Feedback-Directed Exploration of Web Applications to Derive Test Models

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

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

***Abstract*—Dynamic exploration techniques play a significant role in automated web application testing and analysis. However, a general web application crawler that exhaustively explores the states can become mired in limited specific regions of the web application, yielding poor functionality coverage. In this paper, we propose a feedback-directed web application exploration technique to derive test models. While exploring, our approach dynamically measures and applies a combination of code coverage impact, navigational diversity, and structural diversity, to decide a-priori (1) which state should be expanded, and (2) which event should be exercised next to maximize the overall coverage, while minimizing the size of the test model. Our approach is implemented in a tool called FEEDEX. We have empirically evaluated the efficacy of FEEDEX using six web applications. The results show that our technique is successful in yielding higher coverage while reducing the size of the test model, compared to classical exhaustive techniques such as depth-first, breadth-first, and random exploration.**

***Index Terms*—model generation, web app, coverage, testing, diversity**

I. INTRODUCTION

Modern web applications make extensive use of JavaScript to dynamically mutate the Document Object Model (DOM) in order to provide responsive interactions within the browser. Due to this highly dynamic nature of such applications, dynamic analysis and exploration (also know as crawling) play a significant role [10] in many automated web application testing techniques [2], [4], [7], [16], [18], [20], [22], [23], [34]. Such testing techniques depend on data and models generated dynamically through web application crawling.

Most web application exploration techniques used for testing apply an exhaustive search in order to achieve a "complete" coverage of the application state-space and functionality. An assumption often made is that the state-space of the application is completely coverable in a reasonable amount of time. In reality, however, most industrial web applications have a huge dynamic state-space and exhaustive crawling – e.g., through breadth-first search (BFS), depth-first search (DFS), or random search – can cause the state explosion problem. In addition, a generic crawler that exhaustively explores the states can become mired in irrelevant regions of the web application [25], producing large test models that yield poor functionality coverage.

Because exploring the whole state-space can be infeasible (state explosion) and undesirable (time constrains), the challenge we are targeting in this paper is to automatically derive an incomplete test model but with adequate functionality coverage, in a timely manner.

To that end, we propose a novel feedback-directed exploration technique, called FEEDEX, which is focused on efficiently covering a web application's functionality to generate test models. We propose four metrics to capture different aspects of a test model, namely *code coverage impact*, *navigational diversity*, *page structural diversity*, and *test model size*. Using a combination of these four metrics, our approach dynamically monitors the exploration and its history. It uses the feedback obtained to decide a-priori (1) which states should be expanded, and (2) which events should be exercised next, so that a subset of the total state-space is effectively captured for adequate test case generation.

The main contributions of our work include:

- We present a feedback-directed exploration technique that selectively covers a subset of the state-space of a given web application to generate an adequate test model;
- We propose the notions of coverage impact and diversity – i.e., navigational and page structural diversity – to capture different aspects of derived test models;
- We describe an event execution method, which prioritizes events based on their historical productivity in producing relevant states;
- We implement our approach in a tool called FEEDEX, which is freely available;
- We empirically evaluate the efficacy of our approach on six Web 2.0 applications. The results show that our method yields much better code coverage (10% at the minimum), and diversity (23% at the minimum), compared to traditional exhaustive methods. In addition, our approach is able to reduce the size of the derived test model and test suite by at least 38% and 42%, respectively.

II. BACKGROUND AND MOTIVATION

Web 2.0 Applications. The advent of recent Web and browser technologies has led to the proliferation of modern – also known as Web 2.0 – web applications, with enhanced user interaction and more efficient client-side execution. Building on web technologies such as AJAX, Web 2.0 applications execute a significant amount of JavaScript code in the browser.

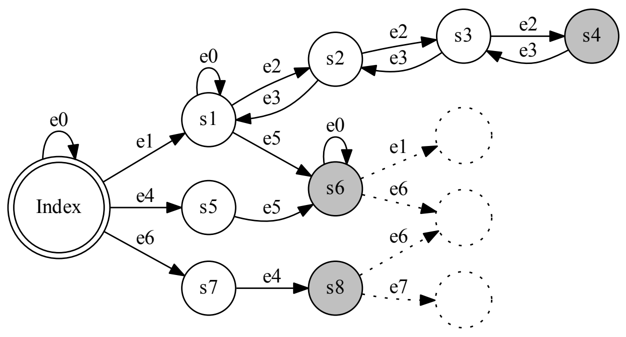


Fig. 1. State-flow graph of the running example.

A large portion of this code is dedicated to interact with the Document Object Model (DOM) to update the user interface at runtime. In turn, these incremental updates lead to dynamically generated execution states within the browser.

Automated Test Model Generation. Model-based testing uses models of program behaviour to generate test cases. These *test models* are either specified manually or derived automatically.

Tool support for exploring (or "crawling") dynamic states of web applications enables automated test model derivation. Unlike traditional static hyper-linked web pages, automatically exploring Web 2.0 applications is challenging because many of the user interface state changes are (1) event-driven, (2) caused by the execution of client-side application code , and (3) represented by dynamically mutating the internal DOM tree in the browser. In the recent past, web crawlers have been proposed that can explore event-driven DOM mutations in Web 2.0 applications. For instance, CRAWLJAX [19] uses dynamic analysis to exercise and crawl client-side states of Web 2.0 applications. It incrementally reverse engineers a model, called *State-Flow Graph* (SFG), which captures dynamic DOM states and event-driven transitions connecting them. This SFG model is formally defined as follows:

Definition 1: A **state-flow graph** SFG for a Web 2.0 application $\mathbf{A}$, is a labeled directed graph, denoted by a 4 tuple $< \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E}, \mathcal{L} >$ where:

1) $\boldsymbol{r}$ is the root node (called Index) representing the initial state after $\mathbf{A}$ has been fully loaded into the browser.
2) $\boldsymbol{V}$ is a set of vertices representing the states. Each $\boldsymbol{v} \in \boldsymbol{V}$ represents a runtime DOM state in $\mathbf{A}$.
3) $\boldsymbol{E}$ is a set of (directed) edges between vertices. Each $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{E}$ represents a clickable $\boldsymbol{c}$ connecting two states if and only if state $\boldsymbol{v}_2$ is reached by executing $\boldsymbol{c}$ in state $\boldsymbol{v}_1$.
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. □

To infer this SFG, events (e.g., clicks) are automatically generated on *candidate clickables*, i.e., user interface elements of the web application that can potentially change the state of the application. An example is a `DIV` element, dynamically bound to an event-listener, which when clicked calls a JavaScript function, which in turn mutates the DOM inside the browser (see Figures 3 and 4). Only when the event generated results in a state change, the candidate clickable is seen as a real *clickable* and the new state and the clickable are added to the SFG.

