Mutation Analysis for Assessing End-to-End Web Tests

Rahulkrishna Yandrapally
University of British Columbia
Vancouver, BC, Canada
rahulyk@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—End-to-end UI testing plays a significant role in the regression testing of web apps, in order to validate end user functionality. Because of their importance, UI test suites are often created and maintained manually by employing browser automation tools such as selenium. However, currently, there exists no reliable method to ascertain the fault-finding capabilities for UI test suite of any given web app. Mutation testing, a well known fault-based testing technique for assessment of test suite adequacy, relies on generating mutants by making small changes to source code imitating programmer errors. However, mutation testing is difficult to employ for any given web app because of the heterogeneous nature of the multiple server-side and client-side components they can contain. In this work, we present MAEWU, a mutation analysis framework for assessing web UI test suites, which is applicable to any web app as it mutates the dynamic DOM in the browser instead of the source code. We propose 16 mutation operators that mutate the behaviour and appearance of web elements to mimic the nine categories of web app faults found through an analysis of 250 bug reports. We evaluate our dynamic mutation analysis framework on six open source web apps. The results from our empirical evaluation demonstrate that MAEWU is effective in assessing Web UI test suites in terms of adequacy and facilitates test suite quality improvements.**

I. INTRODUCTION

Modern web apps are highly dynamic in nature and contain a heterogeneous collection of server-side and client-side components that interact in real-time to update the web page in response to user requests. Consequently, testing web apps programmatically is challenging and is often performed in an end-to-end (E2E) fashion by exercising the GUI functionality of web apps. Given the short release cycles of web apps, automated regression testing using UI tests plays a significant role in the validation of web app changes.

Because of their importance, companies currently invest manpower in creating and maintaining such UI tests suites. However, despite this reliance on UI test suites to validate web app functionality, currently, there is no universal tool to determine their fault-finding capabilities. Therefore, in practice, UI test suite adequacy is determined by coverage of common use case scenarios, and certain server-side and client-side code. However, such coverage metrics are generally considered to be limited for assessing test effectiveness [1], [2].

Instead, mutation analysis, which mimics programmer errors by making small changes to the application has become an accepted norm in establishing the fault revealing capabilities of test suites. Existing mutation analysis tools for web apps are not universally applicable as they are designed for specific programming languages or web development frameworks. Therefore, currently, there exists no mutation analysis framework to assess the actual effectiveness of web-based UI test suites.

In this paper, we propose MAEWU, a **M**utation **a**nalysis framework for **E**nd-to-end **w**eb **U**I test suites. MAEWU mutates the dynamic DOM of the web app in the browser in order to bypass the limitations of a source code-based mutation analysis employed by existing techniques. Consequently, MAEWU only requires the URL of the web app and its test suite to perform mutation analysis; it neither requires access to the source code of the web app nor employs any proxy to instrument the client-side code.

Using the dynamic DOM state as an artifact for mutation is conceptually novel because we mutate the output instead of the actual source code written by programmers. While DOM mutation allows for universal applicability in all web apps, it also poses a unique challenge vis-a-vis its availability during test execution. First, in traditional mutation analysis, a mutation applied to a source code artifact is preserved over multiple invocations during test execution. The same does not apply for a mutated DOM in a browser which can disappear upon navigation or a page reload. In addition, as web pages are dynamically generated, applying a mutation consistently to each appearance of the browser state is challenging as a state equivalence between concrete instances needs to be established. Second, a typical modern web page is essentially a set of individual UI components, where each component can appear in multiple different pages. As a result, mutating a web element such as a navigation link necessitates identifying all its instances across web pages. We make use of an automatic page fragmentation technique and a tree comparison technique in order to accomplish this task. A similar challenge exists for other kinds of GUI testing such as desktop applications [3] or Mobile apps [4]. However, researchers in those areas are able to rely on mutating the source artifacts because of uniformity in technologies used to generate the corresponding UI. For example, Android apps use Java for handling UI interactions and layout files to define UI structures, which can be mutated to cover various classes of bugs. In web testing, however, such

homogeneity of languages exists (e.g., a web app could be built using a combination of JavaScript, HTML, CSS, and PHP).

One of the foremost requirements for an effective mutation analysis tool is the set of mutation operators designed to generate artifact-specific transformations that can mimic programmer errors. While existing techniques [5], [6] have proposed several DOM specific mutation operators, they predominantly rely on mutating the source code artifacts of programming languages like Java or JavaScript. Existing DOM operators also do not cover the wide range of possible DOM transformations that may mimic application faults. As a matter of fact, to the best of our knowledge, there exists no prior research to establish the relationship between UI or DOM changes in the browser and application faults for web apps. Existing work [7] on categorization of web app faults is limited to the location (e.g., server, client) of the bug. Therefore, in this work, we manually analyze 250 bug reports from open source web app bug repositories to identify the UI manifestations of real faults, and design 16 mutation operators (MOs), which manipulate the interactive behaviour and appearance of web elements to mimic real faults.

We evaluate MAEWU on six open source web apps. Our results show that MAEWU (1) generates non-equivalent mutants, (2) consistently applies the mutants dynamically across test executions. The report generated show the mutation score of a given UI Test suite along with the failed mutants to help improve the test suite.

We summarize the contributions of our work as follows:

- We define a set of mutation operators for dynamic mutation of web pages in the browser and remove the need for modifying the source code to evaluate the efficacy of end-to-end test suites.
- We developed a mutation analysis framework, called MAEWU, that is universally applicable to end-to-end test suites of all web applications regardless of the back-end and front-end programming languages used for app development. MAEWU [8] is publicly available.
- We also provide a dataset of 250 labelled bug reports from open source web apps, and 300 labelled mutants that can be used by future researchers to determine the efficacy of UI test suites and mutation analysis techniques.

II. Background

In this section, we provide a brief overview of the current state of practice in web testing. Selenium Web Driver is one of the most popular tools for automated UI testing. It provides API in most popular programming languages such as Java and Python to remotely control web browsers and automate UI interactions. Listing 1 shows an example selenium UI test case taken from our subject set. Typically, test cases start by fetching the home page of the web app by using the provided url. Thereafter, a series of test actions are performed and test oracles are used to verify the resulting browser state according to a business case scenario of the web app.

