DOM-Based Test Adequacy Criteria for Web Applications

Mehdi Mirzaaghaei
University of British Columbia
Vancouver, BC, Canada
mehdi@ece.ubc.ca

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

ABSTRACT

To assess the quality of web application test cases, web developers currently measure code coverage. Although code coverage has traditionally been a popular test adequacy criterion, we believe it alone is not adequate for assessing the quality of web application test cases. We propose a set of novel DOM-based test adequacy criteria for web applications. These criteria aim at measuring coverage at two granularity levels, (1) the percentage of DOM states and transitions covered in the total state space of the web application under test, and (2) the percentage of elements covered in each particular DOM state. We present a technique and tool, called DomCovery, which automatically extracts and measures the proposed adequacy criteria and generates a visual DOM coverage report. Our evaluation shows that there is no correlation between code coverage and DOM coverage. A controlled experiment illustrates that participants using DomCovery completed coverage related tasks 22% more accurately and 66% faster.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Algorithms, Experimentation

Keywords

Test adequacy criteria, DOM, coverage, web applications

1. INTRODUCTION

To check the correct behaviour of their web applications, developers today write test cases using automation frameworks such as CasperJS [1] or Selenium [10]. These frameworks provide APIs for instantiating a web browser capable of executing JavaScript, HTML, and CSS code. Once the application is loaded into the browser, test cases written in these frameworks directly interact with the application's runtime Document Object Model (DOM) [3].

To assess the quality of their test cases, developers measure *code coverage* of client-side JavaScript as well as server-side (e.g., Java, PHP) code. Although code coverage has traditionally been a popular test adequacy criterion [25], we believe it alone is not an adequate metric for assessing the quality of web application test cases.

Web application test cases generally interact with the DOM to (1) write to form inputs, (2) generate events on specific DOM elements to navigate and change the state of the application, and (3) read and assert DOM element properties. Although these actions can trigger the execution of code (e.g., JavaScript, Java) indirectly, the test cases merely check the correctness of the DOM elements and their properties. As such, we believe a proper coverage metric should be geared towards the DOM of the web application under test, in addition to its code execution. In this perspective, the DOM itself should be considered as an important structure of the system that needs to be adequately covered by the test suite.

In this paper, we propose a set of DOM-based test adequacy criteria for web applications. These criteria aim at measuring web application coverage at two different granularity levels, namely (1) the percentage of all DOM states covered in the total state space of the application, and (2) the percentage of all elements covered in a particular DOM state. In general, our goal is not to replace code coverage but to complement it with *DOM coverage*, a metric more tangible for web developers and testers.

We present a technique that automatically extracts and measures the proposed DOM-based adequacy criteria and generates a visual DOM coverage report. This report helps web developers to spot *untested* portions of their web applications. Our work makes the following main contributions:

- A set of test adequacy criteria targeting the DOM at two granularity levels, namely:
  1. *Inter-state criteria* focus on the overall state space of the application and require each DOM state/-transition to be covered at least once;
  2. *Intra-state criteria* examine each covered DOM state separately and require each DOM element in the state to be covered at least once.
- A technique and algorithm to dynamically compute the coverage criteria;
- An open source tool implementing our approach, called DomCovery, which generates an interactive visualization of the coverage report;

ISSTA '14, July 21–25, 2014, San Jose, CA, USA

Figure 1: Running example: a simple gift card purchasing web application.

```
1  Enter amount: <input id="amount" value="50">
2  <input value="add to gift card" type="button" ↩
       onclick="addTotal (parseInt(amount.value)+↩
       adminFee(amount.value))">
3  ...
4  <a href="/pre">pre-loaded gift cards</a>
5  ...
6  <input type="button" id="first" onclick="↩
       addTotal(150);" value="$100">
7  <input type="button" id="second" onclick="↩
       addTotal(250);"  value="$200">
8  <input type="button" id="third" onclick="↩
       addTotal(350);" value="$300">
9  <input type="button" id="fourth" onclick="↩
       addTotal(400);" value="$400">
10 ...
11 Total:<div id="payable">$0</div>
12 <span id="xrt"/>
```

Figure 2: String DOM representation of state 1 of the running example.

```
1  var currencyExchangeRate = getExchangeRate();
2
3  function addTotal(x) {
4      var total = x * currencyExchangeRate;
5      $("#payable").html("$" + total);
6      $("#xrt").html("Exchange rate: " + ↩
           currencyExchangeRate);
7  }
```

Figure 3: JavaScript snippet of the running example.

- An empirical evaluation of DomCovery assessing its efficacy and usefulness, through a controlled experiment. The results show that participants using DomCovery were able to accomplish test coverage understanding tasks 22% more accurately and 66% faster. Our analysis also shows that the proposed DOM coverage criteria cover other dimensions of web applications that are not correlated with traditional code coverage metrics.

2. BACKGROUND AND MOTIVATION

Testing modern web applications is challenging since multiple languages interact with each other at runtime to realize the application: (1) HTML code defines the structure of the page, (2) Cascading Style Sheets (CSS) control the style and layout of the elements, (3) Server-side code (e.g., Java, Ruby) provides dynamic content, and (4) JavaScript code connects everything together by interacting with the server-side and dynamically mutating the HTML structure and its style, to present application state changes.

The final result of all these interactions is manifested through the Document Object Model (DOM) [3] and presented to the end-user. The DOM is a dynamic tree-like structure representing user interface components in the web application. DOM elements can be dynamically generated, changed, or removed at runtime.

To avoid dealing with all these complex interactions separately, testers typically treat the web application as a black-box and test it through its manifested DOM, using testing frameworks such as Selenium [10].

Motivating Example. We use a simple gift card purchasing web application, as shown in Figure 1, to motivate the problem. We also use this application as a running example to illustrate our approach throughout the paper.

This simple web application allows the users to add gift cards with different values to their shopping carts. The user can either enter an arbitrary amount manually or choose from predefined gift cards by clicking on the `pre-loaded gift cards` link, which populates the page with a list of pre-loaded gift cards. In our running example, 4 pre-loaded gift cards are presented to the user, as depicted in Figure 1. Figure 2 shows the string representation of the DOM of that particular state. Each card, when clicked, triggers a JavaScript function called `addTotal` with an input parameter, as shown in Figure 3.

The application charges a $50 administration fee for all gift cards less than or equal to $200. For instance, if the user chooses a $200 gift card, the total will be $250 due to the administration fee. For gift cards greater than $200, there is no fee. In this example, there is a bug in the application: in Line 8 of Figure 2, instead of calling `addTotal` with 300 as input, the developer passes 350, which mistakenly includes the fee.