An example of such a SFG is shown in Figure 1. Such an automatically inferred test model can be utilized in test case generation [20], by adopting different graph coverage methods (e.g., all states/edges/paths/transitions coverage).

Motivation. Given that (1) most industrial web applications have a huge dynamic state-space, which can cause the state explosion problem, (2) dynamic exploration is time-consuming, and (3) in any realistic software development environment, the amount of time dedicated to testing is limited, opting for the exploration of a partial subset of the total state-space of a given web application seems like a pragmatic feasible solution.

Given a specific amount of time, there are, however, different ways of exploring this partial subset. For example, assume that the SFG in Figure 1, including the dashed nodes and edges, represents the complete state model of a web application. Also assume that our allowed crawling time is limited to 4 event executions and our crawler applies a depth-first search. If the explored sequence of exercised clickables is $e1$, $e2$, $e2$, $e2$, we retrieve states $s1$, $s2$, $s3$, and $s4$, all of which belong to same crawling path. It could also be the case that the crawler has a predetermined order for event execution. Consider a crawler in which $e3$ always executes before $e2$. The sequence of the resulting clicks would then become $e1$, $e2$, $e3$, $e2$ which results in discovering only 3 states, namely $s1$, $s2$, and $s3$. Another example, depicted in Figures 3–4, includes exploring states behind *next* and *previous* links. This is a typical scenario in which exhaustive crawlers can become mired in specific parts of the web application, yielding poor functionality coverage.

Because a complete exploration of the whole application can be infeasible (state explosion), undesirable (time constrains), and inefficient (large test model derived that yields low coverage), the challenge we are targeting in this paper is to derive an incomplete test model that can adequately provide functionality coverage.

III. APPROACH

The goal of our work is to automatically derive a test model that captures different aspects of the given web application's client-side functionality. In the next subsections, we present these desired aspects of a test model, followed by a salient description of our *feedback-directed exploration* technique, called FEEDEX.

A. Deriving Test Models

A web crawler is generally evaluated on its ability to retrieve relevant and desirable content [1], [25], [31]. In this work, we propose metrics that target relevance and desirability in the context of web application testing. We believe that a test model derived automatically from a web application should

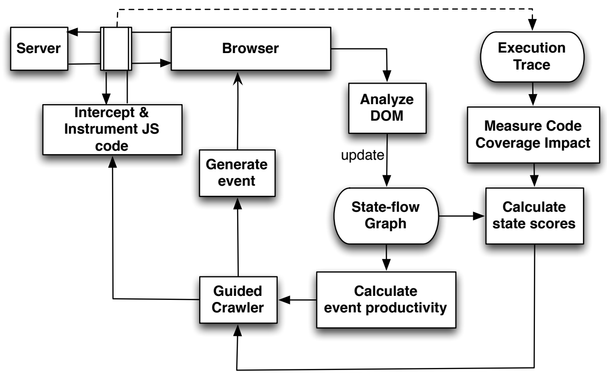


Fig. 2. Processing view of FEEDEX, our feedback-directed exploration approach.

possess the following properties to effectively cover different aspects of the application under test:

Functionality Coverage. A test suite generated from a derived test model can only cover the maximum functionality contained in the test model, and not more. For instance consider Figure 1. If the inferred test model does not capture events $e6$ and $e7$ the generated test suite will not be able to cover the underlying functionality behind those two events. Therefore, it is important to derive a test model that possesses adequate coverage of the web application when the end goal is test suite generation.

Navigational Coverage. The navigational structure of a web application allows its users to navigate it in various directions. For instance $s3$ and $s4$ are both on the same navigational path, whereas $s3$ and $s8$ are on different branches (Figure 1). To cover the navigational structure adequately, we believe that a test model should cover different navigational branches of the web application.

Page Structural Coverage. The structure of a webpage in terms of its internal DOM elements, attributes, and values, provides the main interaction interface with end users. Each page structure provides a different degree of content and functionality. To capture this structural functionality adequately, we believe a test model should cover heterogeneous DOM structures of the web application.

Size. The size of the derived test model has a direct impact on the number of generated test cases and event executions within each test case. Reducing the size of the test model can decrease both the cost of maintaining the generated test suite and the number of test cases that must be rerun after changes are made to the software [12]. Thus, while deriving test models, the size should be optimized as long as the reduction does not adversely influence the coverage. We consider the number of events (edges) in the SFG as an indicator of the model size since test case generation is done by traversing the sequence of events.

B. Feedback-directed Exploration Algorithm

In this paper, we refer to the process of executing candidate clickables of a state as *state expansion*. Figure 2 depicts an overview of our approach. At a high level, it dynamically analyzes the exploration history at runtime, including previously covered states and events, to anticipate about and decide a-priori (1) which *state* should be expanded next, and (2) from the state that is selected for expansion, which *event* (i.e., clickable element) should be exercised.