As the web app evolves, such UI tests created for an earlier version of web app are used to validate existing functionality and detect any regression bugs that may have been introduced in the newer version. An adequate UI test suite should therefore cover the entire functionality of the web app through a combination of test actions and oracles to aid early detection of regression bugs through test failures.

Listing 1: Selenium JUnit Test Case

```
private WebDriver driver = new ChromeDriver();

@Before
public void setUp() {
  driver.get(app_url);
}

@Test
public void testCollabtiveLoginUser() throws Exception {
  driver.findElement(By.id("username")).sendKeys("↩
       username001");
  driver.findElement(By.id("pass")).sendKeys("password001↩
       ");
  driver.findElement(By.cssSelector("button.loginbtn")).↩
       click();
  driver.findElement(
            By.xpath("//*[@id=\"mainmenue\"]/li[2]/a")).↩
                 click();
  assertTrue(
      driver.findElement(By.cssSelector("body")).getText↩
           ()
      .matches("^[\\s\\S]*username001[\\s\\S]*$")
      );
```

As web apps can be incredibly complex with heterogeneous components written in multiple server-side and client-side programming languages, their bugs can be equally daunting to detect and fix. Typically, individual server-side components and client-side JavaScript are tested through unit testing while UI testing is used as a form of end-to-end testing to validate high level use case scenarios from an end-user point of view. Several researchers [7] have attempted to characterize web application bugs in terms of their location, significance and so on by analyzing the bug repositories of open source web apps. As UI tests only have access to the browser state, a UI test suite can only reveal application bugs have a direct manifestation in the UI. Such UI bugs are the focus of our mutation analysis for ascertaining the quality of the test suites.

III. UI Manifestation of Real Faults in Web Applications

In traditional mutation testing, mutation operators are designed to perform code changes that imitate programmer errors that cause application bugs. As majority of the modern web pages are automatically generated, mutating them may not directly imitate programmer errors. On the other hand, relying on mutation of source code artifacts written by programmers is impossible given the fragmented nature of web development ecosystem. Therefore, we decided to design mutation operators to imitate the manifestation of application bugs on the UI of web pages. In order to do so, we first needed to understand the UI manifestation of application bugs in web apps.

Manual analysis of real faults in web applications in the existing research [7] has focused on the location and root-cause analysis of faults in source code. Marchetto et al. [9]

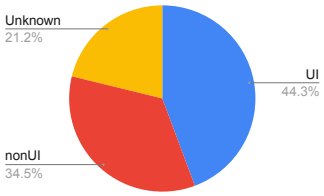

Fig. 1: Analyzed Bugs

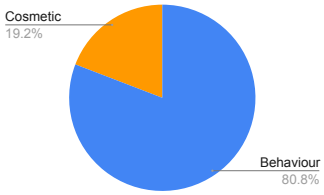

Fig. 2: Bugs with UI Manifestation

also defined 32 categories of faults that can be used for fault seeding in the web app source code. However, their work relied on introducing faults specific to program constructs and technologies in use at the time of publication.

In contrast, in this work, we are focused only on the front-end manifestation of bugs regardless of their root-cause and specific web development frameworks. Therefore, we collected real bugs reported for ten popular open source web apps shown in Table I with a minimum of 1000 downloads in sourceforge [20] or greater than 100 stars in GitHub. In total we collected 6331 reports tagged to be bugs from bug repositories. We then randomly selected 250 bug reports to be analyzed manually, where for each bug report, we ascertained the specific UI characteristics that were considered to be faulty, and created tags to reflect them. Our bug report analysis for UI impact resulted in a hierarchy of categories as shown in Figure 3.

Of the 250 bug reports we analyzed, we found that nearly 44% mention the impact on UI, while 35% did not have any UI manifestation as shown in Figure 1. In order to be able to manually analyze the bug reports, we familiarized ourselves with each of the web apps in terms of the provided UI functionality. However, we still could not assess the UI impact from the report for 21% of the reports. Of the bugs which impact the UI, we found that 81% of them had an effect on the behaviour of the web app as shown in Figure 2. Bugs categorized to have broken functionalities were either because of incorrect handling of events associated with web elements or content related faults. Only 19% of the bugs concerned the appearance of the web page, that were either found to be due to incorrect rendering of the web pages or incorrect css styling resulting in overlapped or incomprehensible content.

Using this study on the UI manifestation of real bugs in web applications, we design our mutation operators for dynamic mutation of web pages in the browser.

TABLE I: Bug Repositories

Name	**loc**	**languages**	**# Bugs**
tikiwiki [10]	4M	PHP, JavaScript	1975
reactive trader [11]	117K	C#, TypeScript	296
mrbs [12]	187K	PHP, JavaScript	495
pgweb [13]	262K	Go, JavaScript	553
tudu lists [14]	29K	Java, JavaScript	68
addressbook [15]	45K	PHP, JavaScript	173
crater [16]	80K	Laravel, Vue	380
claroline [17]	350K	PHP, JavaScript	336
koel [18]	475K	Laravel, Vue	1297
phplist [19]	1.3M	PHP, JavaScript	758

TABLE II: Mutation Operators and their types

Type	Operator Name	Abbr
Attribute	AttributeAdd	AAM
	AttributeDelete	ADM
	AttributeModify	AMM
EventHandler	EventHandlerAdd	EHAM
	EventHandlerDelete	EHDM
	EventHandlerModify	EHMM
Tree	TreeInsert	TIM
	TreeDelete	TDM
	TreeMove	TMM
Content	ContentInsert	CIM
	ContentDelete	CDM
	ContentModify	CMM
Style	StyleVisibility	SVM
	StyleColor	SCM
	StylePosition	SPM
	StyleSize	SSM

IV. Dynamic DOM Mutation

The functionality and data presented to the user through web app GUI is a DOM that is built together by back-end and front-end programs written in languages such as Java, PhP, JavaScript. UI test suites written using browser automation tools such as Selenium therefore indirectly test all these programs while exercising the DOM. Therefore, mutation operators targeting only client side JavaScript or server side JSP cannot assess the quality of the test suites as the resulting mutants only represent a subset of all possible changes that can occur in the DOM. In fact, espousing our sentiment, Mirzaaghaei et al. [21] argue that the coverage of UI test suites should be entirely determined using the GUI of the web app.