Figure 4 shows two typical test cases written using the Selenium framework. The first test case verifies the default functionality by adding a $50 gift card to the shopping cart, and the second verifies the total purchase price of a $200 pre-loaded gift card. As it can be seen, these two test cases provide a 100% JavaScript code coverage (Figure 3), although they fail to cover the erroneous $300 pre-loaded gift card. Thus, a traditional source code coverage report would not help developers in this regard.

Our insight is to look for novel, more relevant adequacy criteria that would help developers understand how their test cases cover the overall functionality of the web application under test. For example, the test cases of Figure 4, cover five elements directly (Lines 1, 2, 4, 7, 11 in Figure 2), but leave out four other elements, including the erroneous element (Lines 6, 8, and 9 in Figure 2). Simply sharing this information with the developers could guide them to identify untested elements of the application, increases the chance of revealing the bug.

```
1  @Test
2  public void defaultCard() {
3    driver.get(baseUrl + "/giftcard");
4    driver.findElement(By.cssSelector("input↩
         type="button"")).click();
5    assertEquals("$100", driver.findElement(By.id↩
         ("payable")).getText());
6  }

8  @Test
9  public void loadedCards() {
10   driver.get(baseUrl + "/giftcard");
11   driver.findElement(By.linkText("pre-loaded ↩
         gift cards")).click();
12   driver.findElement(By.id("second")).click();
13   assertEquals("$250", driver.findElement(By.id↩
         ("payable")).getText());
14 }
```

Figure 4: DOM-based test cases of the running example, using the Selenium framework.

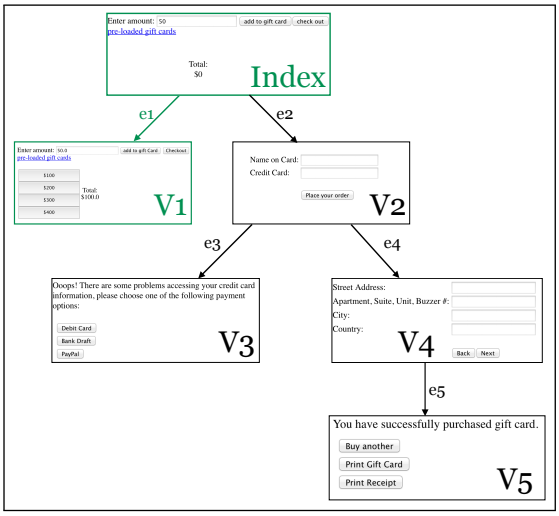

Figure 5: A state-flow graph of the running example. Nodes $Index$ and $V1$ and edge $e1$ are covered (green) by test cases of Figure 4.

3. ADEQUACY CRITERIA

In this section, we present a new set of test adequacy criteria that target the client-side state of web applications to highlight the potential inadequacy of their DOM-based test cases.

Definition 1 (DOM State). *A DOM State $\mathcal{DS}$ is a rooted, directed, labeled tree. It is denoted by a 5-tuple, $< D, Q, o, \Omega, \delta >$, where D is the set of vertices, Q is the set of directed edges, $o \in D$ is the root vertex, Ω is a finite set of labels and $\delta : D \rightarrow \Omega$ is a labelling function that assigns a label from Ω to each vertex in D.*

In this tree, the vertices are DOM elements and the edges are the hierarchical relations between the elements in the DOM. The DOM state is essentially an *abstracted* version of the DOM tree of a web application, displayed on the web browser at runtime. This abstraction is conducted through the labelling function δ, the implementation of which is discussed in Section 4.4.

Definition 2 (DOM-based test case). *A DOM-based test case t for a web application under test $\mathcal{W}$ is a tuple $< URL, l, \mathcal{A}, \mathcal{S}, \Lambda >$ where:*

1. *l represents the DOM state after $\mathcal{W}$ is fully loaded into the browser using the initial URL in t.*
2. *$\mathcal{S}$ is a set of states. Each $s \in \mathcal{S}$ represents a DOM state $\mathcal{DS}$ reachable by an action in t.*
3. *$\mathcal{A}$ is a sequence of event-based actions (such as clicks, mouseovers). Each $a \in \mathcal{A}$ represents an action connecting two DOM states if and only if state s_2 is reached by executing a in state s_1.*
4. *Λ is a set of DOM elements $\in S$. Each element $\in \Lambda$ represents an asserted element by t.*

Definition 3 (Test Suite). *A DOM-based test suite, T, is a set of DOM-based test cases, $T = \{t_1, t_2, t_3, ..., t_n\}$, where $t_i \in T$ is the i_{th} DOM-based test case in T.*

In particular, we propose to assess how a given test suite, T, covers the application's overall DOM states and elements. To this end, we propose two complimentary levels of test adequacy criteria, namely inter-state and intra-state criteria, described in the following two subsections, respectively.

3.1 Inter-state Adequacy Criteria

The first set of criteria we propose is concerned with the overall state space of the web application under test, represented by all the possible DOM states and transitions between them. The number of states in most industrial web applications is huge. Hence, it is intuitive to measure the portion of the total state space covered (or missed) by a test suite. Thus, the goal of the inter-state adequacy criteria is to make sure that all DOM states and transitions are covered, at least once.

We model the total state space of the web application under test as a state-flow graph (SFG) [23].

Definition 4 (State-flow Graph). *A state-flow graph SFG for a web application $\mathcal{W}$ is a labeled, directed graph, denoted by a 4 tuple $< r, \mathcal{V}, \mathcal{E}, \mathcal{L} >$ where:*

1. *r is the root node (called Index) representing the initial DOM state after $\mathcal{W}$ has been fully loaded into the browser.*
2. *$\mathcal{V}$ is a set of vertices representing the states. Each $v \in \mathcal{V}$ represents a DOM state $\mathcal{DS}$ of $\mathcal{W}$.*
3. *$\mathcal{E}$ is a set of (directed) edges between vertices. Each $(v_1, v_2) \in \mathcal{E}$ represents a clickable c connecting two states if and only if state v_2 is reached by executing c in state v_1.*
4. *$\mathcal{L}$ is a labelling function that assigns a label, from a set of event types and DOM element properties, to each edge.*
5. *SFG can have multi-edges and be cyclic.*

DOM State Coverage (DSC). The *DOM State Coverage* (DSC) adequacy criterion is satisfied by a test suite T, for a web application W, if each DOM state in W is visited by at least one test case in T, formally defined as:

Definition 5 (DSC). $T \triangleright DSC \Leftrightarrow \forall v \in \mathcal{V}, \exists t \in T : v \in \mathcal{S}_t$,

where $T \triangleright C$ is defined as the adequacy of the test suite, i.e., T satisfies criterion C, $\mathcal{S}_t$ is the set of all states covered by t, and $\mathcal{V}$ is the set of all states in SFG.