Algorithm 1: Feedback-directed Exploration

```
input : A Web 2.0 application $A$, the maximum
        exploration time $t$, the maximum number of states
        to explore $n$, exploration strategy $STR$
output: The inferred state-flow graph $SFG$
1  $SFG \leftarrow$ AddInitialState($A$)
   Procedure Explore() begin
2      while ConstraintSatisfied($t, n$) do
3          $PES \leftarrow$ GetPartiallyExpandedStates($SFG$)
4          for $s_i \in PES$ do
5              for $s_j \in PES$ & $s_j \neq s_i$ do
6                  $Score(s_i, s_j) \leftarrow$ GetScore($s_i, s_j, STR$)
7              $MinScore(s_i) \leftarrow$ GetMinScore($s_i$)
8          $s \leftarrow$ GetNextToExploreState($PES, MinScore$)
9          $C \leftarrow$ PrioritizeEvents($s$)
10         for $c \in C$ do
11             $browser$.Goto($SFG$.GetPath($s$))
12             $dom \leftarrow browser$.GetDOM()
13             $robot$.FireEvent($c$)
14             $new\_dom \leftarrow browser$.GetDOM()
15             if $dom$.HasChanged($new\_dom$) then
16                 $SFG$.Update($c, new\_dom$)
17                 Explore()
18     return $SFG$
```

Given a Web 2.0 application, a maximum exploration time t, and a maximum number of states to explore n, the intent is to maximize the test model coverage while reducing the model size within t. The rationale behind our technique is to reward states and events that have a substantial impact on different aspects of a test model (and penalize those that do not). Our exploration strategy, shown in Algorithm 1, applies a greedy approach to select and expand a partially expanded state.

Definition 2: A **partially expanded state** is a state, during exploration, which still contains one or more unexercised candidate clickables. □

In Figure 1, both $s6$ and $s8$ are partially expanded states. To expand the next partially expanded state, while exploring, we calculate a *state score* (explained in Section III-C) for each state that needs to be expanded, at runtime; this score prioritizes partially expanded states selectively so that a test model with enhanced aspects can be inferred for testing. Our key insight is that by dynamically measuring and expanding states with the highest scores, we can construct a state-flow graph of an application, yielding higher overall coverage. In addition to the state score, our approach prioritizes events based on their productivity ratio (described in Section III-D).

Algorithm 1 repeats the following main steps until the given time limit t, or state-space limit n is reached:

- For each partially expanded state, compute the *state score* with respect to other unexpanded states (Lines 4-6). The score of a state is the minimum pair-wise state score of that state (Line 7);
- Choose the state with the highest fitness (score) value as the next state to expand (Line 8);
- On the chosen state for expansion, prioritize the events based on their *event score* (Line 9);
- Take the browser to the chosen state and execute the highest ranked event according to the prioritized order (Lines 10-3);
- If the DOM state is changed after the event execution, update the SFG accordingly (Lines 14-16) and call the `Explore` procedure (line 17).

C. State Expansion Strategy

In this section, we describe our state expansion strategy, which is used to prioritize partially expanded states based on their overall contributions towards the different desired properties of the test model.

Code Coverage Impact. One primary objective for generating a test model is to achieve an adequate code coverage of the core functionality of the system. Note that we only consider the client-side code coverage and the server-side code coverage is not the goal of this work. The following example shows how state expansion can affect the code coverage.

Example 1: Figure 3 depicts a simple JavaScript code. Figure 4 shows the corresponding DOM state. Assume that the state-flow graph as shown in the Figure 1 was generated by exploring this simple example. If the crawler finishes after clicking on the *next* and the previous clickables multiple times, only the function `show` and the first two lines of the script code will be covered, i.e., 9 lines in total. Assuming that the total number of JavaScript lines of code is 27 (not all the code is shown in the figure), coverage percentage would be $\frac{9}{27} = 33.33\%$. This indicates that no matter what test case generation method we apply on the inferred SFG, we can not achieve a higher code coverage than 33.33%. However, if the crawler was able to click on the Update clickable before termination, the function `load` would be executed as well, yielding a coverage of $\frac{15}{27} = 55.55\%$. □

The *code coverage impact* for a state s, denoted by *CI*(s), is defined as the amount of increase in the code coverage after s is created. Considering the example again, when the *Index* page is loaded, the first two JavaScript lines are executed and thus initial coverage is $\frac{2}{27} = 7.4\%$, and the *CI*(*Index*)=0.074. After executing `onclick="show()"`, coverage would reach 33.33% with a 25.93% increase and thus $CI(s1) = 0.259$. For each newly discovered state, we calculate the *CI* in this manner. If a resulting state of an event is an already discovered state in the SFG, the *CI* value will be updated if the new value is larger. The *CI* score is taken into consideration when making decisions about expanding partially expanded states.

Path Diversity. Given a state-flow graph, states located on different branches are more likely to cover different navigational functionality (as in Section III-A) than those on a same path. Thus, guiding the exploration towards diversified paths can yield a better navigational functionality coverage.

```
myImg = new Array("1.jpg","2.jpg","3.jpg","4.jpg");
curIndex = 1;

function show (offset) {
  curIndex = curIndex + offset;
  if (curIndex > 4) {
    curIndex = 1; }
  if (curIndex == 0) {
    curIndex = 4 ; }
  document.imgSrc.src = myImg[curIndex - 1];
}
...
function load () {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/update/', true);
  xhr.onreadystatechange = function(e) {
       document.getElementById("container").innerHTML = ↩
            this.responseText
  };
  xhr.send();
}
```

Fig. 3. Simple JavaScript code snippet.

```
<body>
  <img name="imgSrc" id="imgSrc" src="1.jpg">
  <span onclick="show(-1)">previous</span>
  <span onclick="show(1)">next</span>
  ...
  <a href="#" onclick="load(); return false;">Update!</a>
  <div id="container"></div>
</body>
```

Fig. 4. A simple DOM instance.