Considering the dynamic DOM state as a mutation target is interesting because it is essentially the output of the web app under test. In fact, the dynamic DOM is what end-to-end UI tests assess. UI tests are therefore expected to detect visual/-textual changes in the rendered web page through oracles, and detect behavioural changes by triggering UI actions. Mutation analysis of UI tests therefore will have to be designed to satisfy both these aspects of the UI testing to be considered useful.

Taking into account the UI impact of bugs, we designed

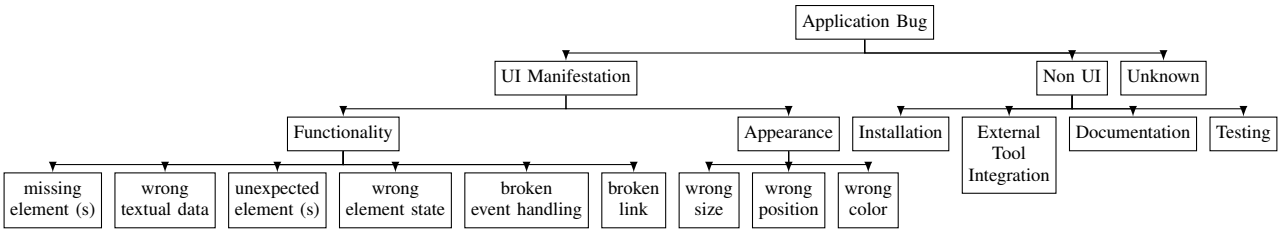

Fig. 3: Web Application Bug Categories

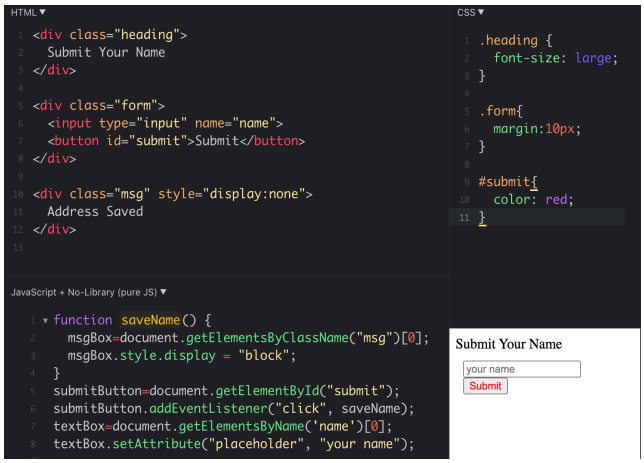

Fig. 4: Sample Web Page with Source Code

16 mutation operators, and placed them into five categories based on the aspect of web elements they target, as shown in Table II. Generated mutants can potentially imitate bug categories corresponding to UI functionality, appearance or both. For example, a changed id or class attribute can potentially impact the behaviour as well as appearance of the corresponding web element.

However, it is important to note the possibility that the mutation *may not impact* the web page functionality or appearance at all in any way. For example, mutating the position of an already invisible element will not change the web page.

In the rest of this section, we describe the mutation operators in greater detail and provide examples using the sample web page shown in Figure 4.

A. Attribute

In modern web apps, attributes are used to define characteristics or properties of web elements, and therefore changing attribute values can impact both their appearance and behaviour. Some attributes such as *"src"* for <img> elements are used to define the image urls. Attributes such as *"action"* for <form> elements can even specify server communication.

For example, consider the sample web page shown in Figure 4. Using css rules, appearance of *heading* and *form* is set using *class* selector, while *id* is used for *submit* button. Our three attribute MOs are currently configured to mutate commonly used attributes taken from current HTML standard [22]. However, developers can even use custom attributes to accomplish the same. In any case, being an extensible framework, MAEWU can be configured to use any set of attributes considered to be important for specific web apps, in order to generate interesting mutants.

B. Tree

Through the tree mutation operators, we aim to alter the DOM structure of the web page. When a tree operator is applied to a web element, the element as well as its children get affected. For example, in our sample web page, applying TreeDelete operator on the *div* of the class form will lead to its deletion as well as its child elements, the *input* element and the submit *button*.

The TreeMove operator can imitate the appearance "position" bugs as well as impact functionality of the moved subtree because of the changed parent through inherited event handling or style rules. The TreeInsert and TreeDelete operators are designed to imitate the functionality bugs– *unexpected-elements* and *missing-elements* respectively.

C. Content

Of the content related bugs we analyzed as part of the study, some of the root causes included errors fetching data from back-end, parsing form input data, and client-side scripting errors that prevent rendering of data.

In each of these analyzed bugs, the displayed textual content is either incorrect, unexpected, or completely missing from the web pages. Our three mutation operators – *ContentInsert*, *ContentDelete* and *ContentModify* – are aimed at imitating these content bugs.

It is also worth pointing out that most UI test cases typically access only interactive web elements and are likely to miss the content related bugs as a result.

D. Style

Cosmetic or appearance bugs we analyzed related to unexpected positioning, size and color of specific web elements, primarily caused by wrong css properties computed in runtime. Our analysis of appearance related bugs revealed three computed (from static rules) CSS properties – *color, position, size* – We designed the three style MOs – *StyleColor, StylePosition, StyleSize* – that mutate corresponding css properties in runtime to imitate bugs causing content to be either incomprehensible, place elements in unexpected positions or have unexpected sizes.

Our fourth style MO, *StyleVisibility* imitates the functionality bugs *missing-elements* and *unexpected-elements* by toggling the *visible* CSS property of web elements.

E. Event

We designed three mutation operators to cover the bugs related to *broken event handling*. Bugs of this category affect the behaviour of the interactive elements like buttons while often having no apparent change in the appearance and visible content of the web pages. In order to imitate such bugs, we created three mutation operators that insert, remove or modify the event handlers of web elements.

As such, broken event handling can also happen because of a variety of reasons such as bugs on the server side, broken server communication, or even bugs in JavaScript libraries being used on the client side. However, since we are interested in imitating the eventual behaviour observed by the end users, we directly modify the event handlers for the web elements. In order to ensure the JavaScript code for event handlers themselves are valid, we reuse already seen event handlers for other elements within the web app.

V. Technique

Our technique, MAEWU, aims to assess the mutation score for a given UI test suite and a web app URL by dynamically applying mutation operators to browser states. MAEWU contains four main components 1) Trace Generator (TG), 2) Trace Analyzer (TA), 3) Mutant Generator (MG), and 4) Mutation Engine (ME).