Lets assume that Figure 5 presents the state-flow graph of our running example, which consists of 6 states in total. Ideally, a test suite should cover all the 6 states. However, the test suite shown in Figure 4 only covers two of these states, namely, $Index$ and $S1$. This suggests that the test suite is inadequate and not capable of satisfying the DSC criterion.

The *DOM State Coverage* for a test suite T can be calculated as below:

$$DSC(T) = \frac{\left|\bigcup_{i=1}^{n} \mathcal{S}_{t_i}\right|}{|\mathcal{V}|}, \quad (1)$$

where n is the total number of test cases in T. For instance, the test suite of Figure 4 has a DSC coverage of 33% (2/6).

DOM Transition Coverage (DTC). The test suite execution can be mapped to a path in the state-flow graph of the web application. To transition from one state to another, a test case typically performs a series of actions by firing events (e.g., click) on a source element; if the action mutates the DOM, a target state is visited. The goal of this coverage criterion is to ensure that the test suite covers all actions from one state to another, at least once.

DEFINITION 6 (DTC). $T \triangleright DTC \Leftrightarrow \forall e(v_i, v_j) \in \mathcal{E},\ \exists t \in T : v_i, v_j \in \mathcal{S}_t$ *and* $(v_i, v_j) \in \mathcal{A}_t$

In other words, a test suite T satisfies the DTC criterion if and only if for each transition edge e in SFG, there is at least one test case $t \in T$ that covers e.

Similarly, the DOM transition coverage for a test suite T can be computed as follows:

$$DTC(T) = \frac{\left|\bigcup_{i=1}^{n} \mathcal{A}_{t_i}\right|}{|\mathcal{E}|} \quad (2)$$

Going back to the running example, the transition ($index$,-$s1$) in Figure 5 is covered, however, the other 4 transitions are missed, resulting in a DTC of 20% (1/5).

3.2 Intra-state Adequecy Criteria

The second set of criteria we propose examines each individual DOM state to highlight the elements covered by the test suite, within each state. The goal is to make sure that all DOM elements of the application are covered at least once. We distinguish between four different categories of DOM element coverage at this level. Our intra-state adequacy criteria focus on elements that are (1) explicitly covered, (2) actionable, i.e., capable of causing a state transition, (3) implicitly covered, and (4) checked through explicit oracles (i.e., in assertions).

Explicit Element Coverage (EEC). Each test case t is composed of one or more steps in which various DOM elements are directly accessed through element location strategies such as ID, XPath, Tag-name, text-value, attribute-value, or CSS selectors. For example, Line 4 in the `defaultCard` test case of Figure 4 uses the `cssSelector` method to retrieve the INPUT element and perform a click action.

DEFINITION 7 (EEC). *A DOM element, $ee \in v_i$, is **explicitly covered** if and only if there is at least one test case $t_i \in T$ such that ee is directly accessed from t through an element location strategy.*

Actionable Element Coverage (AEC). When a given JavaScript function f is set as an event-handler of a DOM element d of a DOM state v_i, d becomes a potential actionable element in v_i, capable of changing the state. Examples of event types typically attached to DOM elements are onclick, ondblclick, and onmouseover. Since these actionable elements are responsible for state transitions in a web application, we believe it is important to measure their coverage when assessing the quality of a test suite.

DEFINITION 8 (AEC). *A DOM element, $ae \in v_i$, is a **covered actionable element** if and only if ae has an event-handler h, is explicitly covered, and there is at least one test case $t_i \in T$ such that t_i triggers h.*

For instance, Line 12 in Figure 2 covers the actionable element with ID "second" of Figure 1, which has an event-listener for the `click` event type. Note that the test cases do not cover the erroneous actionable element in Line 8 of Figure 2.

Implicit Element Coverage (IEC). Interactions with actionable DOM elements in a test case fire their event-listeners, which can result in the execution of client-side JavaScript code. The JavaScript code executed can in turn access elements from the DOM. We call these DOM elements implicitly covered elements.

DEFINITION 9 (IEC). *A DOM element, $ie \in v_i$, is **implicitly covered** if and only if there is at least one test case $t_i \in T$ such that ie is indirectly accessed from t_i through the execution of the client-side source code of the web application.*

For instance, the DOM element with ID "xrt" is implicitly accessed through the JavaScript function `addTotal` (Line 6 in Figure 3) when the test case clicks on the button with ID "second" (Line 12 in Figure 4).

Checked Element Coverage (CEC). Test cases contain assertions on specific DOM elements to check whether the application behaves as expected. Finding out which portion of the program is actually checked through explicit assertions can be valuable for assessing the quality of a test suite [30].

DEFINITION 10 (CEC). *A DOM element, $ce \in v_i$, is a **checked element** if and only if it is checked in at least one oracle of a test case $t_i \in T$.*

Test cases typically check for the presence of DOM elements, their attributes, or values. For example, in the `defaultCard` test case of Figure 4, the textual value of the DOM element with ID "payable" is explicitly checked and is expected to be $100.

EEC, IEC, and CEC criteria can be computed as follows:

$$X_{v_i}(T) = \frac{\left|\bigcup_{j=1}^{n} Y_{t_j}\right|}{|D_{v_i}|}, \quad (3)$$

where X is an adequacy criteria in {EEC, IEC, CEC}, Y_{t_j} is set of Explicit, Implicit, or Checked elements of test case t_j , and D_{v_i} contains all the elements in DOM state v_i.

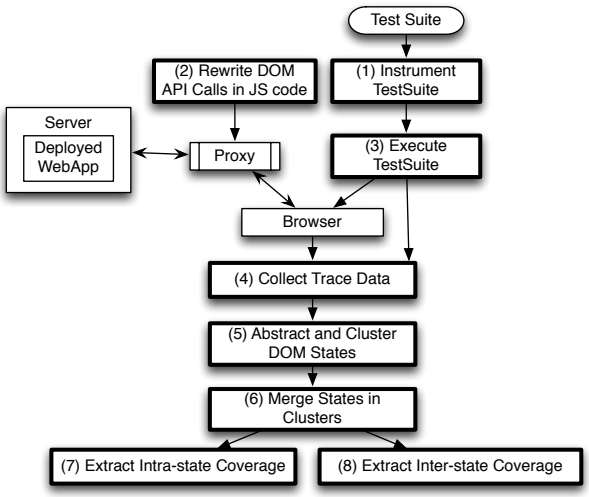

Figure 6: Overview of our DOM coverage approach.