Definition 3: A **simple event path** $\boldsymbol{P_{s_i}}$ of state s_i is a path on SFG from the Index node to s_i, without repeated nodes. □

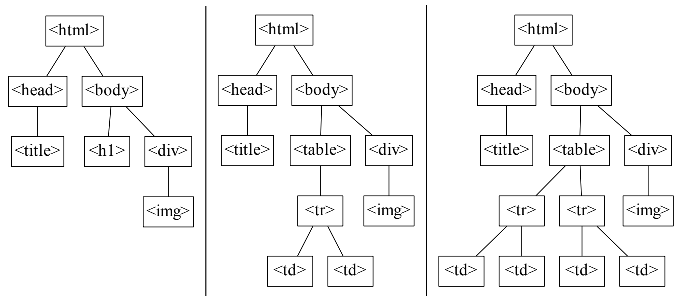
Note that in our definition we only consider "simple paths" without repeated nodes to avoid the ambiguity of having cycles and loops, such as those shown in Figure 1, where for instance, there exists an infinite number of paths from *Index* to $s2$. We formulate the *path similarity* of two states s_i and s_j as:

$$PathSim(s_i, s_j) = \frac{2 \times |P_{s_i} \cap P_{s_j}|}{|P_{s_i}| + |P_{s_j}|} \tag{1}$$

where $|P_{s_i}|$ is the length of P_{s_i}, and $|P_{s_i} \cap P_{s_j}|$ denotes the number of shared events between P_{s_i} and P_{s_j}. The path similarity notion captures the percentage of events shared by two simple paths. The less events two states share in their paths, the more diverse their navigational functionality. Thus, let $MaxPathSim(s_i, s_j)$ be the maximum path similarity of s_i and s_j considering all possible simple paths, from Index to s_i and s_j, respectively. The *path diversity* of s_i and s_j, denoted by $PD(s_i, s_j)$, is then calculated as:

$$PD(s_i, s_j) = 1 - MaxPathSim(s_i, s_j) \tag{2}$$

Example 2: Consider the running example of Figure 1. We calculate pair-wise path diversity scores for states $s4$, $s6$, and $s8$. $PathSim(s4, s6) = \frac{2 \times |P_{s4} \cap P_{s6}|}{|P_{s4}| + |P_{s6}|}$ can be computed in two ways as there exist two simple event paths from *Index* to $s6$:



(a) DOM tree t_1 (b) DOM tree t_2 (c) DOM tree t_3

Fig. 5. DOM tree comparison.

For P_{s6} through s1, $PathSim(s4, s6) = \frac{2\times1}{4+2} = \frac{2}{6} = \frac{1}{3}$, and for P_{s6} through $s5$, $PathSim(s4, s6) = \frac{2\times0}{4+2} = 0$. Thus $MaxPathSim(s4, s6) = \frac{1}{3}$ and $PD(s4, s6) = 1 - \frac{1}{3} = \frac{2}{3}$. Similarly, $PD(s4, s8)$=1-0=1, and $PD(s6, s8)$=1-0=1. This indicates that $s4$ and $s6$ are not that diverse with respect to each other, but they are very diverse with respect to s8. □

DOM Diversity. In many Web 2.0 applications, the DOM is mutated dynamically through JavaScript to reflect a state change, without requiring a URL change. Many of such dynamic DOM mutations are, however, incremental in nature and correspond to small delta changes, which might not be interesting from a testing perspective, especially given the space and time constraints. Therefore, guiding the exploration towards diversified DOM states can result in a better page structural coverage.

In most current AJAX crawlers, string representations of the DOM states are used for state comparison. The strings are compared by either calculating the edit distance [19], a strip and compare method [19], or by computing a hash of the content [9]. These approaches ignore the tree structure of DOM trees. To account for the actual structural differences, we adopt the *tree edit distance* between two ordered labeled trees, which was proposed [32] and implemented [27] as the minimum cost of a sequence of edit operations that transforms one tree into another. The operations include deleting a node and connecting its children to the parent, inserting a node between a node and the children of that node, and relabelling a node.

We define state DOM diversity as the normalized DOM tree edit distance. Let t_i and t_j be the corresponding DOM trees of two states s_i and s_j. The DOM diversity of s_i and s_j, denoted by $DD(s_i, s_j)$, is defined as:

$$DD(s_i, s_j) = \frac{TED(t_i, t_j)}{max(|t_i|, |t_j|)} \tag{3}$$

where $TED(t_i, t_j)$ is the tree edit distance between t_i and t_j, and $max(|t_i|, |t_j|)$ is the maximum number of nodes in t_i and t_j.

Example 3: Figure 5 depicts three DOM trees with $|t_1|$=7, $|t_2|$=10, and $|t_3|$=13. t_2 can be produced from t_1 by (1) relabelling <h1> in t_1 to <table>, and (2) inserting three nodes under <table>. Thus $TED(t_1, t_2) = 4$ and their DOM diversity equals $\frac{4}{10}$=0.4. Similarly $TED(t_1, t_3) = 7$ and thus their DOM diversity equals $\frac{7}{13}$=0.53. This shows t_3 is more DOM diverse than t_2 with respect to t_1. $TED(t_2, t_3) = 3$ and their DOM diversity equals $\frac{3}{13}$=0.23 □

Overall State Score. The state score is a combination of code coverage impact, path diversity, and DOM diversity. Our state expansion fitness function is a linear combination of the three metrics as follows:

$$\begin{aligned} Score(s_i, s_j) &= w_{CI} \cdot CI(s_i, s_j) + w_{PD} \cdot PD(s_i, s_j) \\ &\quad + w_{DD} \cdot DD(s_i, s_j) \end{aligned} \tag{4}$$

where, w_{CI}, w_{PD}, and w_{DD} are user-defined weights (between 0 and 1) for code coverage impact, path diversity, and DOM diversity, respectively.

D. Event Execution Strategy

The goal of our event execution strategy is to reduce the size of events sequences (edges) in the SFG, while preserving the coverage. Reducing the size of events is important since it reduces the size of generated test cases, which in turn minimizes the time overhead of test rerun [12].

Intuitively, we try to minimize the execution of events that are not likely to produce new states. We categorize web application user interface events into four groups based on their impact on the application state transitional behaviour: (1) An event that does not change the DOM state is called a *self-loop*, e.g., events that replace the DOM tree with an exact replica, e.g., refresh with no changes, or clear data in a form; (2) A *state-independent event* is an event that always causes the same resulting state, e.g., events that always result in the Index page; (3) A *state-dependent event* is an event that after its first execution, always causes the same state, when triggered from the same state. (4) A *Nondeterministic event* is an event that may results in a new state, regardless of where it is triggered from. Such events can result in different states when triggered from the same state. In Figure 1, for instance, $e0$ is a self-loop event, $e5$ is a state-independent event, and $e4$ is a state-dependent event.