A high level architecture of MAEWU shown in Figure 5 indicates the inputs and outputs for each of the components. The mutation analysis performed by MAEWU can be divided in two main processes. While the first process concerns generation of a set of candidate mutants, the second part concerns evaluating the test suite efficacy using the candidate mutants.

To generate the candidate mutants, given a test suite, TG collects the test trace as a series of dynamic DOM states associated with test steps, and TA analyzes them to identify the reappearance of web elements across states by using a page fragmentation technique. MG then creates candidate mutants by applying mutation operators on the identified web elements. To evaluate the efficacy of the test suite, ME applies each candidate mutant to the dynamic DOM in the browser while executing the test suite.

In the rest of this section, we describe each of our components in detail.

A. Trace Collection

Given a UI test suite, the trace generator (TG) captures a test suite trace as a sequence of trace elements, recording browser states as defined in 1 before and after each test step. In addition, an "observer" script that runs in the browser records JavaScript accesses to web elements for each browser state during test execution.

Algorithm 1: Extract Logical Web Elements

```
   Input: states = [S1, S2..]      /* Set of Browser States */
   Output: LWE  /* Set of Logical Web Elements (ω) */
1  LWE=[]
2  foreach S ∈ states do
3      elems = getWebElems(S)  /* all elements of state */
4      foreach ε ∈ elems do
5          foreach ω in LWE do
6              if belongsTo(ε,ω) then
7                  ω.add(ε)    /* belongs to cluster */
8                  added ← true
9          end
10         if not added then
11             ω_new ← [ε]        /* create new ω */
12             LWE.add(ω_new)
13     end
14 end
15 Function belongsTo(ε, ω):
16     ε' ← lwe[0]
17     F1 ← getFragment(node1)   /* fragments from VIPS */
18     F2 ← getFragment(node2)
19     if TreeComp(F1, F2) = 0 then   /* Tree Edit Distance */
20         if XPath(node1) = XPath(node2) then
21             return True     /* same relative XPath */
22         return False
23     return False
24 End Function
```

Definition 1 (**Browser State** (S))**.** is a tuple <D,V, K> where D is the HTML source or DOM and V is the screenshot of the web page. K is the JavaScript access map for the browser state recorded by the observer script, where each map entry corresponds to a web element and its accesses.

A trace element as defined in 2 records the browser state transition in the web app caused by the test step execution.

Definition 2 (**Trace Element**)**.** is a tuple (ε, S_b, S_a, α), where the action α is performed upon the web element ε in the browser state S_b results in the transition to the browser state S_a.

B. Trace Analysis

Once the trace is collected for a web app, the trace analyzer (TA) first extracts a list of all web elements in the recorded browser states and a set of text tokens. The set of text tokens which we call the mutation data tokens (Δ) are extracted from content nodes as well as attribute values. Thereafter, since our approach of web app mutation use web elements from dynamic DOM as mutation artifacts, we designed the trace analyzer (TA) to cluster equivalent web elements into logical web elements (ω) in order to achieve a consistency in mutation. Formally, logical web elements are defined in 3.

Definition 3 (**Logical Web Element** (ω))**.** is a tuple <E, κ>, where E is the set of concrete web elements $\{\varepsilon_1, \varepsilon_2, ..\}$ in which any two web elements $\varepsilon_i, \varepsilon_j$ are equivalent to each other. κ is the combined list for JavaScript accesses of all web elements in E, where access for each element is extracted from the access map K of its parent state (S).

Consider the two example pages in Figure 6 that contain several web elements in common. If we decide to mutate the

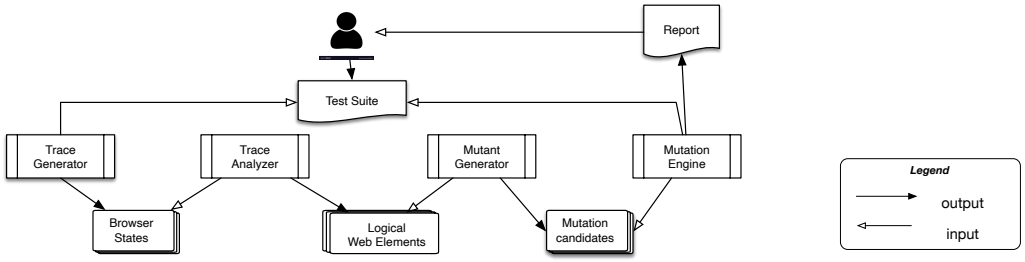

Fig. 5: Technique Overview.

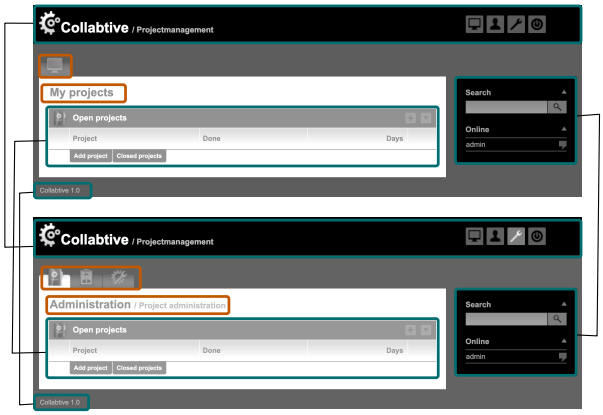

Fig. 6: Example Web Pages

"search box" in page1, we need to ensure the same mutation is applied to it in the page2 as well if a reliable mutation score for the UI test suite is to be computed. A similar problem does not arise for the traditional source code mutation since the applied mutation is available for every instantiation of the corresponding line of code, regardless of the dynamic program state. For example, a mutation applied to a HTML source artifact in the server will be available each time the artifact is accessed from the browser.

However, establishing the equivalence of web elements is challenging because of the dynamic nature of modern web apps. Existing research shows that techniques relying on attributes such as ids to compare web elements tend to be unreliable, because such attributes are often generated dynamically. Similarly using XPath locators in web pages is also not desirable because we want to compare individual web elements across different web pages. For example in Figure 6, the XPath for "add Project" in the two pages is not the same.