AEC is calculated as follows:

$$AEC_{v_i}(T) = \frac{\sum_{j=1}^{n} \left|(EE_{t_j} \cup CE_{t_j}) \cap AE_{v_i}\right|}{|AE_{v_i}|}, \qquad (4)$$

Where EE_{t_j} and CE_{t_j} are sets of covered explicit and checked elements of test case t_j, respectively. AE is the set of all actionable elements in state v_i.

Subsumption Relations. A test adequacy criterion $\mathcal{A}$ subsumes test adequacy criterion $\mathcal{B}$ if every test suite satisfying $\mathcal{A}$ also satisfies $\mathcal{B}$ [27]. A test suite satisfying checked element coverage also covers explicitly covered elements, thus CEC subsumes EEC. Moreover, if a test suite satisfying explicit element coverage also satisfies actionable elements. Hence, EEC subsumes AEC.

4. MEASURING DOM COVERAGE

In this section, we present a technique to automatically measure the adequacy criteria proposed in Section 3. The main steps of our DOM coverage approach are depicted in Figure 6 and described in the following subsections.

4.1 Instrumenting the Test Suite

We instrument all the test cases of the application to yield information regarding particular (1) element access strategies (e.g., `findElement` in Selenium) and element identification methods (e.g., `By.id`), and (2) snapshots of intermediate DOM states visited after each action, in each test case. The instrumentation wraps around any method call that directly accesses (i.e., reads/writes) DOM elements, which can happen in the body of test methods, setup of test cases, helper methods, page objects, or test oracles. The instrumentation is non-intrusive, meaning that it alters neither the functionality of the application nor the behaviour of the test cases. The output of this step helps us to measure the DOM *states* and *explicit* elements covered by the test suite.

4.2 Rewriting DOM API Calls in JavaScript

Test suite instrumentation alone is not enough to extract coverage data for some of our adequacy criteria such as implicit or actionable elements. These elements are typically manipulated by client-side JavaScript code. Hence, we automatically inject a JavaScript file into the application before its initial execution (box 2 in Figure 6). This code is responsible for wrapping all DOM API method calls, in the JavaScript code of the application under test. These wrappers log exactly which DOM elements are accessed when the test cases execute. For *implicit* elements, we rewrite calls pertaining to DOM element accesses such as `getElementByID` and `getElementsByTagName`. For *actionable* elements, we rewrite DOM event APIs that add/remove event-handlers to DOM elements such as `addEventListener` and `removeEventListener`. We annotate these elements so that we know exactly which elements are actionable and which of these are covered by the test suite.

Algorithm 1: Clustering DOM States.

```
input : New State: NS, Covered Elements: elements,
        Cluster of states: Clusters, Similarity threshold: T
output: Updated clusters
1  begin
2      ME ← 0
3      foreach Cluster ∈ Clusters do
4          foreach S ∈ Cluster do
5              ME ← contains(S, elements)?1 : 0
6              dissimilarityScore ← distance(NS, S)/2^ME
7              if dissimilarityScore ≤ T then
8                  addToCluster(NS, Cluster)
9              else
10                 NC ← newCluster()
11                 addToCluster(NS, NC)
12                 Clusters.add(NC)
13     return Clusters
```

4.3 Collecting Trace Data

Next, we execute the instrumented test suite. For each test case, we collect data pertaining to the fully qualified name of each test case, and the DOM states visited by each test case. For each visited DOM state, we annotate the *explicit*, *actionable*, *implicit*, and *checked* elements on the DOM tree and save a snapshot. For each of these four element types, we also record the location strategy (e.g., XPath, ID, cssSelector) and values used in the corresponding test case to access them.

4.4 DOM State Abstraction and Clustering

An event-based action in a test case can change the state of the web application and thus the DOM of a web application can undergo various permutations to reflect these changes. However, some of such intermediate permutations are small in nature, which do not represent a proper state change, e.g., a row is added to a table. Hence, a transformation on the raw DOM tree is required to abstract away these subtle differences (see labelling function δ in Definition 1).

DOM Meta-Models. To capture the essential structural pattern of a DOM state, we automatically reverse engineer a meta-model of each DOM state in the form of an XML Schema. This meta-model abstracts away subtle differences between DOM structures, e.g., same table but with different number of rows. To find semantic structural differences we compute the *edit distance* between the syntax trees of the schemas. We define a threshold beyond which such differences become significant to represent a new state.

Algorithm 2: Merging a cluster of DOM states.

```
   input  : Cluster of DOM states Φ
   output: Merged DOM
1  begin
2      mergedDOM ← NULL
3      i, MaxIndex ← 0
4      MaxSize ← Φ[0].size
5      while i < Φ.size do
6          VisSize ← Φ[i].getAllElements.size()
7          if VisSize > MaxSize then
8              MaxSize ← VisSize
9              MaxIndex ← i
10     mergedDOM ← Φ[MaxIndex]
11     Φ.remove(MaxIndex)
12     foreach dom ∈ Φ do
13         coveredElements ← getCoveredElements(dom)
14         annotateElements(coveredElements, mergedDOM)
15     return mergedDOM
```

Matching Covered Elements. In addition to the meta-model, we track the presence of matching elements (i.e., explicit, actionable, implicit, checked) to identify similar DOM states in each test case. The rationale here is that two DOM trees cannot be clustered if they contain different sets of elements accessed by a test case.

Our clustering algorithm, presented in Algorithm 1, puts a DOM state in the same cluster if the dissimilarity with respect to another DOM state in the cluster is less than a certain threshold. The dissimilarity score is computed through Equation 5:

$$Score = \frac{\Delta(MM_{dom_1}, MM_{dom_2})}{2^{ME}}, \tag{5}$$

where $Score$ is the dissimilarity score between $dom1$ and $dom2$, MM computes the meta-models, and the matching elements ME is calculated using Equation 6.

$$ME = \begin{cases} 1 & elements_{dom1} \subseteq D_{dom2} \\ 0 & otherwise \end{cases} \tag{6}$$

where $elements_{dom1}$ represents all the explicit, actionable, implicit, checked elements of *dom1*, and D_{dom2} is the set of all elements in *dom2*.

4.5 Merging States in Clusters

In this phase, we merge the states in each cluster into one single state. This reduction makes it easier for the users to distinguish covered elements.

Algorithm 2 presents our merging algorithm. We start with the DOM state with the largest tree among all the states in the cluster (Lines 3-9). We select the largest DOM state to show as many annotated covered elements as possible on a single state. We highlight the matching elements that are covered in other DOM states of the cluster. Method *getCoveredElements*() in Line 13 extracts covered elements including explicit, actionable, implicit, and checked elements. Then, *annotateElements*() in Line 14 annotates all the covered elements inside the merged state. *annotateElements* extracts each element's location strategy and access identification type, then it finds the elements in the *mergedDOM* to tag the element with corresponding annotations. We perform these actions until all states in each cluster are merged.