A crawler that distinguishes between these different events can avoid self-loops, minimize state-independent and nondeterministic event executions, and emphasize state-dependent events to explore uncovered states. To that end, we define *event productivity* (EP) as follows.

Let $RS_i(e)$ denote the resulting state of the i-th execution of the event e, and n be the total number of executions of e (including the last execution). The event productivity ratio of e, denoted by $EP(e)$, is defined as:

$$EP(e) = \begin{cases} 1 & ; \text{if } n = 0 \\ \frac{\sum_{i=1}^{n} MinDD(RS_i(e))}{n} & ; \text{otherwise} \end{cases} \tag{5}$$

where $MinDD(RS_i(e)) = \min_{s \in SFG}\{DD(RS_i(e), s)\}$, i.e., the minimum diversity of $RS_i(e)$ and all existing states in the SFG. Note that $0 \leq EP(e) \leq 1$ and its value can change after each execution of e, while exploring.

The above definition captures three properties. Firstly, it gives the highest ratio to the unexecuted events (in case $n = 0$) since the resulting state is more likely to be a new state compared to already executed events. Naturally, this also helps in covering more of the JavaScript code, since the event-listeners typically trigger the execution of one or more JavaScript function(s). Secondly, it penalizes events that result in an already discovered state, such as self-loops and state-independent events, with $MinDD(RS_i(e))$=0. Thirdly, the productivity ratio is proportional to the structural diversity of the resulting state with respect to previously discovered states. This gives a higher productivity ratio to events that have resulted in more diverse structures, guiding the exploration towards more DOM diverse states.

Remark 1. We do not consider path-diversity (PD) in the calculation of EP. This is because when the execution of an event results in a new state, the resulting state shares much of its navigational path with the source state that leads to PD close to 0, which discourages new state discovery. On the other hand, if the resulting state is an already discovered state in the SFG, its shortest event path may not share much with other paths and therefore might get a high PD. This is also contrary to penalizing events causing already discovered states.

The next example shows how this definition is applied on self-loops, state-independent events, and forward-backward events.

Example 4: Consider Figure 1 again. For simplicity assume DOM diversity is 1 for new states and 0 for existing states. An example of a self-loop event is $e0$. Assume that the first observation of $e0$ is at state $Index$. Since we have never executed $e0$ before, $EP(e0) = 1$. After the first execution, $EP(e0) = \frac{0}{1} = 0$ because $MinDD(Index) = 0$ as the diversity of the source state ($Index$) and the resulting state ($Index$) is 0. By the second execution, say at $s1$, $EP(e0) = \frac{0+0}{1+1} = 0$. Now consider $e5$ as a state-independent event. For the first time, say at $s1$, $EP(e5) = 1$. After its first execution, since $s6$ is a new state (we assume diversity is 1), the productivity ratio will be $EP(e5) = \frac{1}{1} = 1$. However, the second execution results in a duplicate ($s6$ again) and thus $EP(e5) = \frac{1+0}{1+1} = 0.5$. Events $e2$ and $e3$ are examples of previous-next events (see Figure 4). After the first execution of $e3$ at state $s2$, $EP(e3) = \frac{0}{1} = 0$, because $s1$ was already in the SFG before executing $e3$ and thus $MinDD(s1)$=0. □

Remark 2. Prioritizing events only makes sense when we allow the crawler to exercise the same clickable multiple times. If the objective of the test model generation is to merely achieve high code coverage, then clicking on the same clickable again is unlikely to increase the code coverage; however, if the event is *state-dependent* or *nondeterministic*, multiple execution of the same clickable can have an impact on DOM and path diversity.

E. Implementation

Our proposed approach is implemented in a tool called FEEDEX, which is publicly available.[1]

[1]http://salt.ece.ubc.ca/content/feedex/

To explore Web 2.0 applications, we build on top of CRAWLJAX [19], [20]. The default engine supports both depth-first and breadth-first (multi-threaded, multi-browser) crawling strategies. FEEDEX replaces the default crawling strategy of CRAWLJAX (as described in [19]) with our feedback-directed exploration algorithm. The tool can be configured to constrain the state space, by setting parameters such as the maximum number of states to crawl, maximum crawling time, and search depth level.

As shown in Figure 2, FEEDEX intercepts and instruments the JavaScript code to collect execution traces while crawling. To parse and instrument the JavaScript code, we use Mozilla Rhino.[2] After each event execution, FEEDEX analyzes the collected execution trace and measures the code coverage impact, which in turn is used in our overall exploration decision making.

State path diversity is calculated according to Equation 2 by finding simple paths (Definition 3) from *Index* state to each state in the SFG. In order to compute each simple event path, we apply a DFS traversal from the *Index* state to a state node in the SFG and disregard already visited ones to avoid cycles or loops. For the computation of the tree edit distance for DOM state diversity as in Equation 3, we take advantage of the *Robust Tree Edit Distance* (RTED) algorithm [27], which has optimal $O(n^3)$ worst case complexity and is robust, where n is the maximum number of nodes of the two given trees. Considering that the number of nodes in a typical DOM tree is relatively small, the overhead of the DOM diversity computation is negligible.

In order to calculate the productivity ratio for an event, we store, for each event, a set of tuples comprising of source and target states, corresponding to all the previous executions of that event.

IV. EMPIRICAL EVALUATION

To assess the efficacy of the proposed feedback-directed exploration, we have conducted a controlled experiment. Our main research question is *whether our proposed exploration technique is able to derive a better test model compared to traditional exhaustive methods.*

Our experimental data along with the implementation of FEEDEX are available for download.[1]

A. Experimental Objects

Because our approach is targeted towards Web 2.0 applications, our selection criteria included applications that (1) use extensive client-side JavaScript, (2) are based on dynamic DOM manipulation and AJAX interactions, and (3) fall under different domains. Based on these criteria, we selected six open source applications from different domains which are shown in Table I. We use CLOC[3] to count the JavaScript lines of code (JS LOC). The reported LOC in Table I is excluding blank lines, comment lines, and JavaScript libraries such as jQuery, DWR, Scriptaculous, Prototype, Bootstrap,

[2]https://github.com/mozilla/rhino/

[3]http://cloc.sourceforge.net

TABLE I
EXPERIMENTAL OBJECTS (STATISTICS EXCLUDING BLANK/COMMENT LINES, AND JAVASCRIPT LIBRARIES).

