inferred implicit assertions. Figure 5.5 shows the intra code dependency obtained by performing forward slicing on the running example. As shown in the figure, the properties of object `selItem` are set in line 16, that is recorded during the backward slice process. Given line 16 as the forward slice criteria, we mark `availItems.count` (line 19) as an implicit assertion.

Candidate Assertions

In addition to explicit and implicit assertions, we also verify the correctness of code-level entities pertaining to DOM updates, which are essentially important but not checked in the existing DOM-based test cases. We derive such unit-level as-

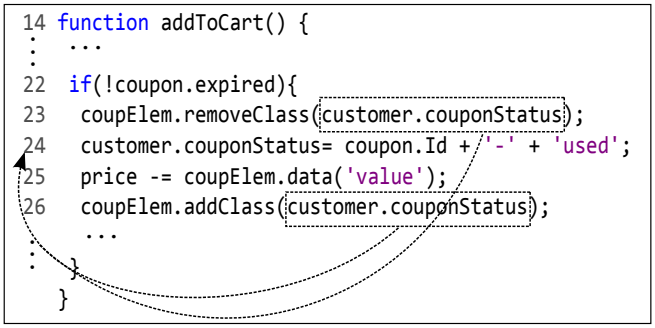


Figure 5.6: Relating candidate DOM element to JavaScript code.

sertions, namely candidate assertions, from the candidate DOM element properties previously obtained from the test case execution (box 3 in Figure 5.2). As the test case runs, we monitor the DOM's evolution and match the list of mutated DOM elements and their properties with property updates of the candidate DOM elements. Once a match is found, we infer backwards slice statements pertaining to the mutation of DOM element's property (GETBWSLICE in line 21 of the algorithm). Therefore, in this case the slicing criteria which is given as input to the backwards slicing module is an update to the property of the candidate DOM element. After gathering the related JavaScript statements within the application, we extract accessible entities of these statements (ACCESSIBLES in line 25) which form our candidate assertions. *candidateAsstn* in line 25 contains our candidate assertions.

Recall from the running example, one such potential DOM property which we record as part of Section 5.4.1, is `class` attribute associated with DOM element with ID `couponButt`. As shown in Figure 5.6 monitoring DOM changes reveal that line 26, where the `class` attribute of the element is set, is the initial point of contact between DOM mutation and the JavaScript code. Given line 26 as the slicing criteria, `customer.couponStatus` (line 24) is marked as the candidate assertion.

5.4.4 Tool Implementation: Atrina

We have implemented our JavaScript unit test assertion generation in an automated tool called Atrina. The tool is written in Java and is publicly available for download [1]. We use proxy server to intercept HTTP responses which contain JavaScript

code. JavaScript Mutation Summary library [7] is used to track DOM changes during the execution of the test suite. Trace information is collected by the proxy once received from the browser. To instrument Selenium test cases we convert them into an abstract syntax tree (AST) by employing Eclipse Java development tools (JDT). Once the transformation is done, we run the Java code of the changed AST on the application under test.

5.5 Empirical Evaluation

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

RQ1 How accurate is Atrina in mapping DOM-based assertions to the corresponding JavaScript code?

RQ2 How effective is Atrina in generating unit test assertions that detect faults?

RQ3 Is our approach more effective than DOM-based assertions written manually by the tester in terms of fault finding capability?

RQ4 How does our approach compare to existing mutation-based assertion generation techniques?

5.5.1 Objects

Our study includes four open source JavaScript web applications that have SELENIUM test cases. Table 5.1 presents the experimental objects and their properties. Phormer [8] is a photo gallery web application. EnterpriseStore [3] is an asset management web application. WolfCMS [11] is a content management system, and Claroline [2] is a collaborative online learning and course management system.

5.5.2 Setup

To address our research questions, we provide the URL as well as the available manually written DOM-based test suite of each experimental object to Atrina. Unit level test assertions are then automatically generated by the tool.

Table 5.1: Characteristics of the experimental objects.

ID	Name	LOC (JS)	# Test Cases	# Assertions
1	Phormer	1.5K	7	18
2	EnterpriseStore	57K	19	17
3	WolfCMS	1.3K	12	42
4	Claroline	36K	23	35

Accuracy (RQ1). To evaluate the accuracy of Atrina, we measure precision and recall. We manually compare the slices generated by Atrina with the JavaScript code that is relevant to each assertion. Precision and recall are defined as follows:

Precision is the fraction of lines in a slice produced by Atrina, that are actually related to the human-written DOM-based assertion: $\frac{TP}{TP+FP}$

Recall is the fraction of the correct set of related lines of code to each assertion, which is actually present in the slice produced by Atrina: $\frac{TP}{TP+FN}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of lines of code that are correctly reported, falsely reported, and missed to report as related to the DOM-based assertion.

Effectiveness (RQ2). To assess the effectiveness of Atrina, we measure the fault finding capability of the assertions generated by the tool. Moreover, to understand the effect of each type of assertion produced by Atrina in detecting faults, we compare the fault detection rate of using (1) exclusively explicit assertions, (2) explicit assertions and implicit assertions, and (3) explicit assertions and candidate assertions. Since explicit assertions compose the core body of our assertions, we consider implicit and candidate assertions in conjunction with explicit ones rather than separate them.