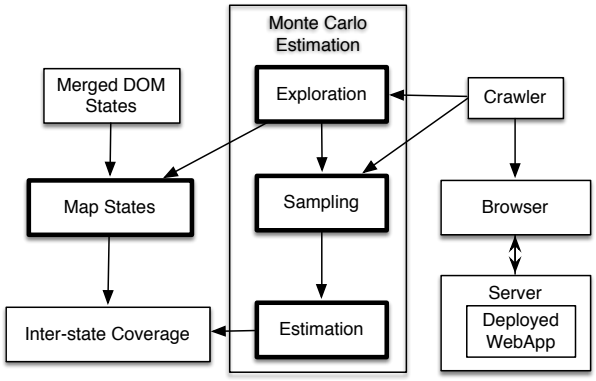

Figure 7: Inter-state Coverage Extraction.

4.6 Extracting Inter-state Coverage

The sum of merged states produced in the previous step represents the total number of DOM states covered by the test suite. However, to compute the inter-state coverage according to Equations 1–2, we need to know the total number of DOM states and transitions for the application under test.

Knowing the exact state space size of many large dynamic web applications is an undecidable problem. Hence, we extrapolate an estimate of the state space through a Monte Carlo estimation method [31], which combines (1) dynamic exploration, (2) sampling, and (3) estimation. The method assumes there are limited resources, such as time, available. Figure 7 depicts our inter-state coverage extraction procedure.

Dynamic Exploration. We use an event-based dynamic crawling technique [23], which given a URL of the web application under test, and a dedicated amount of time, exhaustively explores the application and produces a state-flow graph (see Definition 4) as output. Total number of DOM explored states in this step is $\mathcal{CS}$. We use the meta-model DOM state abstraction mechanism in the crawling step, as described in Section 4.4.

Sampling. The resulting state-flow graph could be partial for large applications, given the time constraints. In the second step, the method randomly samples states from the state-flow graph that still contain *unexecuted* actions (e.g., clickables). If a sampled unexecuted action, when executed, results in a *productive state*, i.e., a state that does not exist in the state-flow graph, the sampling step continues in a depth-first manner, until no other unexecuted actions are found or a depth limit (3) is reached. A sampled transition that results in a productive state is called a *productive transition*. The random sampling continues until the dedicated sampling time quota is reached.

Estimation. Once we know the number of sampled states, sampled productive transitions, and unexplored transitions, the total number of unvisited states can be estimated using the following formula [31]:

$$\frac{Sampled\ States}{Sampled\ Prod.\ Trans.} \approx \frac{\mathcal{M}}{Unexplored\ Trans.}, \tag{7}$$

where $\mathcal{M}$ is the number of unvisited DOM states that is estimated. We run this estimation method multiple times and

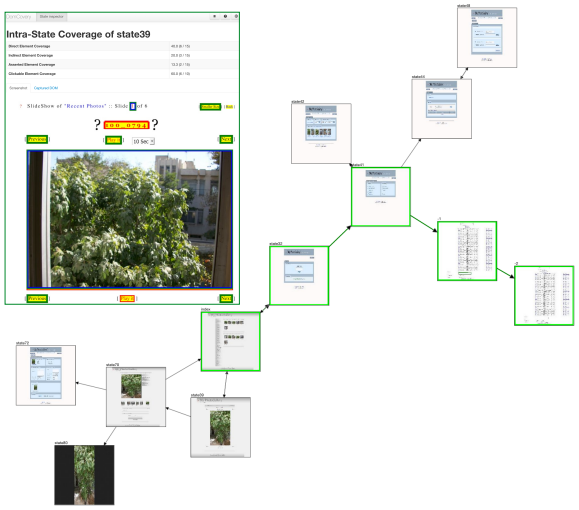

Figure 8: Inter-state coverage report for Phormer: the SFG highlighted with covered states (green border) and transitions (green arrow). Upper left corner shows one of the DOM states with intra-state coverage details.

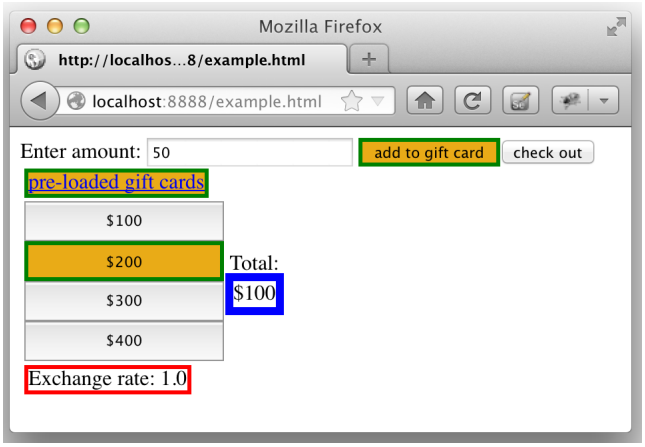

Figure 9: Intra-state coverage report: *Explicit Elements* in green, *Implicit Elements* in red, *Checked Elements* in blue, and *Actionable Elements* in yellow.

use average numbers obtained for $\mathcal{M}$ as an approximation of unvisited states.

We calculate DOM state coverage by revising Equation 1 as follows:

$$DSC(T) = \frac{\left|\bigcup_{i=1}^{n} \mathcal{MS}_{t_i}\right|}{\left|\bigcup_{i=1}^{n} \mathcal{MS}_{t_i} \cup \mathcal{CS}\right| + \mathcal{M}} \quad (8)$$

where $\mathcal{MS}_{t_i}$ is the merged states covered by test case $t_i \in T$ and $\mathcal{CS}$ is the set of states crawled during the dynamic exploration. The denominator is a summation of (1) the conjunction of covered merged states and crawled states (*Map States* in Figure 7) and, (2) the estimated unvisited states. Note that DOM transition coverage is computed in a similar way.

4.7 Extracting Intra-state Coverage

At this point, each merged state is annotated with covered elements. These annotations are in the form of styling attributes so that they can easily be highlighted in the final coverage report. Figure 9 shows the coverage report generated for the test cases (Figure 4) of our running example. The report highlights the covered elements in different colours. For each merged state, since we know the total number of DOM elements and actionable elements, we calculate the coverage of the four intra-state element types according to Equations 3–4.