ID	Name	JS LOC	Description	Resource
1	ChessGame	198	A JavaScript-based simple game	p4wn.sourceforge.net
2	TacirFormBuilder	209	A JavaScript-based simple HTML form builder	https://github.com/ekinertac/TacirFormBuilder
3	TuduList	782	An AJAX-based todo lists manager in J2EE and MySQL	julien-dubois.com/tudu-lists/
4	FractalViewer	1245	A JavaScript-based fractal zoomer	http://onecm.com/projects/canopy
5	PhotoGallery	1535	An AJAX-based photo gallery in PHP without MySQL	sourceforge.net/projects/rephormer
6	TinyMCE	26908	A JavaScript-based WYSIWYG editor	tinymce.com

TABLE II
STATE-SPACE EXPLORATION METHODS EVALUATED.

Method	Exploration Criteria
DFS	Expand the last partially expanded state
BFS	Expand the first partially expanded state
RND	Randomly expand a partially expanded state
FEEDEX	Expand the partially expanded state with the highest score ($w_{CI} = 1, w_{PD} = 0.5, w_{DD} = 0.3$), and prioritize events

and google-analytics. Note that we also exclude these libraries in the instrumentation step.

B. Experimental Setup

All our experiments are performed on a core-2 Duo 2.40GHz CPU with 3.46 GB memory, running Windows 7 and Firefox web browser.

1) Independent Variables: We compare our feedback-directed exploration approach with different traditional exploration strategies.

Exploration Constraints. We confine the designated exploration time for deriving a test model to 300 seconds (5 minutes) for all the experimental runs.[4] We set no limits on the crawling depth nor the maximum number of states to be discovered. We configure the tool to allow multiple executions of the same clickable element, as the same clickable can cause different resulting states.

State-space Exploration. Table II presents the different state expansion strategies we evaluate in the first part of our experiment. The first three (DFS, BFS, RND) are exhaustive crawling methods. DFS expands states in a depth-first fashion, i.e., the last discovered state would be expanded next. BFS applies breath-first exploration by expanding the first discovered state next. RND performs random exploration in which a partially expanded state is chosen uniformly at random to be expanded next. Note that for these traditional exhaustive exploration methods we consider the original event execution strategy, i.e., a user-defined order for clicking on elements. For this experiment, the order is defined as: `A`, `DIV`, `SPAN`, `IMG`, `BUTTON`, `INPUT`, and `TD`.The last method is an instantiation of our feedback-directed exploration score (See Equation 4) where $w_{CI} = 1$, $w_{PD} = 0.5$, and $w_{DD} = 0.3$. We empirically evaluated different weightings and found this setting among other settings can generally produce good results. FEEDEX prioritizes clickable elements based on the event productivity ratio EP (Equation 5) and executes them in that order (see Section III-D).

2) Dependent Variables: We analyze the impact of different state-space exploration strategies on the code coverage, the overall average path diversity and DOM diversity, as well as the size of the derived test model.

Code Coverage Score. We measure the final statement code coverage achieved by each method, after 5 minutes of exploration. In order to measure the JavaScript code coverage, we instrument the JavaScript code as explained in Section III-E.

Diversity Scores. In order to measure the path diversity of a SFG, we measure average pair-wise navigational diversity of leaf nodes (states without any outgoing edges) since the position of the leaf nodes in the graph is an indication of the diversity of its event paths (i.e., paths from the Index node to the leaves). The *AvgPD* is defined as:

$$AvgPD(SFG) = \frac{\sum_{\forall s_i, s_j \in L(V)} PD(s_i, s_j)}{2 \cdot m \cdot (m-1)} \quad (6)$$

where $L(V)$ denotes the set of leaf nodes in the SFG and $m = |L(V)|$, i.e., the number of leaf nodes. This value is in the range of 0 to 1.

To assess the page structural diversity of the derived test models from each method, we compute the overall average pair-wise structural diversity (*AvgDD*) in the derived SFG as:

$$AvgDD(SFG) = \frac{\sum_{\forall s_i, s_j \in V} DD(s_i, s_j)}{2 \cdot n \cdot (n-1)} \quad (7)$$

where $n = |V|$, i.e., the number of states in the SFG and *AvgDD* is in the range of 0 to 1.

Test Model and Test Suite Size. As discussed in Section III-A, the derived test model (SFG) can be used to generate test cases through different graph traversal and coverage methods. The event size of the derived test model has a direct impact on the number and size of test cases that can be generated, regardless of the test generation method used. We consider (1) the number of edges (events) in the SFG, as the *size of test model*, and (2) the number of "distinct" simple event paths in the SFG, as the *size of test suite* (equal to the number of all possible Selenium[5] test cases generated from the test model). Two simple event paths (Definition 3) are distinct if they visit different sequence of states. Note that simple event paths can not have cycles or loops. As described in Section III-E, simple paths are generated using DFS traversal from the Index state to every other state in the SFG.

[4]Dedicating 5–10 minutes to test generation is acceptable in most testing environments [13].

[5]http://seleniumhq.org

TABLE III
RESULTS OF DIFFERENT EXPLORATION METHODS.

Exploration Method	Statement Coverage	Navigational Path Diversity	Page Structural Diversity	Size of the Test Model	Size of the Test Suite
DFS	37.55%	0.010	0.035	578	247
BFS	43.82%	0.410	0.065	475	165
RND	40.44%	0.369	0.066	450	241
FEEDEX	48.13%	0.443	0.081	280	95
Improvement (min–max%)	10–28%	7–4000%	23–130%	38–86%	42–61%

C. Results and Findings

Table III shows the results of different exploration methods. We report the average values obtained from the six experimental objects. For the random state expansion method (RND), we report the average values over five runs. The table shows statement code coverage, navigational path diversity (*AvgPD*), page structural diversity (*AvgDD*), number of edges (events) as well as the number of distinct paths in the derived test model.