The experimental objects do not come with a rich version history to apply Atrina on real regression changes. Therefore we mimic regression faults by automatically injecting mutations to the application, and evaluate the tool's ability in detecting the seeded faults. Using our recently developed mutation testing tool, MUTANDIS [73], we automatically inject 50 random first-order mutations into the JavaScript code of the applications. The mutation operators are chosen from a list of common operators such as changing the value of a variable, modifying a conditional statement, altering unary operations, as well as common mistakes made

by developers when developing a given web application [76], e.g., changing the ID/tag name passed into DOM access functions such as `getElementById` or `getElementsByTagName`, and modifying the attribute name/value in `setAttribute`. The fault is considered detected if an assertion generated by Atrina fails when run on the mutated code, and our manual examination confirms that the failed assertion is detecting the seeded fault.

Comparison with human-written DOM-based Assertions (RQ3). To assess the usefulness of Atrina, we compare the human written DOM-based assertions with the unit-level test assertions generated by our approach in terms of fault finding capability. Similar to RQ2, we perform fault injection on both. The faults injected into our experimental objects in response to RQ3 are the same as the ones that we seed in applications to answer RQ2.

Comparison with Mutation-based Assertion Generation (RQ4). To assess how Atrina performs with respect to the current state-of-the-art oracle generation technique, we compare our tool's fault finding capability with the mutation-based assertion generation approach [43, 75]. To generate mutation-based assertions for the JavaScript code, we use human-written DOM-based test suite as a means to execute the application and infer the execution traces required for the purpose of mutation analysis. We perform the following steps to generate test assertions using mutation analysis.

1. Remove assertions from the human-written DOM-based test suite.
2. Execute the test suite on the original version of the application to obtain execution traces.
3. Inject mutations for the purpose of oracle generation.
4. Execute the human-written test suite on the generated mutants, and produce test oracles by comparing execution traces obtained from the mutants and the original version of the application.

The number of mutants, that are generated to produce test assertions is 50 for each application. Note that the implementation and evaluation of the mutation analysis technique both use mutation operators from our prior work [76]. Therefore, our evaluation is biased in favour of mutation-based assertion generation approach over our technique.

Table 5.2: Accuracy achieved by Atrina.

ID	# TP	# FP	# FN	Precision (%)	Recall (%)
1	174	0	0	100	100
2	861	18	162	98	84
3	193	0	0	100	100
4	1446	29	385	98	79
AVG	-	-	-	99	90

5.5.3 Results

Accuracy (RQ1). Table 5.2 shows the number of correctly reported (true positive), the number of incorrect reported (false positive), and the number of missed (false negative) JavaScript lines of code, which are related to human-written DOM-based assertions. The table also shows the percentage of precision and recall achieved by Atrina. The Recall calculated for Phormer (ID 1) and WolfCMS (ID 3) is 100%. For EnterpriseStore (ID 2) and Claroline (ID 4), the recall rate is 84% and 79% respectively. The precision computed for the Phormer and WolfCMS is 100%. For EnterpriseStore application as well as Claroline, the precision rate is 98%.

We noticed that the lower recall rate obtained by Atrina is mainly due to the use of third party libraries. Currently, we focus only on the application source code and do not consider libraries in our slicing technique. The underlying assumption is that faults mainly originate from the application's code. The small drop observed in precision is due to the functions, that are called but are not instrumented due to limitations in our current implementation. If the definition of a called function is not instrumented, we assume that the function call is related to our slice, while it may not be. We also observed that in rare cases a variable is seemingly assigned by a return value of a function, though the `return` statement is not found in the actual code of the called function. Our current implementation includes such variable assignments in the pertaining slices. Note that both recall and precision can be improved to 100% with a more robust implementation of our technique.

Effectiveness (RQ2). Figure 5.7 depicts the fault detection rate (percentages) achieved by (1) Atrina, (2) explicit assertions when included individually, and (3) explicit assertions in conjunction with either implicit assertions or (4) candidate

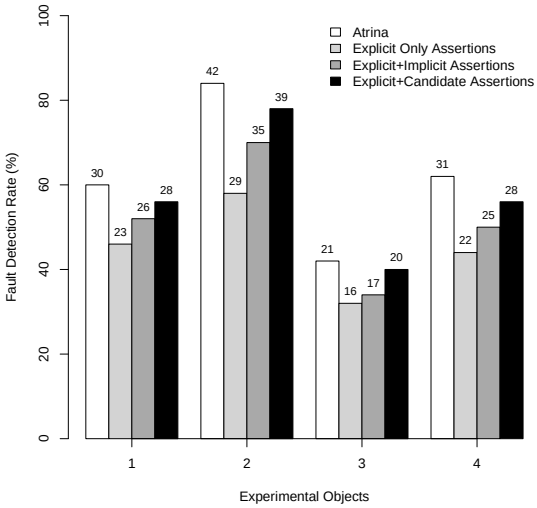


Figure 5.7: Fault detection rate using different types of generated assertions.

assertions. The number on each bar represent the number of faults detected by the corresponding assertion types. As shown in Figure 5.7, Atrina detects on average 62% of the total faults (ranges from 42-84%). The percentage of faults detected by including only explicit assertions is less than that detected through the combination of explicit with either implicit assertions or candidate assertions. This indicates that implicit as well as candidate assertions are essential entities in improving the fault finding capability of Atrina. By eliminating implicit and candidate assertions, fault detection rate drops 27% on average and up to 31% for the EnterpriseStore application (ID 2).