4.8 Tool Implementation: DOMCovery

We have implemented our DOM coverage approach in an automated tool called DomCovery, which is written in Java and is freely available [4]. The current implementation of DomCovery supports Selenium 2.0 test cases written in Java. To parse and instrument test cases, we use JavaParser [6]. To collect implicit and actionable elements, we inject the rewriting JavaScript code through a web proxy, into the HTML documents originating from the server. For the dynamic exploration and sampling steps, we build upon Crawljax [23, 24]. DomCovery accepts as input the test suite and URL of a given web application and automatically performs all the operations and computes the coverage. Moreover, DomCovery generates a visual report, which specifies all the DOM states and highlights the covered DOM states and transitions in the resulting state-flow graph. For each DOM state, DomCovery shows the DOM structure and highlights the explicit, implicit, checked, and actionable elements. A snapshot of the generated report is depicted in Figure 8. The overview page of the report includes all the coverage percentages as well.

5. EMPIRICAL EVALUATION

To evaluate the efficacy of our approach, we conducted two studies pertaining to (1) a correlation analysis between our DOM-based coverage and traditional code coverage, (2) a controlled experiment for assessing the effectiveness of DomCovery in helping testers identify (un)covered portions of a web application under test. Our research questions:

RQ1 Is there a correlation between code-based and DOM-based test adequacy criteria?

RQ2 Is DomCovery effective in helping testers identify covered and untested portions of a web application under test?

5.1 Experimental Objects

We selected two open source web applications of moderate size. Phormer [9] is an online photo gallery written in JavaScript, CSS, XHTML, and PHP. It provides features such as uploading, commending, rating, and displaying slideshows for photos. Phormer has around 6,000 lines of JavaScript, PHP and CSS code in total. It was rated 5.0 star on SourceForge and had over 39,000 downloads when this evaluation was conducted. The second application in our study is Claroline [2], an open source collaborative and online learning platform. It has been downloaded more than 330,000 times. Claroline consists of 3.8 KLOC of JavaScript,

Table 1: Kandall τ correlation coefficients for code-based and DOM-based coverage criteria.

Code \ DOM	DSC	DTC	EEC	IEC	CEC	AEC
Function	0.18	0.17	-0.07	0.43	-0.16	0.00
Statement	0.19	0.19	-0.15	0.47	-0.26	-0.09
Branch	0.06	0.03	-0.06	0.43	-0.15	-0.05

3.2 KLOC of CSS, 6.2 KLOC of HTML, and 293.7 KLOC of PHP.

Since these applications come with no test suites,[1] we asked two students – with expertise in web development – to write test cases for these two objects, individually. 20 test cases were written for each experimental object using Selenium (WebDriver) [10].

Based on a preliminary study, DOM comparison threshold values (see Section 4.4) of 34 and 17 were determined and used for Phormer and Caroline, respectively.

5.2 Correlation Analysis

To address **RQ1**, we measured correlations between client-side JavaScript code coverage metrics – i.e., function, statement, branch coverage – and DOM coverage criteria – i.e., DSC (state), DTC (transition), EEC (explicit element), AEC (actionable element), IEC (implicit element), and CEC (checked element) coverage.

We use the Kendall rank coefficient (τ) [11], since it assumes neither that the data is distributed normally nor that the variables are related linearly. Kendall coefficient values are in $[-1, +1]$, where -1 shows a strong negative correlation, 0 indicates no correlation, and $+1$ shows a strong positive correlation.

For each of the experimental objects, we executed each test case in isolation and measured (1) code coverage using JSCover [7]; we configured JSCover to measure all application specific JavaScript code excluding libraries, and (2) DOM coverage using DomCovery.

Table 1 shows the correlation coefficients between code and DOM-based coverage criteria. As the results show, there is no strong correlation between code-based metrics and DOM-based metrics. For instance, the correlation coefficient of statement coverage and DOM state coverage (DSC) is 0.19, which is considered as no correlation. The same is true for all the other data points in the table except for IEC. IEC shows a "moderate" correlation with function (0.43), statement (0.47), and branch (0.43) coverage. This is, however, not surprising. By the definition of implicit element coverage (see Definition 9), the more JavaScript code is covered, the more implicit DOM elements can be covered by a test case.

5.3 Controlled Experiment

To address **RQ2**, we conducted a controlled experiment [33]. In this experiment, we compared how effective DomCovery is when compared to current web application code coverage and development tools.[2]

Experimental Object and Subjects. We used Phormer as the experimental object in this study.

[1]This is the case for many open source web applications.

[2]Documents including tasks and questionaries used for our experiment can be viewed at [4].

Table 2: Experimental Tasks

Id	Task Description
T1	Locating the explicit elements covered by a test suite
T2	Locating the implicit elements accessed via a test suite
T3	Locating the checked elements asserted in a test suite
T4	Locating the actionable elements covered by a test suite
T5	Identifying the features under tests given a test suite
T6	Identifying the untested features given a test suite

As subjects, a group of 10 participants were recruited from the graduate students (5 Master and 5 PhD) at UBC. The participants had an average of 3.6 years experience in web development and 1.7 years in software testing. We chose to use a "between-subject" design; i.e., the subject is either part of the control (Ctrl) or experimental (Exp) group. The experimental group used only DomCovery while the control group was allowed to use existing tools such as JSCover for JavaScript code coverage and web development/debugging tools such as Firebug [5]. The assignment of participants to groups was done randomly. None of the participants had any previous experience with DomCovery and all of them volunteered for the study.

Independent and Dependent Variables. The tool used for performing the tasks is our independent variable and has two levels: DomCovery represents one level, and *other tools* used in the experiment represent the other level (i.e., JSCover: code coverage tool, Firebug: development/debugging plugin). Task completion time (i.e., measure of efficiency) and correctness (i.e., measure of effectiveness) are our dependent variables.

Task Design. The subjects were required to perform a set of tasks pertaining to test case quality analysis.

We designed six tasks in total, presented in Table 2. The first four tasks were specifically designed to evaluate correctness and time savings of our tool in finding explicit, actionable, implicit and checked elements. We printed a specific state of the Phormer application (as seen in a browser) and asked the participants to highlight the explicit, implicit, checked, and actionable elements. We counted the number of correct identifications of the elements as a correctness measure. Tasks 5 and 6 were designed to evaluate the usefulness of DomCovery in helping users to identify tested and untested features of the application. We extracted a list of features from the documentation of Phormer. For example, one feature is *Photo Rating*: users can rate the photos in a scale of 1 to 5. For task 5, we gave the participants a list of features on a sheet together with a test suite and asked them to checkmark the tested features of the application. We then counted the number of correct identifications of tested features. For example, in Task 5, there were eight features in total out of which only three were tested by the given test suite. Task 6 was designed similarly, but instead we asked the participants to identify the untested features of the application.