Our solution is based on the observation that a single web page does not necessary provide a singular functionality. Instead, each web page UI is stitched together dynamically and contains independent UI components such as navigation bars that reappear in different web pages. We associate the ownership of web elements to smaller page fragments instead of the entire web pages, and use the equivalence of these fragments to establish similarity of web elements.

Our element extraction algorithm shown in algorithm 1 uses a popular page segmentation technique VIPS [23] to generate smaller page fragments, compare fragments using a tree comparison technique [24], and, uses relative XPaths of web elements inside these fragments for establishing their equivalence.

C. Generating Mutation Candidates

Given a set of logical web elements (ω) and available mutation data tokens (Δ), Mutant Generator (MG) generates a set of all possible mutation candidates by selecting appropriate mutation operators based on the characteristics of the web element and a random mutation data token if required.

Definition 4 (**Mutation Candidate** (C))**.** is a tuple <ω, O, δ> where ω is the logical web element upon which the mutation operator O is applied using the optional mutation data δ.

However, modern web pages are notoriously heavy [25], [26], where an "average" web page is 2MB in size, can contain more than 600 web elements of 32 different types. Without a notion of significance or importance associated with each mutant to allow for a selection strategy, the total number of mutants to analyze can quickly become unmanageable especially given the resource intensive nature of UI testing.

In this work, we employ a biased-spread random mutant selection strategy where the probability (Equation 1) of selecting a mutant depends on its score (Equation 2). Our score for a candidate combines 1) four static features of the web element – $isLeafNode$, $hasText$, $isInteractive$, $isDisplayed$, 2) three dynamic features based on the collected web element trace – $numRepetitions$, $numTestAccesses$, $numJavaScriptAccesses$, and 3) its relationship to already selected mutants. The score is positively impacted by high static (St) and dynamic (Dn) scores, and negatively impacted by the spread score (Sp) based on the presence of already selected mutants for the same web element.

The selection probability is defined as

$$Pr(C_j^a) = \frac{score(C_j^a)}{\sum_{i=1}^{n} score(C_i^a)} \quad (1)$$

where a, b and c are constants such that $(0 < a, b, c < 1)$, the set of 'n' available candidates ($\{C_{1..n}^a\}$) and the set of 'm' already selected candidates ($\{C_{1..m}^s\}$), and the candidate score is

$$score(C^a) = a * St(\omega^a) + b * Dn(\omega^a) - c * Sp(C^a, \{C_{1..m}^s\}) \quad (2)$$

TABLE III: Experimental Subjects

Subject			*Test Suite*		
	Version	*Loc*	*Cases*	*Loc*	*Loc*
AddressBook	8.0.0	16298	27	49	1325
Claroline	1.11.10	352537	40	46	1822
Collabtive	3.1	264642	40	48	1935
MantisBT	1.1.8	141607	41	43	1748
MRBS	1.4.9	34486	22	51	1114
PPMA	0.6.0	575976	23	54	1232
Total		866995	196	47	9176

Based on their relevance to the corresponding MO, the static features of web element capture the usefulness of a given candidate based on its likelihood of imitating a bug. For example, a candidate with the MO "ContentDelete" and a *visible* page heading are likely to imitate bugs related to missing content. On the other hand, the dynamic features capture the importance of a given web element based on the frequency of its appearance and its extent of usage during test execution. While the static features we define are inspired from AST based features for program statements [27], dynamic features are similar to ranking based on execution traces in source code mutation [28]. Our spread score (Sp), inspired from the spread-random mutant selection strategy which selects only one mutant per source code statement, decreases the candidate score instead of filtering them out.

We compute Static (St) and dynamic (Dn) scores as a sum of all corresponding feature values, where values for a static feature (boolean) is 1 if true or 0 otherwise. Finally, the probability of selecting a mutant (Equation 1) is then computed as the ratio of a candidate score to the total score of all candidates.

D. Mutating Dynamic DOM and Mutation Score

For each mutation candidate ($C =< O, \omega >$), the Mutation Engine (ME) resets the web app and runs the test suite while applying the mutation operator (O) to all the concrete instances (ε) of the logical web element (ω).

Existing techniques on mutation analysis for web applications mutate the source code and compare the output DOM to determine the mutation kill score [5], [6] where there are observable changes. This often involves manual analysis of source code [28] to determine equivalent mutants as well. In our analysis, the mutant is considered to be *killed* by the test suite if any of the test cases fail either because of a failing test action or a test oracle.

VI. Evaluation

To assess the efficacy of our mutation testing approach, we answer the following research questions.

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

RQ2 How useful are the generated mutants for improving end-to-end UI test suites?

TABLE IV: Bug Severity based on User Perception

Description of Severity	**Severity Score**
I did not notice any fault	0
I noticed a fault, but I would return to this website again	1
I noticed a fault, but I would probably return to this website again	2
I noticed a fault, and I would not return to this website again	3
I noticed a fault, and I would file a complaint	4

TABLE V: Mutant Stubbornness

Required UI Testing effort	**Stubbornness Score**
No action needed	0
Locate the element in the page	1
Perform action on the element	2
Assert content, attribute or CSS property of the element	2
Perform action and Assert a property in the resulting browser state	3
Perform action and navigate to a different page to assert the effect of the action	4

TABLE VI: Mutant Generation per subject

	addressbook	***claroline***	***collabtive***	***mrbs***	***mantisbt***	***ppma***	***total***
# candidates	13K	72K	329K	92K	61K	21K	586K
# selected	50	50	50	50	50	50	300
Lwe size	45	37	74	69	45	69	56
Static Score	2.82	3.1	2.88	2.98	3.12	3.1	3
Dynamic Score	63	44	77	73	65	79	67
# Web Elements	2.2K	1.8K	3.6K	3.4K	2.2K	3.4K	17K

TABLE VII: Mutants Generated by MAEWU

	# Mutants	***Non-Eq***	***Equiv***	***% Non-Eq***	***% Equiv***	***# Killed***	***Mutation Score***
addressbook	50	48	2	96	4	14	28
claroline	50	46	4	92	8	6	12
collabtive	50	43	7	86	14	6	12
mantisbt	50	48	2	96	4	6	12
mrbs	50	47	3	94	6	9	18
ppma	50	46	4	92	8	15	30
Total	300	278	22	93	7	56	19