Figure 5.7 shows that the improvement contributed by implicit assertions is 8% on average, while the improvement due to candidate assertions is 21% on average. This indicates that candidate assertions play a more prominent role in increasing the number of faults detected by Atrina in comparison with implicit assertions. Not surprisingly, explicit assertions contribute the most among the other assertion types generated by Atrina. Explicit assertions detect 73% of the total faults on average (ranges from 69-77%). These assertions are derived directly from the DOM-based oracles written by the developer of the application who has a deep knowledge of the application's behaviour. Therefore, it is expected that explicit assertions derived

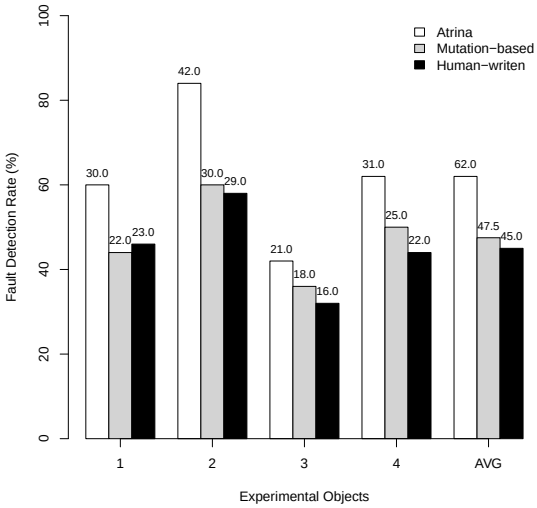


Figure 5.8: Fault finding capability.

directly from such oracles have the highest impact on fault finding capability of our tool.

Comparison with human-written DOM-based Assertions (RQ3). Figure 5.8 compares the fault detection rate achieved by the code-level assertions generated by Atrina with the human-written DOM-based assertions. The numbers shown on each bar represent the actual number of faults detected by the corresponding assertion generation technique. As shown in the figure, the percentage of faults found by Atrina is higher than manually written DOM-based assertions for all applications. Overall, our approach outperforms manual assertions in terms of fault finding capability by 37% on average (ranges from 30-45%). We observed that on average, 52% of the candidate DOM properties that we select to construct our candidate assertions were ignored in human-written DOM assertions, although their values are updated through the JavaScript code. We further noticed that for each failed manual DOM assertion as a result of an injected fault, at least one explicit assertion fails in Atrina (three failed explicit assertion on average). We observed that most often DOM assertions written by the tester are too generic in nature. Therefore even when a DOM assertion detects a JavaScript fault, pinpointing the root cause of the error can be quite challenging. However, code-level assertions make it easier

for the tester to localize the fault, as they directly correlate with the code.

We observed in several cases that the value of a DOM element property that is checked in the human-written test suite is later used in a JavaScript code that involves internal computations only. If the seeded fault falls in the corresponding computational statements, the resulting error is not captured through the manually written DOM assertions. In such cases, implicit assertions are capable of detecting the error, which points to the importance of incorporating these types of assertions in our approach. We also noticed that around 33% of the faults found by implicit assertions are undetected by the human-written ones. This is because they require executing a more complex sequence of events to propagate to the observable DOM (e.g., when an object's property is assigned in a function to be later used in updating a value of a DOM element after a specific event is triggered).

Comparison with Mutation-based Assertion Generation (RQ4). Figure 5.8 presents the results of comparing fault finding capability of Atrina with mutation-based assertion generation technique. As shown in the figure our approach produces unit assertions that are more effective than those produced by mutation-based technique. Atrina surpasses mutation-based approach by 29% on average (ranges from 17-40%), although both implementation and evaluation of the mutation-based technique use common mutation operators, which favours mutation-based assertion generation as we discussed in Section 5.5.2. This points to the importance of incorporating the information that exists in human-written DOM-based test cases.

5.5.4 Discussion

Time Efficiency. While the results demonstrate that Atrina is more effective than mutation-based approach in terms of fault detection, we further investigate efficiency of our approach in terms of time overhead. We compute overhead of Atrina as the summation of time required for (1) instrumenting the application, and (2) analyzing the collected trace to compute JavaScript slices. To calculate time overhead of the mutation-based approach, we consider the total time required for running the test suite multiple times (once per mutation), generating mutants, as well as the time needed to compare the original and the mutated version of the application to generate assertions. Figure 5.9 shows the results of time overhead