Experimental Procedure. After obtaining the consent, the participants were asked to fill a pre-questionnaire form, specifying their level of expertise in areas related to web development and testing. Next, the participants in the experimental group were trained to use DomCovery for performing the basic operations needed in the study. They were debriefed on features and capabilities of DomCovery; they were given the opportunity to use the tool until they felt comfortable using it. Finally, they were given a user manual

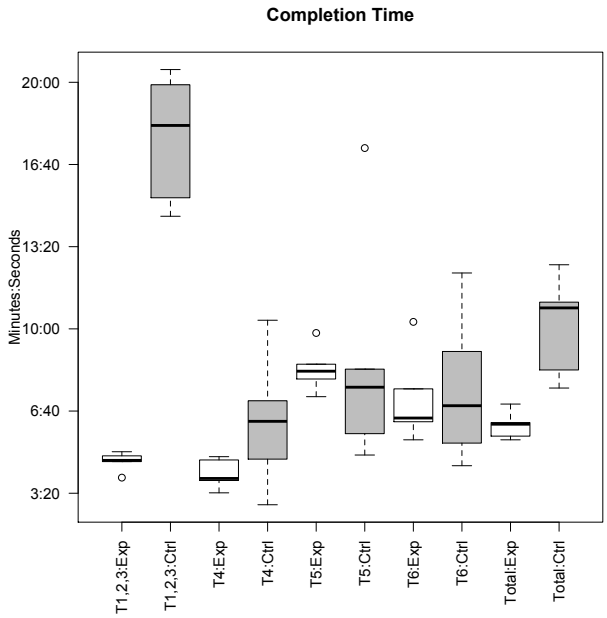

Figure 10: Box plots of task completion duration for experimental (Exp) and control (Ctrl) groups.

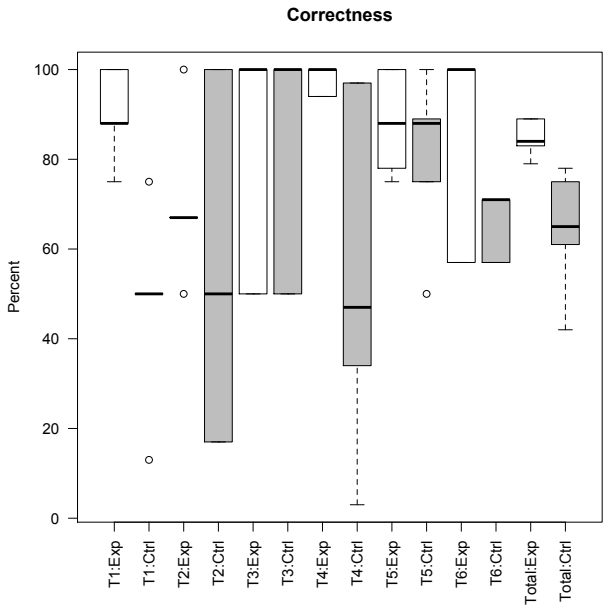

Figure 11: Box plots of task completion correctness for experimental (Exp) and control (Ctrl) groups.

of DomCovery, which could be used during the experiment if needed. The participants in the control group were given a tutorial on using JSCover and Firebug, although most were already familiar with Firebug.

After the training sessions, the participants were asked to perform the six tasks. We collected and measured total task completion duration for all tasks. For assessing the correctness, we had prepared an answer-key for each task separately, prior to the study.

Finally, participants were asked to fill out a short post-questionnaire form to gather qualitative feedback.

Results. Figure 10 shows box plots of task completion time. We measured the amount of time (minutes) spent on each task by the participants, and compared the task completion duration for experimental and control groups using a t-test. There was a statistically significant difference (p-value=0.05) in the task completion time for DomCovery (Mean=4:23, SD=2:04) and *other tools* (Mean=11:59, SD=5:37).To investigate whether certain categories of tasks (Table 2) benefit more from using DomCovery, we then tested each task separately. The results showed improvements in time for all tasks. The improvements were statistically significant for tasks 1–4, and showed a 174% reduction in time on average, with DomCovery. For tasks 5 and 6 the completion time were not statistically significant. Our results show that on average, participants using DomCovery require 66% less time than the control group, for performing the same tasks.

Figure 11 shows box plots for the correctness measurements of the individual tasks and in total. We analyzed the scores of correctness (percentage) of participants' answers using a t-test. The results were in favour of DomCovery and were statistically significant (p-value=0.05) in tasks 1, 2, and 4: DomCovery (Mean=84%, SD=17%) and *other tools* (Mean=65%, SD=34%). The results show that, on average, participants using DomCovery achieved 22% higher correctness than those in the control group.

6. DISCUSSION

6.1 Identifying Covered Elements

Identifying explicit elements covered was relatively easier for the participants in the control group compared with finding implicit elements. The reason is that explicit elements can be identified by inspecting the code of the test cases. However, many industrial test cases use patterns such as *Page Objects* [8] to hide element location information. In these scenarios, locating event explicitly covered elements becomes difficult for testers. As our results show, having an automated tool can significantly reduce the efforts involved in finding these elements.

Locating implicit elements covered was the most challenging task for the control group. The participants spent most of their time trying to find these elements. Most of the participants used Firebug to trace the JavaScript code. With dynamic characteristics of client-side code, even one of the most experienced participants made a mistake in spotting implicit elements in the control group. For example, clicking a DOM element triggered a JavaScript function, which in turn read values of a specific DOM element when a certain condition held. The participant assumed that the condition always held and identified that element as implicitly covered by the test case, which was wrong. Other intricate characteristics of modern web applications such as event capturing and bubbling further complicate the understanding [14] and identification of implicit elements. DomCovery makes this task easy by dynamically tracking JavaScript-DOM interactions and providing a visualized report of all the implicitly covered elements to the user.

6.2 Identifying (Un)tested Features

To perform tasks 5 and 6, the participants in the experimental group followed highlighted (meaning covered) states in the state-flow graph visualized in the coverage report. They mapped these states to the given list of features. Although a DOM state does not directly map to a feature, the participants could identify the features under test by investigating multiple covered states of the web application. Although the results were not statistically significant, participants using DomCovery performed the tasks more accurately and faster. Particularly for task 6, participants in the control group struggled to find the untested features.

6.3 Post-questionnaire Feedback

All participants using DomCovery found the visual DOM coverage report helpful, especially how it gave them a broad overview of the covered states of the web application. Some mentioned that they liked the implicit element coverage the most, since those are hard to find manually. Some participants mentioned that although they found the graph visualization useful, if the application is very large, the graph could become difficult to navigate. Currently DomCovery supports zooming in and out so that users can concentrate their focus on the states that matter most to them. Others suggested to include a navigation flow between covered states when interacting with the state-flow graph so that they do not have to go back and forth in order to go to another state.