TABLE VIII: Mutant quality Per Operator

	AAM	*ADM*	*AMM*	*EHAM*	*EHDM*	*EHMM*	*SCM*	*SPM*	*SSM*	*SVM*	*TDM*	*TIM*	*TMM*	*CDM*	*CIM*	*CMM*	*Total*
#Mutants	9	10	13	15	4	4	29	26	37	37	27	30	17	17	6	19	300
#Non-Eq	4	10	12	12	2	4	22	26	36	35	27	30	17	17	6	18	278
#Killed	0	2	5	0	0	0	0	7	6	8	13	1	6	3	0	5	56
Non-Eq %	44	100	92	80	50	100	76	100	97	95	100	100	100	100	100	95	93
Mutation Score	0.00	0.20	0.42	0.00	0.00	0.00	0.00	0.27	0.17	0.23	0.48	0.03	0.35	0.18	0.00	0.28	0.20
Bug Severity	2.0	2.3	2.5	2.5	2.5	2.5	0.7	1.3	0.8	2.2	2.4	1.8	1.4	2.0	1.8	2.0	1.7
Stubbornness	2.5	2.8	3.1	3.0	3.0	2.8	2.0	2.0	2.0	1.9	1.5	2.0	2.1	2.0	2.0	2.1	2.1

A. Experimental Setup

We use six open-source web apps as our subject systems, each with a manually written JUnit Selenium UI test suite used in previous web testing research [29]. Table III lists the name, version and size of our subjects and the corresponding test suite characteristics. All our experiments were run on a Red Hat Enterprise Linux Server (RHEL-7) and Chrome-v84 web browser.

B. Competing Techniques

We found two existing mutation analysis tools for web app UI testing – WebMuJava and AjaxMutator. WebMuJava is developed by Praphamontripong et al. [5], [30] for JSP and Java Server based web apps. It is not publicly available. Nishiura et al. [31] developed AjaxMutator to mutate Ajax and DOM API calls used in client-side JavaScript of web apps. We explain the reason for not including AjaxMutator below.

1) Issues using AjaxMutator: AjaxMutator uses the Rhino JavaScript parser to extract mutation targets for four specific features of client-side JavaScript – event registration; timer; Ajax calls; and DOM API to *append* and assigning *attribute*. The current implementation for AjaxMutator takes a *single* JavaScript file as input and generates mutants. It then runs a given Selenium Test Suite for *all* the generated mutants to compute the mutation score. However, we were unable to use it on our subjects.

The first issue we faced is regarding the input JavaScript file expected by AjaxMutator. All of our test subjects, which are modern web apps contained JavaScript in multiple files and libraries along with "InlineHTML" within other program files such as PHP, JSP. In order to get AjaxMutator to work on our subjects, we wrote a file parser to extract JavaScript from source files. However, Rhino could not parse these extracted JavaScript files, with sizes exceeding 50K lines for four of our subjects where upon we spent considerable amount of time trying to clean the files manually without success.

Secondly, for the two subjects we could generate mutants, we found no clear mechanism to reliably apply the generated mutants into the web app when we use this extracted JavaScript file. Because of this limitation, we could not even assess the resulting impact of these mutants on the actual functionality of the web app.

C. Procedure and Metrics

For each of our subjects, we configure MAEWU with the URL, the accompanying test suite, and a maximum limit of 50 mutants to be selected. Once MAEWU generates the mutants and computes the mutation score, we manually analyzed the impact on the behaviour of the web app for each of these selected mutants.

1) Analyzing mutants: In order to verify the impact of the mutation on the web page, we compare the live mutated state to the original state first in terms of visual appearance, and second, by exercising the functionality offered by target web element. If required, we also analyze the client-side JavaScript to understand the impact of the mutation.

We classify the mutants that impact neither the functionality nor the appearance of the web page to be equivalent. We then manually label the generated mutants to assess their 1) perceived bug severity, and 2) mutant stubbornness to determine their quality. [32]

The mutation score, bug severity score and stubbornness are computed only for non equivalent mutants.

2) Computing Mutation Score: We compute the mutation score as the percentage of killed non-equivalent mutants to the total number of non-equivalent mutants for each of the subject apps.

3) Computing Bug Severity: Based on the mutant impact on the UI, we compute a bug severity score shown in Table IV using 18 manually labelled boolean features adapted from previous work [33]. In the interest of space, we skip discussing the actual adaption which is available along with the full decision tree to compute severity in our tool repository [8].

4) Computing Mutant Stubbornness: Existing work defines stubbornness through either, source code features [34] that make certain mutants difficult to kill or, by their relationship to the test suites [35], [36] such as the number of tests that can kill the mutant. The difficulty in finding the right program input [34] to kill the mutant is the common theme in categorizing mutants to be stubborn.

In this work, we model our stubbornness score based on the amount of effort required for a UI tester to kill the mutant. The stubbornness score lies between 0 and 4 as shown in Table V. On the one extreme are easy-to-kill mutants that cause test failures by virtue of just reaching the mutated browser state.

For example, it the mutation results in a blank page. On the other end of the extreme are the mutants that can result in back-end changes that infect other browser states while keeping the mutated state unaffected in terms of appearance or functionality. For example, a wrong transformation of user input in the infected state that is saved and retrieved from a database in another page.

Next, we discuss the results of our analyses and to save space, we will use the abbreviated operator names for the rest of the paper as defined in Table II.

D. Results

1) RQ1: Table VI shows the total number of candidates extracted by the Mutant Generator, and the characteristics of the 50 selected candidates per subject. On an average, the logical web element (ω) size for each selected candidate is 56, with *collabtive* having the highest repetition of concrete web elements at 78. Overall, nearly 17K concrete web elements were selected to be mutated in order to apply these 300 mutations in the browser at runtime. The average static score which determines the quality of mutant based on the web element and operator characteristics is 3, whereas the average dynamic score is 67. Note that all the selected logical web elements were *covered* by the test suites either directly by performing action on them during test execution or indirectly by causing a JavaScript to access these elements in the browser as recorded by our ObserverScript.

The results of our manual classification of these mutants based on their impact on UI functionality and appearance is shown in Table VII. On an average, 93% of the mutants generated by MAEWU were found to be non-equivalent. As shown in Table VIII, SCM operator is responsible for 7 out of the 22 equivalent mutants that we found in total. In each of these 7 equivalent mutants, we found that the mutation applied by SCM using the css property "color" has been overridden by a css rule for the child elements. AAM generated 5 equivalent mutants because the added attributes like "id" were not used for any of the JavaScript in the page, rendering that attribute addition meaningless.