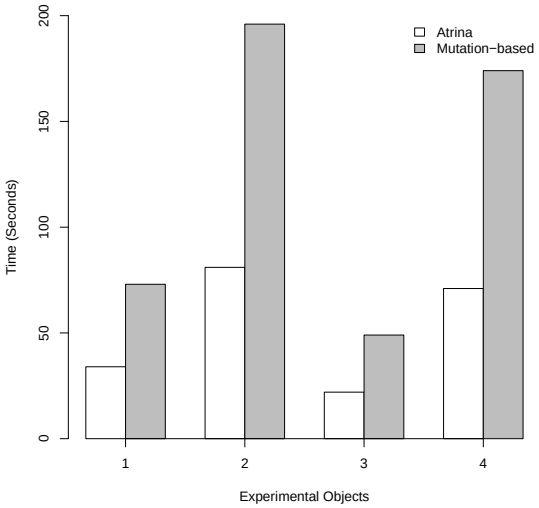


Figure 5.9: Time overhead for each approach.

computed for each approach. Our results show that the time overhead for Atrina is 52 seconds on average, while the overhead computed for mutation-based technique is 123 seconds on average. As shown in the figure, for the EnterpriseStore application (ID 2), which is the largest application we considered (57K LOC), time efficiency is increased by 58% using Atrina. This indicates that our approach significantly outperforms mutation-based assertion generation as far as time efficiency is concerned.

Fault Masking. As we mentioned in Section 5.4.3, the concrete value of an entity in the computed backward slice can potentially be used as the expected value of the entity in explicit assertions to test the current version of the application. The actual values of the related entities in the backward slice are correct unless there exists a masked fault which is concealed in the chain of computations and thus does not propagate to the checked state of the DOM element. However, we conjecture that fault masking rarely happens in JavaScript web applications as it is more prevalent in programs with many small expressions whose results are stored in several intermediate values. We also observed no fault masking occurrence during the evaluation of Atrina on four JavaScript applications used in this study.

Limitations. The effectiveness of the generated assertions by Atrina in terms of

fault finding capability depends on the quality of human-written DOM-based test cases. If the DOM assertions contained in the DOM-based test suite check useless information, the explicit assertions obtained by our tool point to entities that may not be important from the tester's point of view. This can also negatively affect the fault finding capability of implicit assertions as they are indirectly inferred from the DOM-based assertions. Moreover, if the human-written test suite does not execute application's state with effective DOM elements, our tool is not able to infer effective candidate assertions.

5.5.5 Threats to Validity

An external threat to the validity of our evaluation is the limited number of JavaScript applications used to measure the effectiveness of our approach. We mitigated this threat by using web applications from various domains, code size, and functionality. Another threat concerns validating failed assertions through manual inspection that can be error-prone. To mitigate this threat, we carefully examine the code in which the assertion failed to make sure that the injected fault was indeed responsible for the assertion failure. Moreover, manual computation of the JavaScript slices to measure precision and recall is a time intensive task, and thus could be error-prone. However, we made every effort to mitigate this threat by precisely examining the application's code.

The regression faults we inject to evaluate the effectiveness of Atrina may not be realistic. We mitigate this threat by injecting mutations that represent common JavaScript applications faults, as well as using real-world web applications, and SELENIUM test cases written by developers.

5.6 Related Work

While automated test generation has significantly addressed in the literature, there has been limited work on supporting the construction of test oracles. Recently, Harman et al. [53] have conducted a comprehensive survey of current techniques used to address the oracle problem. Mesbah et al. [70] automatically produce generic invariants in a form of soft oracles to test AJAX applications. JSART [72] automatically infers JavaScript invariants from the execution traces for the purpose

of regression testing. Jalangi [99] is a framework to support writing of heavy-weight dynamic analyses. The framework detects generic JavaScript faults such as null, undefined values, and type inconsistencies. Jensen et al. [58] incorporate server interface descriptions to test the correctness of communication patterns between client and server through learning the communication patterns from sample data in AJAX applications. Xie et al. explore test oracle generation for GUI systems [107]. Eclat [85], and DiffGen [102] are used for automatically generating invariant-based oracles. Our work is different from these approaches in that we use the available DOM-related information in a human written test suite to infer unit-level assertions at the JavaScript code-level. Moreover, we generate assertions that capture application's behaviour, rather than generic and soft oracles.

Fraser et al. [43] propose a mutation-based oracle generation system called μTEST. μTEST automatically generates unit tests for Java object-oriented classes by employing a genetic algorithm which target mutations with high impact on the application's behaviour. They further enhance the system [42] to improve human comprehension through identifying relevant pre-conditions on the test inputs and post-conditions on the outputs. The authors assume that the tester will manually correct the generated oracles. However, the results on the effectiveness of such approaches which rely on the "generate-and-fix" assumption to construct test oracles are not conclusive [44]

Staats et al. [101] propose an oracle data selection technique, which is based on mutation testing to produce oracles and rank the inferred oracles in terms of their fault finding capability. This work suffers from the scalability issues of mutant-generation based techniques as well as the problem of estimating the proper number of mutants required for generating effective oracle data set. Similar to mutation-based techniques, differential test case generation approaches [37, 102] also target generating test cases that show the difference between two versions of a program. Pastore et al. [86] exploit crowd sourcing approach to check assertions. In this approach the developer produces tests and provides sufficient API documentation for the crowd such that crowd workers can determine the correctness of assertions. However, recruiting qualified crowd to generate test oracles can be quite challenging.