6.4 Threats to Validity

A threat to the validity of our results could be that the participants are not representative of industrial web developers. We tried to mitigate this threat by recruiting graduate students who possessed moderate expertise with web development. Our study could, however, benefit from an increase in the number and expertise of the participants. Another threat could pertain to the fact that the test cases were written by students. Unfortunately, very few open source web applications are available that have (working) test cases. To mitigate this threat, again we made sure the undergrad students had expertise in the web development and testing area. Concerning reproducibility of our results, DomCovery, the experimental objects, and all our experimental results are available for download making the study replicable.

7. RELATED WORK

The most common approach to assess the quality of a test suite in the literature is through code coverage [34]. However, code coverage has limitations when it comes to test suite assessment [32, 26, 20, 18, 19, 15]. Most empirical studies conclude that code coverage metrics are too generic and not adequate for assessing test quality. Researchers have proposed various domain-specific test adequacy criteria to mitigate this problem, which span from security-aware, GUI event based, to database-level criteria.

Koster and Kao [21] propose a state coverage method for assessing the adequacy of unit tests. Although they consider the state of the application, their approach works on unit test level and does not apply to the web applications.

Memon et al. [22, 16] propose event-based interaction coverage criteria for GUI test cases of desktop applications. They exploit the hierarchical structure of the GUI to identify important events to be covered. Their approach, on covering event sequences is similar to our state transition criteria. We propose a whole set of new adequacy criteria that are geared towards modern web applications.

Sampath et al. [29] introduce page-level coverage criteria such as URL coverage. They report the coverage of a test suite as static pages visited during test execution for a given web application. However with increasing usage of JavaScript and dynamic DOM, the applicability of this approach is limited to traditional multi-page web applications. Our approach focuses on modern dynamic web applications that go through DOM mutations at runtime.

Dao et al. [17] discuss a set of criteria for security testing, including wrapper coverage, vulnerability-aware sink coverage, and vulnerability-aware wrapper coverage. Their approach is used for evaluating the quality of a test suite in revealing security vulnerabilities. Sakamoto et al. [28] use template variable coverage, accompanied with a framework, to generate test cases that improve that coverage. However, this criteria works only on web applications that are generated using template engines.

Alalfi et al. [12, 13] present a coverage criterion for dynamic web applications based on page access, use of server variables, and interactions with the backend database. The goal of their coverage criteria is different from our work. Namely the are interested in data related coverage, whereas we focus on client-side functionality coverage.

Finally, Zou et al. [35] propose a hybrid coverage criteria, which combines HTML coverage with statement coverage of the code. However, their approach and evaluation seems preliminary; they support neither inter-state coverage estimation nor intra-sate element coverage extraction as our work does.

8. CONCLUSION

In this paper, we presented six DOM-based coverage criteria for assessing the quality of web application test suites. Our criteria fall into two main categories of inter and intra-state coverage. Inter-state criteria include DOM state and transition coverage. Intra-state criteria pertain to explicit, implicit, checked, and actionable DOM element coverage. We presented the implementation of our approach in a tool called DomCovery. DomCovery generates a visual coverage report along with coverage percentages. The results of our empirical evaluation show that DOM-based coverage criteria are not correlated with traditional code-based coverage metrics. Our controlled experiment with ten participants shows that DomCovery can improve time (66%) and correctness (22%) of tasks related to test suite quality analysis of web applications.

For future work, we plan to include more fine-grained DOM-based coverage criteria such as DOM attribute coverage, build a state navigation flow for DomCovery, and perform an industrial experiment to further assess the effectiveness of our approach.

9. ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants program and Swiss National Science Foundation (PBTIP2145663). We are grateful to all participants of the controlled experiment for their time and commitment.

10. REFERENCES

[1] CasperJS. http://casperjs.org.

[2] Claroline. http://www.claroline.net/.

[3] Document Object Model (DOM). http://www.w3.org/DOM/.

[4] DOMCovery. http://salt.ece.ubc.ca/software/domcovery/.

[5] Firebug. https://getfirebug.com/.

[6] JavaParser. https://code.google.com/p/javaparser/.

[7] Jscover. http://tntim96.github.io/JSCover/.

[8] Page Objects. https://code.google.com/p/selenium/wiki/PageObjects.

[9] Phormer photogallery. http://sourceforge.net/projects/rephormer/.

[10] Selenium HQ. http://seleniumhq.org/.

[11] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[12] M. Alalfi, J. Cordy, and T. Dean. Automating coverage metrics for dynamic web applications. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 51–60. IEEE, 2010.

[13] M. Alalfi, J. R. Cordy, and T. R. Dean. DWASTIC: Automating coverage metrics for dynamic web applications. In *Proceedings of the ACM Symposium on Applied Computing*, 2009.

[14] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[15] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 148–157. IEEE, 1999.

[16] R. C. Bryce and A. M. Memon. Test suite prioritization by interaction coverage. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, pages 1–7. ACM, 2007.

[17] T. Dao and E. Shibayama. Coverage criteria for automatic security testing of web applications. In *Information Systems Security*, volume 6503 of *Lecture Notes in Computer Science*, pages 111–124, 2011.

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 302–313. ACM, 2013.

[19] J. R. Horgan, S. London, and M. R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994.

[20] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2014.

[21] K. Koster and D. Kao. State coverage: a structural test adequacy criterion for behavior checking. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, pages 541–544. ACM, 2007.

[22] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 256–267. ACM, 2001.

[23] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[24] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Trans. Softw. Eng.*, 38(1):35–53, 2012.

[25] J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Communication of ACM*, 6(2):58–63, 1963.

[26] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–68. ACM, 2009.

[27] M. Pezze and M. Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[28] K. Sakamoto, K. Tomohiro, D. Hamura, H. Washizaki, and Y. Fukazawa. POGen: A test code generator based on template variable coverage in gray-box integration testing for web applications. In *Fundamental Approaches to Software Engineering*, volume 7793, pages 343–358, 2013.

[29] S. Sampath, E. Gibson, S. Sprenkle, and L. Pollock. Coverage criteria for testing web applications. *Computer and Information Sciences, University of Delaware, Tech. Rep*, pages 2005–017, 2005.

[30] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 90–99, 2011.

[31] A. Taleghani and J. M. Atlee. State-space coverage estimation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 459–467. IEEE Computer Society, 2009.

[32] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 194–212, 2012.

[33] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in software engineering*. Springer, 2012.

[34] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Survey*, 29(4):366–427, 1997.

[35] Y. Zou, C. Fang, Z. Chen, X. Zhang, and Z. Zhao. A hybrid coverage criterion for dynamic web testing. In *Proceedings of International Conference on Software Engineering and Knowledge Engineering*, 2013.