The most interesting and hard to classify equivalent mutants were created by EHAM and EHDM operators which generated 3 and 2 equivalent mutants respectively. In our implementation, we used the JavaScript "element.addEventListener" API to add and replace event listeners. We provide a random event handler function that is recorded by the Trace Generator for the subject. However, we found two reasons for EHAM having no impact on app functionality. First, because the functions being called within the new event listener were not available in the browser state being mutated, and therefore they just fail silently with no change to functionality in case of EHAM. Second, our mutation script could not override the default behaviour of the elements as defined by the browser. In future, we plan to automatically detect when such default behaviour impacts JavaScript manipulation of event listeners and select candidates accordingly.

The rest of the operators had a close to or equal to 100% success rate in generating non equivalent mutants.

2) RQ2: Table VIII shows the results of our manual analysis of mutants as well as the mutation scores for the test suites for each of our mutation operators.

Our manual analysis revealed that all three event handlers MOs have a predictably high bug severity score because they are designed to break the behaviour of the web elements which is perceived to be the most severe fault in a web page. While maintaining a high severity score, EHAM, EHDM and EHMM also created mutants with the highest stubbornness score because killing these mutants required both performing action as well as verifying the result of the action. A mutation score of 0 for the UI test suites being evaluated is also indicative of the difficulty in killing these mutants.

Interestingly, ADM and AMM also have a high bug severity score similar to event handler MOs because they are also capable of affecting the behaviour. For example, deletion of the "href" and "input.type" attribute was particularly effective in breaking the element behaviour. However, the test suites were able to kill these mutants more frequently with 42% of AMM mutants being killed because they impact common interactive web elements that are often used by the test suites.

TDM operator has also generated mutants with a high bug severity primarily because it causes the pages to lose information and functionality by deleting parts of the page. However, these mutants are very easy to kill with a stubbornness score of only 1.5. Often, to kill the mutant, it is enough for a test script to try and locate the web element

Indeed, other MOs that can cause loss or change of information – SVM, CDM, CIM and CMM – have a high bug severity score and generate stubborn mutants because a tester has to either perform an action or assert the expected property of the target web element in order to kill the mutant. Interestingly, however, TMM which also can cause information loss through DOM hierarchy manipulation often caused only cosmetic defects, and therefore has a lower severity score.

In terms of usefulness, SCM generated the mutants with worst severity of 0.7 while generating majority of the equivalent mutants in our entire experiment. SSM and SPM have a similarly low severity score overall but generated mutants that impacted the behaviour of the page by blocking or reducing the accessibility of content or functionality of other web elements.

Overall, we found that the operators that are able to generate mutants of high bug severity and high stubbornness are most useful in exposing the weaknesses of the UI test suites. On an average, MAEWU generated mutants with a severity of 1.7 and a stubbornness score of 2.1, while exposing the limitations of existing test suites for our subjects which have a low mutation score of 0.20.

E. Discussion

1) Web App Dynamism and Test Fragility: In RQ2, we used two factors, bug severity and stubbornness score, in understanding if a mutant can be useful in improving existing UI test suites. However, an important aspect of modern web

apps that we did not take into consideration in the current work is the presence of dynamic data. For example, content or element properties like 'id' that are dynamically generated and should not be considered to be bugs. So, generating mutants that may be similar to these dynamic changes to the web page will not be useful in determining the fault revealing capabilities of the test suites. Indeed, such mutants may deter practitioners from employing the framework, as these mutants are similar to equivalent mutants.

However, one of the biggest challenges of maintaining an end-to-end UI test suite for web apps is the fragility of web element locators [37], which require costly manual analysis and test maintenance. An interesting idea would be to use the MOs designed in this work, and select mutants that can reveal fragile test locators and test oracles.

2) Bug Severity for modern web apps: The existing study on the bug severity [33] used in our evaluation was primarily based on older web apps with limited dynamism. The study classifies any css related problem to be cosmetic in nature and gives a very low severity score. However, modern web apps rely heavily on fluid layout models in order to make the web app functionality accessible on multiple device and display configurations. While SCM, SSM, SPM generated mutants of very low severity on a fixed display configuration, these can be valuable in validating layout features and revealing layout bugs. Therefore, we believe a bridging study to better model the appearance related bugs that impact modern web apps is needed to determine bug severity for such mutants.

VII. Related Work

For mutation analysis of web apps for assessment of UI test suites, there are only two existing research papers. Praphamontripong et al. [5], [30] define and implement mutation operators for JSP and Java Server based web applications in a tool called webMuJava which extends general Java based mutation operators. Nishiura et al. [31] defined mutation operators specific to client-side JavaScript of web applications used for DOM manipulations. To overcome the limitations of existing work owing to their usage of source code mutation, we propose to mutate dynamic DOM to mutate GUI functionality. Maezawa et al. [38] validates ajax code using three delay based mutation operators.

In addition, for web testing in general, Shahriar et al. [6] defined 11 mutation operators on PHP and JavaScript source code to find bugs related to cross-site scripting. Walsh et al. [39] implement CSS mutation operators that change the CSS rules related to the size (e.g., width) of web elements in order to simulate cross-browser page rendering faults. Mirshokraie et al. [28] developed mutation operators specific to JavaScript in web apps in addition to generic JavaScript operators. We do not consider either of these tools to be competing techniques to our work because they are neither intended to assess UI test suites, nor necessarily applicable to all web apps.

In broader area of mutation analysis for GUI applications, Alegroth et al [40], apply mutations to desktop Java application to evaluate GUI testing approaches. Oliveira et al. [41] developed scripts to automate mutant generation for seven of the 18 mutation operators defined in [40] to show that GUI mutation operators are better than traditional method level Java mutation operators in seeding GUI faults in applications. Linares-Vásquez et al., [42], [43] created a taxonomy of Android bugs with the purpose of defining source-code mutation operators for Android apps. Deng et al. [4], [44] defined mutation operators to change core components of Android apps (e.g., intents, event handlers, XML files and activity lifecycle). Additionally, Luna et al. [45] presented Edroid, a tool that uses 10 mutation operators oriented to validate changes in the GUI.

In both the fields, namely desktop GUI and mobile testing, source code mutation has been employed to a good effect because of the homogeneity of the programs under test.