In the context of leveraging the existing test cases to generate more complex

tests, Pezzè et al. [88] propose a technique to construct integration tests which focus on class interactions by utilizing the unit test cases. The integration tests are formed by combining initialization and execution sequences of simple unit tests to form new ones. However, the proposed technique does not deal with assertions. eToc [104] and EvoSuite [41] use search based techniques to evolve the initial population of test cases. Their main goal is to increase the code coverage achieved by the test suite. However, in this work our aim is to increase the fault finding capability by focusing on test assertions rather than increasing the code coverage. Milani Fard et al. [39] propose Testilizer which utilizes DOM-based test suite of the web application to explore alternative paths and consequently regenerate assertions for new detected states. Our work is different from this approach in that we exploit DOM-related information in a human written test suite to capture the behaviour of the application at the unit-level JavaScript. Furthermore, they do not generate code-based assertions which we do.

Chapter 6

Conclusions

JavaScript is increasingly being used to create modern interactive web applications that offload a considerable amount of their execution to the client-side. JavaScript is a notoriously challenging language for web developers to use, maintain, analyze and test. The work presented in this dissertation aims at improving the state-of-the-art in testing JavaScript web applications by proposing a new set of techniques and tools.

6.1 Contributions

The main contributions of the thesis are as follows:

- A new automated technique for JavaScript regression testing, which is based on dynamic analysis to infer invariant assertions; The obtained assertions are injected back into the JavaScript code to uncover regression faults in subsequent revisions of the web application under test.
- The first JavaScript mutation testing tool, which is capable of guiding the mutation generation towards behaviour-affecting mutants in error-prone portions of the code; The mutation testing method combines dynamic and static analysis to mutate branches that are within highly ranked functions and exhibit high structural complexity.
- An automatic technique to generate test cases for JavaScript functions and events; We use a mutation-based algorithm to effectively generate test oracles, capable of detecting regression JavaScript and DOM-level faults. The

technique uses a combination of function converge maximization and function state abstraction algorithms to efficiently generate unit test cases.

- Exploiting an existing GUI-based (i.e., DOM) test suite to generate unit-level assertions for applications that highly interact with the GUI through the underlying JavaScript code; We utilize existing assertions as well as useful execution information inferred from a GUI test suite to automatically generate assertions used for testing individual JavaScript functions.

6.2 Revisiting Research Questions

In the beginning of this thesis, we designed two research questions. We believe that the contributions show that we have addressed the research questions.

Research Question 1.

How can we generate effective test cases for JavaScript web applications?

In order to answer the first research question, Chapter 2 targets web application testing from the invariant assertions points of view. These invariants formulate the main characteristics of the application under test that will remain unchanged as the application evolves. Therefore, these type of assertions can be used towards regression testing. The empirical study on nine open source JavaScript applications show that the proposed approach is able to effectively infer stable assertions and detect regression faults with minimal performance overhead.

Our invariant generation technique is based on the assumption that the program specifications are not changed frequently in subsequent revisions. However, if major changes affect the core properties of the application, the inferred invariants from the original version may not be valid any longer. Moreover, unlike post-condition assertions, invariant-based ones are inline assertions, which are checked at different points of the program's code. Therefore, it can become difficult for the tester to comprehend these type of assertions.

In order to generate oracles that can be used during the common testing cycles of a large scale system, including unit and GUI testing, we proposed JSEFT in Chapter 4. JSEFT generates test cases combined with post-condition assertions at the two complementary levels of unit and event-based tests. We use mutation

testing to produce our assertions. To evaluate the effectiveness of JSEFT we consider a state-of-the-art JavaScript test generation framework as a basis to compare our technique. The results of the empirical evaluation indicate that the approach generates test cases with high fault finding capability.

Further analysis of the results revealed that (1) although, the generated assertions by JSEFT are effective in detecting the injected faults, the use of mutation testing for the purpose of assertion generation can negatively impact the performance, and (2) event-based tests can potentially miss the code-related errors (32% on average) if the fault does not propagate to the observable GUI state. We observed that the rate of missed faults is higher for the applications that have tight interaction with the GUI through the underlying executable code. These two observations form the basis of Chapter 5, where we make use of the GUI-dependent assertions as a guide to generate code related unit-level assertions.

In Chapter 5, we proposed Atrina which utilizes the existing GUI-based (i.e., DOM) assertions as well as useful execution information inferred from a GUI test suite to automatically generate assertions used for testing individual functions. This work is currently under preparation, however, the initial results confirm that the generated unit-level assertions surpass the fault finding capability of DOM-based assertions with 37% on average. We also found out that Atrina outperforms mutation-based assertion generation technique in terms of the time efficiency.

During the evaluation of different test generation techniques proposed in this thesis, we realized that using hidden scopes (i.e., function closures) is quite popular in writing JavaScript applications. Hidden scopes in JavaScript language provide a way to make variables and functions private, thus keeping them out of the global scope. While function closures can be called during the testing process at the highest program scope they belong to, it is not possible to call them directly in test cases, which makes it challenging to assess their outcomes. One possible future direction is to measure the extent of such hard-to-test code written by developers by conducting a thorough empirical study. The results of the study can be used towards generating effective test cases by identifying hard-to-test scopes, and if possible expose them to the testing unit through automated code refactoring. JavaScript developers can also make use of the results of empirical study as a coding recommendation to make their future applications more testable and consequently more

maintainable.