Results present that for FEEDEX, there is on average between (min–max%) 10–28% improvement on the final statement coverage (after 5 minutes), 7–4000% on path diversity, 23–130% on page structural diversity, 38–86% reduction in the number of edges, and 42–61% reduction in the distinct paths in the test model. It is evident from the results that FEEDEX is capable of generating test models that are enhanced in all aspects compared to the traditional exploration methods. Test models created using FEEDEX have smaller size (thus need less time to execute test cases) and higher code coverage and state diversity. The simultaneous improvement in all the evaluation criteria points to the the effectiveness of our state expansion and event execution strategy.

Given the limited amount of exploration time, we believe the achieved improvements for code coverage is substantial. Our approach also significantly increases the average DOM diversity compared to the RND method. Note that the main reason for having small *AvgDD* values for all the methods, is the normalization by the large number of possible pair-wise links in the computation of *AvgDD* (Equations 7). There is a low improvement of 7% achieved by FEEDEX over BFS with respect to the navigational path diversity (*AvgPD*). This is expected due to the fact that BFS, by its nature, generates models with many branches that do not share too many event paths, and are more sparse, compared to the other methods. The amount of shared paths also contribute to reducing the number of distinct paths, thus reducing the size of test suite. Therefore, although there is some improvement, FEEDEX can not improve *AvgPD* significantly compared to BFS. Among traditional methods, DFS is the least effective. Specifically, *AvgPD* achieved by DFS is much lower than others due to the fact that it keeps expanding states in the same branch in most cases, and FEEDEX can improve the navigational diversity significantly (4000%).

While evaluating different weights in FEEDEX scoring function, we found that assigning 1 to w_{DD} and 0 to the rest, is effective in producing more structural diverse models, if an application has many different DOM states, such as PhotoGallery, and not that effective for applications with minimum DOM changes, such as ChessGame (a chess board that remains relatively the same). Similarly, assigning 1 to w_{PD} and 0 to the rest, is more effective in increasing *AvgPD* in applications with many navigational branches (features) such as TinyMCE. In general, feedback-directed exploration technique, with the settings we used, is superior over the exhaustive methods with respect to all aspects of generated test models. Considering the significant improvements achieved by using FEEDEX, the computational overhead of our method is negligible.

V. DISCUSSION

Limitations. A limitation of FEEDEX is that the maximum effectiveness depends on the weights used in the scoring function (Equation 4) and on the application state-space and functionality, which is not known in advance. For example in our other experiments we observed that setting $w_{DD}, w_{PD} = 0$ and $w_{CI} = 1$, can generate test models with higher code coverage but less diversity and larger test model size. In general, the setting we used in this paper can generate a test model which enhances all aspects of a test model compared to traditional methods. We do not claim that Equation 4 is the best way to combine CI, PD, and DD, or the weights that we empirically found, always outperform previous work. Instead, we rely on the intuition of the feedback-directed heuristic which we believe effectively works most of the time.

Applications. The main application of our technique is in automated testing of Web 2.0 applications. The automatically derived test model, for instance, can be used to generate different types of test cases used for invariant-based assertions after each event, regression testing, cross-browser testing [4], [7], [16], [18], [20], [22]. Our exploration technique towards higher client-side code coverage can also help with a more accurate detection of unused/dead JavaScript code [21]. Moreover, FEEDEX is generic in terms of its scoring function, thus by changing the fitness function (line 6 of Algorithm 1), it can generate a test model based on other user-defined criteria.

Threats to Validity. A threat to the external validity of our experiment is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat we selected our experimental objects from four different domains: gallery, task management, game, and editor. The selected Web 2.0 applications exhibit variations in functionality and structure and we believe they are representative of modern web applications.

One threat to the internal validity of our experiment is related to the evaluation metrics including *AvgDD* and *AvgPD* proposed by the authors of the paper based on which the effectiveness of FEEDEX is evaluated. However, we believe these metrics capture different properties of a test model as described in Section III.

With respect to reliability of our results, FEEDEX and all the web-based systems are publicly available, making the experiment reproducible.

VI. RELATED WORK

Crawling Web Applications. Web crawling techniques for traditional hypertext-based websites have been extensively studied in the past two decades. More related to our work is the idea behind "scoped crawling" [25] to constrain the exploration to webpages that fall within a particular scope; this way, obtaining *relevant content* becomes more efficient than through a comprehensive crawl. Well-known examples of scoped crawling include hidden-web crawling [15] and focused crawling [17]. Scope can also be defined in terms of geography, language, genre, format, and other content-based properties of webpages. All these techniques focus on the *textual contents* of webpages. Our approach, on the other hand, is geared towards functionality coverage of a web application under test.

Crawling modern Web 2.0 applications engenders more challenges due to the dynamic client-side processing (through JavaScript and AJAX calls), and is relatively a new research area [5], [9], [19]. Many web testing techniques [2], [4], [7], [16], [18], [20], [22], [23], [33], [34] depend on data gathered and models generated through crawling. However, such techniques rely on exhaustive search methods. To the best of our knowledge, this work is the first to propose a feedback-directed exploration of web applications that enhances different aspects of a test model.

Some metrics have been proposed to measure crawling effectiveness and diversity. Marchetto et al. [2] propose crawlability metrics that capture the degree to which a crawler is able to explore the web application. These metrics combine dynamic measurements such as code coverage, with static indicators, such as lines of code and cyclomatic complexity. Their crawlibility metrics are geared towards the server-side of web applications. Regarding our diversity metrics, although the notion of diversity has been used for classifying search query results [1], [28], [29], we propose new metrics for capturing state and navigational diversity as two aspects of a web application test model.

More related to our work are guided test generation [33] and efficient AJAX crawling techniques [5], [6], [8]. Thummalapenta et al. [33] recently proposed a guided test generation technique for web applications, called WATEG. Their work crawls the application in an exhaustive manner, but directs test generation towards higher coverage of specific business rules, i.e., business logic of an application's GUI. Our work differs from WATEG in two aspects. Firstly, we guide the exploration and not the test generation, and secondly, our objective is to increase code coverage and state diversity, and at the same time decrease test model size. Efficient strategies for AJAX crawling [5], [6], [8] try to discover as many states as possible in the shortest amount of time. The goal of our work is, however, not to crawl the complete state-space as soon as possible (which is infeasible for many industrial web applications), but to drive a partial model, which adequately covers different desired properties of the application.