VIII. Threats to validity

External validity threats concern the generalization of our findings since we used a limited number of subject apps and analyzed only 300 mutants overall. We have chosen six subject apps used in previous web testing research, pertaining to different domains, and fully randomized the mutants to be analyzed in order to mitigate the threat. Threats to *internal validity* come from the manual labelling of mutant categories and features, which was unavoidable because no automated method could provide us with the required ground truth. The manual bug analysis, mutant analysis and labelling was performed by the first author, and the methodology was developed together by the two authors by analyzing example bugs, mutants independently and establishing a discussion to resolve conflicts. For bug severity analysis, we used a labelling methodology outlined in prior work to mitigate the threat to validity. For reproducibility of our findings, we made our tool publicly available [8] along with usage instructions and used subject systems.

IX. Conclusion and Future Work

Despite the significance of UI test suites in validating web app functionality, currently, no mutation analysis tool exists for ascertaining their fault-finding capabilities. Existing tools for web app mutation testing rely on source code mutation and therefore cannot be universally applied because of the heterogeneous web app development ecosystem. In this work we developed MAEWU, an extensible mutation analysis framework for web apps which mutates the dynamic DOM during test execution. Given only the web app URL and a UI test suite to assess, MAEWU is able to automatically extract unique web elements in the web app and generate non-equivalent mutants that imitate UI manifestation of real web app bugs, select useful mutants based on the web element characteristics and perform mutation analysis to reveal the limitations of the test suite. As part of the future work, we plan to improve the mutant selection strategy by incorporating human feedback to compute mutant score.

REFERENCES

[1] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 435–445. [Online]. Available: https://doi.org/10.1145/2568225.2568271

[2] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 214–224. [Online]. Available: https://doi.org/10.1145/2786805.2786858

[3] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[4] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[5] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, April 2010, pp. 132–141.

[6] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, ser. IWSESS '09. USA: IEEE Computer Society, 2009, p. 47–53. [Online]. Available: https://doi-org.ezproxy.library.ubc.ca/10.1109/IWSESS.2009.5068458

[7] Y. Guo and S. Sampath, "Web application fault classification - an exploratory study," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 303?305. [Online]. Available: https://doi.org/10.1145/1414004.1414060

[8] "MAEWU - Mutation analysis framework for E2E web testing," https://github.com/mutationwebapp/maewu, 2021.

[9] A. Marchetto, F. Ricca, and P. Tonella, "Empirical validation of a web fault taxonomy and its usage for fault seeding," in *2007 9th IEEE International Workshop on Web Site Evolution*, 2007, pp. 31–38.

[10] tikiwiki, " TikiWiki," https://sourceforge.net/projects/tikiwiki/, 2021.

[11] Reactive Trader, " Reactive Trader," https://github.com/AdaptiveConsulting/ReactiveTraderCloud/, 2021.

[12] mrbs, " Meeting Room Booking System," https://mrbs.sourceforge.io/, 2021.

[13] PGWeb, " PGWEB," https://sourceforge.net/projects/pgweb.mirror, 2021.

[14] Tudu Lists, " Tudu Lists," https://sourceforge.net/projects/tudu/, 2021.

[15] PHP AddressBook, "Simple, web-based address & phone book," http://sourceforge.net/projects/php-addressbook, 2021, accessed: 2018-10-01.

[16] Crater, " Crater," https://sourceforge.net/projects/crater.mirror/, 2021.

[17] Claroline, "Claroline. Open Source Learning Management System." https://sourceforge.net/projects/claroline/, 2021.

[18] Koel, " Koel," https://sourceforge.net/projects/koel.mirror/, 2021.

[19] PHP List, " PHP List," https://sourceforge.net/projects/phplist/, 2021.

[20] SourceForge, "SourceForge." https://sourceforge.net/, 2021.

[21] M. Mirzaaghaei and A. Mesbah, "Dom-based test adequacy criteria for web applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 71–81. [Online]. Available: https://doi.org/10.1145/2610384.2610406

[22] "HTML Living Standard ," https://html.spec.whatwg.org/.

[23] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Vips: a vision-based page segmentation algorithm," 01 2003.

[24] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Trans. Database Syst.*, vol. 40, no. 1, pp. 3:1–3:40, Mar. 2015.

[25] "HTTP Archive," https://httparchive.org/reports/page-weight.

[26] "The average web page ," https://www.advancedwebranking.com/html/.

[27] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," vol. 25, no. 1, pp. 434–487. [Online]. Available: https://doi.org/10.1007/s10664-019-09778-7

[28] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429–444, May 2015.

[29] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. ACM, 2019, p. 12 pages.

[30] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 102–111.

[31] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden, "Mutation analysis for javascript web applications testing," vol. 2013, 01 2013.

[32] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 537–548. [Online]. Available: https://doi.org/10.1145/3180155.3180183

[33] K. Dobolyi and W. Weimer, "Modeling consumer-perceived web application fault severities for testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 97–106. [Online]. Available: https://doi.org/10.1145/1831708.1831720

[34] X. Dang, X. Yao, D. Gong, and T. Tian, "Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 334–348, 2020.

[35] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S. F. Andler, P. Potena, and M. Bohlin, "Using mutant stubbornness to create minimal and prioritized test sets," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 446–457.

[36] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 919–930. [Online]. Available: https://doi.org/10.1145/2568225.2568265

[37] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *Proceedings of 8th International Conference on Software Testing, Verification and Validation*, ser. ICST 2015. IEEE, 2015, pp. 1–10.

[38] Y. Maezawa, K. Nishiura, H. Washizaki, and S. Honiden, "Validating ajax applications using a delay-based mutation technique," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 491–502. [Online]. Available: https://doi.org/10.1145/2642937.2642996

[39] T. A. Walsh, P. McMinn, and G. M. Kapfhammer, "Automatic detection of potential layout faults following changes to responsive web pages (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 709–714.

[40] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[41] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[42] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 233–244. [Online]. Available: https://doi.org/10.1145/3106237.3106275

[43] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 33–36.

[44] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Inf. Softw. Technol.*, vol. 81, no. C, p. 154–168, Jan. 2017. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.04.012

[45] E. Luna and O. E. Ariss, "Edroid: A mutation tool for android apps," in *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2018, pp. 99–108.