Research Question 2.

How can we effectively assess the quality of the existing JavaScript test cases?

To address the second research question, in Chapter 3 we proposed MUTANDIS, a generic mutation testing approach, that guides the mutation generation towards error-prone sections of the program. The empirical evaluation indicates that MUTANDIS can (1) significantly reduce the number of equivalent mutants, and (2) guide testers towards designing test cases for important portions of the code from the application's behaviour point of view.

Reducing the number of equivalent mutants can potentially eliminate stubborn (hard-to-kill) mutants, which are particularly important for assessing the fault finding capability of test cases. The current evaluation results show that MUTANDIS does not negatively influence the stubbornness of the mutants. However, our approach is not particularly designed to generate such mutations. We found out that the stubbornness of the mutants generated by MUTANDIS stems from the inherent characteristics of the JavaScript functions, such as function variadicity. Therefore, one interesting future work direction is to enhance the mutation generation technique by taking into account such particular function characteristics. This way we can reduce the number of equivalent mutants while increasing the number of stubborn mutants.

Another future work is to consider GUI-level mutations, which are particularly designed to assess the quality of GUI-level tests. However, GUI mutation is involved with a number of challenges. Since in GUI mutation, the output is the resulting state from an executed event, the scope of the mutation operators differs from the traditional code-based mutant generators. Therefore, we need to define (1) a new set of GUI-based mutation operators, and (2) a new equivalent mutant detection technique.

6.3 Concluding Remarks

The work presented in this thesis has focused on providing JavaScript applications with a rich set of new test automation techniques. Given the growing popularity of JavaScript and the challenges of testing this dynamic language, we see many

opportunities for using the proposed techniques in practice. Further, developers can use our approaches not only to test the applications, but also to assess and compare the adequacy of their web application testing techniques.

Bibliography

[1] Atrina. http://salt.ece.ubc.ca/software/Atrina/. → pages 108

[2] Claroline. http://www.claroline.net/. → pages 109

[3] WSO2 EnterpriseStore. https://github.com/wso2/enterprise-store. → pages 109

[4] jQuery API. http://api.jquery.com. → pages 68

[5] JSCover. http://tntim96.github.io/JSCover/. → pages 85

[6] JSeft. http://salt.ece.ubc.ca/software/jseft/. → pages 66, 83, 84

[7] Google. Mutation Summary Library. http://code.google.com/p/mutation-summary/. → pages 109

[8] Phormer Photogallery. http://sourceforge.net/projects/rephormer/. → pages 109

[9] Qunit. http://qunitjs.com. → pages 2, 74, 83

[10] Selenium. http://docs.seleniumhq.org/. → pages 2

[11] WolfCMS. https://github.com/wolfcms/wolfcms. → pages 109

[12] String replace JavaScript bad design. http://www.thespanner.co.uk/2010/09/27/string-replace-javascript-bad-design/, 2010. → pages 44

[13] 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 Evolutionary Computation Conference (GECCO)*, pages 1338–1349. ACM, 2004. → pages 28, 62

[14] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the Int. Conf. on Software Testing, Verification, and Validation (ICST'08)*, pages 298–307. IEEE Computer Society, 2008. ISBN 978-0-7695-3127-4. doi:http://doi.ieeecomputersociety.org/10.1109/ICST.2008.56. → pages 25

[15] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013. → pages 1

[16] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 402–411. ACM, 2005. → pages 5, 28

[17] 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)*, pages 571–580, 2011. → pages 4, 25, 29, 66, 67, 86, 91, 94

[18] K. Ayari, S. Bouktif, and G. Antoniol. Automatic mutation test input data generation via ant colony. In *Proc. Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1074–1081. ACM, 2007. → pages 63

[19] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability (STVR)*, 11(2):113–136, 2001. → pages 5, 61

[20] V. Basili, L. Briand, and W. Melo. A validation of object orient design metrics as quality indicators. *IEEE Transaction on Software Engineering (TSE)*, 22(10):751–761, 1996. → pages 32

[21] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proc. Euro. Softw. Eng. Conf. and Symp. on the Foundations of Softw. Eng. (ESEC-FSE)*, pages 81–91. ACM, 2009. → pages 11, 16

[22] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proc. International Conference on Software Engineering (ICSE)*, pages 419–429. ACM, 2012. → pages 64

[23] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000. → pages 24

[24] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 169–180. ACM, 2006. doi:http://doi.acm.org/10.1145/1146238.1146258. → pages 13, 25

[25] L. Bottaci. Type sensitive application of mutation operators for dynamically typed programs. In *Proc. 5th International Workshop on Mutation Analysis*, pages 126–131, 2010. → pages 62

[26] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998. → pages 33, 35

[27] T. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982. → pages 5, 28

[28] A. Burgess. The 11 JavaScript mistakes you are making. http://net.tutsplus.com/tutorials/javascript-ajax/the-10-javascript-mistakes-youre-making/, 2011. → pages 44