Static analysis. Researchers have used static analysis to detect bugs and security vulnerabilities in web apps [11], [35]. Jensen et al. [14] model the DOM as a set of abstract JavaScript objects. However, they acknowledge that there are substantial gaps in their static analysis, which can result in false-positives. Because JavaScript is a highly dynamic language, such static techniques typically restrict themselves to a subset of the language. In particular, they ignore dynamic JavaScript interactions with the DOM, which is error-prone and challenging to detect and resolve [24].

Dynamic Analysis. Dynamic analysis and automated testing of JavaScript-based web applications has gained more attention in the recent past. Marchetto et al. [16] proposed a state-based testing technique for AJAX applications in which semantically interacting event sequences are generated for testing. Mesbah et al. proposed [20] an automated technique for generating test cases with invariants from models inferred through dynamic crawling. JSArt [22] and DoDOM [26] dynamically derive invariants for the JavaScript code and the DOM respectively. Such inferred invariants are used for automated regression and robustness testing.

Artemis [3] is a JavaScript testing framework that uses feedback-directed random testing to generate test inputs. Compared to FEEDEX, Artemis randomly generates test inputs, executes the application with those inputs and uses the gathered information to generated new test inputs. FEEDEX on the other hand, explores the application by dynamically running it and building a state-flow graph that can be used for test generation. The strength of FEEDEX is that it guides the application at runtime towards a better code and more diverse navigational and structural coverage.

Kudzu [30] combines symbolic execution of JavaScript with random test generation to explore sequence of events and produce input values depending on the execution of the control flow paths. Their approach uses a complex string constraint solver to increase the code coverage by producing acceptable input string values. In our work we did not intend to increase the code coverage by considering the problem of input generation for the crawler and only focused on improving the crawling strategy. Compared to Kudzu, FEEDEX is much simpler, more automated, and does not require such heavy computation.

VII. CONCLUSIONS AND FUTURE WORK

An enabling technique commonly used for automating web application testing and analysis is crawling. In this paper we proposed four aspects of a test model that can be derived by crawling, including *functionality coverage*, *navigational coverage*, *page structural coverage*, and *size of the test model*. We have presented a generic feedback-directed exploration approach, called FEEDEX, to guide the exploration towards client-side states and events that produce a test model with enhanced aspects. Our experiment on six Web 2.0 applications shows that FEEDEX yields higher code coverage, diversity, and smaller test models, compared to traditional exhaustive methods (DFS, BFS, and random).

In this work, we only measure the client-side coverage. In future work, we intend to include the server-side code coverage in the feedback loop to our guided exploration engine. Our state expansion method is currently based on a memory-less greedy algorithm. We plan to incorporate a learning mechanism into the approach to improve its effectiveness. Finally, we will continue to conduct more experiments on different types of web applications.

Acknowledgment: This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme and Alexander Graham Bell Canada Graduate Scholarship (CGS-D).

REFERENCES

[1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *Proc. of the International Conference on Web Search and Data Mining*, pages 5–14. ACM, 2009.

[2] N. Alshahwan, M. Harman, A. Marchetto, R. Tiella, and P. Tonella. Crawlability metrics for web applications. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 151–160. IEEE Computer Society, 2012.

[3] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. International Conference on Software Engineering (ICSE)*, pages 571–580, 2011.

[4] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proc. of the International World Wide Web Conference (WWW)*, pages 654–668, 2002.

[5] K. Benjamin, G. von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I.-V. Onut. A strategy for efficient crawling of rich internet applications. In *Proc. of the International Conference on Web Engineering (ICWE)*, pages 74–89. Springer-Verlag, 2011.

[6] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, G.-V. Jourdan, G. von Bochmann, and I. V. Onut. Building rich internet applications models: Example of a better strategy. In *Proc. of the International Conference on Web Engineering (ICWE)*. Springer, 2013.

[7] S. R. Choudhary, M. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 171–180. IEEE Computer Society, 2012.

[8] M. E. Dincturk, S. Choudhary, G. von Bochmann, G.-V. Jourdan, and I. V. Onut. A statistical approach for efficient crawling of rich internet applications. In *Proc. of the International Conference on Web Engineering (ICWE)*, pages 362–369. Springer, 2012.

[9] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax crawl: Making Amax applications searchable. In *Proc. of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89. IEEE Computer Society, 2009.

[10] V. Garousi, A. Mesbah, A. Betin Can, and S. Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 2013.

[11] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. of the International World Wide Web Conference (WWW)*, pages 561–570. ACM, 2009.

[12] M. Harrold, R. Gupta, and M. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, first edition, 2010.

[14] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. ESEC/FSE*, pages 59–69. ACM, 2011.

[15] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[16] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 121–130. IEEE Computer Society, 2008.

[17] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Transactions on Internet Technology (TOIT)*, 4(4):378–419, 2004.

[18] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, 2011.

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

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

[21] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proc. of the IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2013.

[22] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. of the Internatinoal Conference on Web Engineering (ICWE)*, pages 238–252. Springer, 2012.

[23] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013.

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

[25] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[26] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing. In *Proc. of the International Symposium on Sw. Reliability Eng. (ISSRE)*, pages 191–200. IEEE Computer Society, 2010.

[27] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011.

[28] T. Sakai. Evaluation with informational and navigational intents. In *Proc. of the International Conference on World Wide Web (WWW)*, pages 499–508. ACM, 2012.

[29] R. Santos, C. Macdonald, and I. Ounis. Selectively diversifying web search results. In *Proc. of the International Conference on Information and knowledge management*, pages 1179–1188. ACM, 2010.

[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[31] P. Srinivasan, F. Menczer, and G. Pant. A general evaluation framework for topical crawlers. *Information Retrieval*, 8(3):417–447, 2005.

[32] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[33] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE Computer Society, 2013.

[34] P. Tonella and F. Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):103–127, 2004.

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