[29] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006. → pages 25

[30] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. → pages 77

[31] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. → pages 29, 44, 45

[32] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 281–290. ACM, 2008. → pages 13, 25

[33] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. → pages 77

[34] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *Proc. International Conference on Software Testing Verification and Validation (ICST)*, pages 84–93. IEEE Computer Society, 2013. → pages 63

[35] J. Domínguez-Jiménez, A. Estero-Botaro, A. Garca-Domnguez, and I. Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123, 2011. → pages 62

[36] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31: 187–202, 2005. ISSN 0098-5589. → pages 25

[37] 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)*, 35(1):29–45, 2009. → pages 91, 118

[38] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. → pages 13, 16, 25

[39] A. M. Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proc. IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 67–78, 2014. → pages 94, 119

[40] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proc. International Conference on Software Engineering (ICSE)*, pages 752–761. ACM, 2013. → pages 105

[41] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proc. International Conference on Software Testing Verification and Validation (ICST'12)*, pages 121–130. IEEE Computer Society, 2012. → pages 119

[42] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 364–374, 2011. → pages 4, 91, 118

[43] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2): 278–292, 2012. → pages 3, 63, 78, 91, 94, 111, 118

[44] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proc. International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 188–198. ACM, 2013. doi:http://doi.acm.org/10.1145/1146238.1146258. → pages 118

[45] V. Garousi, A. Mesbah, A. B. Can, and S. Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374–1396, 2013. → pages 7

[46] M. Gligoric, L. Z. C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. Intl. Symp. Softw. Testing and Analysis (ISSTA)*, pages 224–234. ACM, 2013. → pages 6, 62

[47] F. Groeneveld, A. Mesbah, and A. van Deursen. Automatic invariant detection in dynamic web applications. Technical Report TUD-SERG-2010-037, TUDelft, 2010. → pages 16

[48] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Conference on USENIX security symposium*, SSYM'09, pages 151–168, 2009. → pages 25

[49] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Intl. conference on World Wide Web (WWW)*, pages 561–570, 2009. ISBN 978-1-60558-487-4. → pages 25

[50] P. Gurbani and S. Cinos. Top 13 JavaScript mistakes. http://blog.tuenti.com/dev/top-13-javascript-mistakes/, 2010. → pages 44

[51] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 291–301. ACM Press, 2002. → pages 13, 25

[52] M. Harman, Y. Jia, and W. Langdon. Strong higher order mutation-based test data generation. In *Proc. ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 212–222. ACM, 2011. → pages 63

[53] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, Department of Computer Science, University of Sheffield, 2013. → pages 117

[54] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability*, 9(4):233–262, 1999. → pages 62

[55] T. Ho. Premature invocation. http://tobyho.com/2011/10/26/js-premature-invocation/, 2011. → pages 44

[56] J. Hsu. JavaScript anti-patterns. http://jaysoo.ca/2010/05/06/javascript-anti-patterns/, 2010. → pages 44

[57] K. Jalbert and J. Bradbury. Predicting mutation score using source code and test suite metrics. In *Proc. International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 42–46, 2012. → pages 49

[58] 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. → pages 3, 91, 118

[59] C. Ji, Z. Chen, B. Xu, and Z. Zhao. A novel method of mutation clustering based on domain analysis. In *Proc. 21st International conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 422–425, 2009. → pages 5, 61

[60] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258. ACM, 2008. → pages 6, 61

[61] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37 (5):649–678, 2010. → pages 5, 6, 28, 61

[62] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In *Proc. International Conference on Software Testing Verification and Validation (ICST)*, pages 701–710. IEEE Computer Society, 2012. → pages 63

[63] W. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, 2010. → pages 62

[64] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering (TSE)*, 40(1):23–42, 2013. → pages 5, 6, 28, 57, 61

[65] A. Marchetto and P. Tonella. Using search-based algorithms for Ajax event sequence generation during testing. *Empirical Software Engineering*, 16 (1):103–140, 2011. → pages 4, 66, 90

[66] 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)*, pages 121–130. IEEE Computer Society, 2008. → pages 4, 25, 66

[67] T. McCabe. A complexity measure. *IEEE Transaction on Software Engineering (TSE)*, SE-2(4):308–320, 1976. → pages 39

[68] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proc. International Conference on Software Engineering (ICSE'12)*, pages 408–418, 2012. → pages 7

[69] 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)*, 6(1):3:1–3:30, 2012. → pages 12, 16, 46, 83, 90

[70] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012. → pages 4, 25, 29, 66, 70, 90, 117

[71] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/2.161279. → pages 10

[72] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. International Conference on Web Engineering (ICWE'12)*, pages 238–252. Springer, 2012. → pages 7, 8, 40, 46, 90, 117

[73] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. 6th International Conference on Software Testing Verification and Validation (ICST'13)*, pages 74–83. IEEE Computer Society, 2013. → pages 7, 27, 81, 83, 110

[74] 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*, pages 610–615. IEEE Computer Society, 2013. → pages 7, 65

[75] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: Automated javascript unit test generation. In *to appear in Proc. 8th International Conference on Software Testing Verification and Validation (ICST'15)*. IEEE Computer Society, 2015. → pages 7, 65, 94, 111

[76] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for javascript web applications. *(in press) IEEE Transactions on Software Engineering (TSE)*, 2015. → pages 7, 27, 111

[77] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. International Conference on Software Engineering (ICSE)*, pages 452–461. IEEE Computer Society, 2006. → pages 32

[78] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. International Conference on Software Engineering (ICSE)*, pages 351–360. ACM, 2008. → pages 5, 61

[79] F. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: An empirical study. In *Proc. of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109. IEEE Computer Society, 2011. → pages 29, 51

[80] 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)*, pages 55–64. IEEE Computer Society, 2013. → pages 2, 92

[81] F. J. Ocariza, K. Pattabiraman, and A. Mesbah. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST'12)*, pages 31–40. IEEE Computer Society, 2012. → pages 12

[82] A. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *International Conference on Computer Assurance (COMPASS)*, pages 224–236, 1996. → pages 28, 61

[83] A. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3): 165–192, 1997. → pages 6, 28, 61

[84] A. Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, 2012. → pages 44

[85] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. European Conference on Object-Oriented Programming (ECOOP'05)*, pages 50–527, 2005. → pages 118

[86] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *Proc. International Conference on Software Testing Verification and Validation (ICST'13)*, pages 342–351. IEEE Computer Society, 2013. → pages 118

[87] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for Web 2.0 application robustness testing. In *Proc. Int. Conf. Sw. Reliability Engineering (ISSRE'10)*, pages 191–200. IEEE Computer Society, 2010. → pages 25

[88] M. Pezzè, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proc. International Conference on Software Testing Verification and Validation (ICST'13)*, pages 11–20. IEEE Computer Society, 2013. → pages 119

[89] C. Porteneuve. Getting out of binding situations in JavaScript. http://www.alistapart.com/articles/getoutbindingsituations/, 2008. → pages 45

[90] S. Ratcliff, D. White, and J. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 2011. → pages 25

[91] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proc. of the conf. on Programming language design and implementation (PLDI'10)*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0019-3. doi:http://doi.acm.org/10.1145/1806596.1806598. → pages 60

[92] E. Rodríguez-Carbonell and D. Kapur. Program verification using automatic generation of invariants. In *Proc. 1st International Colloquium*

on Theoretical Aspects of Computing (ICTAC'04), volume 3407 of *Lecture Notes in Computer Science*, pages 325–340. Springer-Verlag, 2005. → pages 25

[93] D. Roest, A. Mesbah, and A. van Deursen. Regression testing Ajax applications: Coping with dynamism. In *Proc. 3rd Int. Conf. on Sw. Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010. → pages 8, 25

[94] 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. → pages 44, 45

[95] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. → pages 17

[96] 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)*, pages 513–528. IEEE Computer Society, 2010. ISBN 978-0-7695-4035-1. doi:http://dx.doi.org/10.1109/SP.2010.38. → pages 4, 66, 91

[97] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification, and Reliability*, 23(5):353–374, 2012. → pages 5, 6, 28, 55, 63

[98] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. Intl. Symp. Softw. Testing and Analysis (ISSTA)*, pages 69–79. ACM, 2009. → pages 40, 63

[99] 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. → pages 3, 91, 118

[100] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM, 2005. ISBN 1-59593-993-4. doi:http://doi.acm.org/10.1145/1101908.1101947. → pages 25

[101] 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)*, pages 870–880, 2011. → pages 92, 118

[102] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. International Conference on Automated Software Engineering (ASE'08)*, pages 407–410. IEEE Computer Society, 2008. → pages 91, 118

[103] A. Tarhini, Z. Ismail, and N. Mansour. Regression testing web applications. In *Int. Conf. on Advanced Comp. Theory and Eng.*, pages 902–906. IEEE Computer Society, 2008. ISBN 978-0-7695-3489-3. doi:http://doi.ieeecomputersociety.org/10.1109/ICACTE.2008.82. → pages 8, 25

[104] P. Tonella. Evolutionary testing of classes. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, pages 119–128, 2004. → pages 119

[105] E. Weyl. 16 common JavaScript gotchas. http://www.standardista.com/javascript/15-common-javascript-gotchas/, 2010. → pages 44, 45

[106] M. Woodward. Mutation testing - its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993. → pages 56

[107] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1), 2007. → pages 118

[108] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *Proc. of Int. Conf. on Computer Software and Applications (COMPSAC)*, pages 652–656. IEEE Computer Society, 2003. ISBN 0-7695-2020-0. → pages 25

[109] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proc. International Conference on Software Engineering (ICSE)*, pages 919–930. ACM, 2014. → pages 50, 58

[110] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. International Conference on Software Engineering (ICSE)*, pages 435–444. ACM, 2010. → pages 5, 61

[111] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. Intl. Symp. Softw. Testing and Analysis (ISSTA)*, pages 235–245. ACM, 2013. → pages 63

[112] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the Intl. Conference on the World-Wide Web (WWW)*, pages 805–814. ACM, 2011. → pages 25