Leveraging Existing Tests in Automated Test Generation for Web Applications

Amin Milani Fard Mehdi Mirzaaghaei Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
{aminmf, mehdi, amesbah}@ece.ubc.ca

ABSTRACT

To test web applications, developers currently write test cases in frameworks such as SELENIUM. On the other hand, most web test generation techniques rely on a crawler to explore the dynamic states of the application. The first approach requires much manual effort, but benefits from the domain knowledge of the developer writing the test cases. The second one is automated and systematic, but lacks the domain knowledge required to be as effective. We believe combining the two can be advantageous. In this paper, we propose to (1) mine the human knowledge present in the form of input values, event sequences, and assertions, in the human-written test suites, (2) combine that inferred knowledge with the power of automated crawling, and (3) extend the test suite for uncovered/unchecked portions of the web application under test. Our approach is implemented in a tool called TESTILIZER. An evaluation of our approach indicates that TESTILIZER (1) outperforms a random test generator, and (2) on average, can generate test suites with improvements of up to 150% in fault detection rate and up to 30% in code coverage, compared to the original test suite.

Categories and Subject Descriptors

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

General Terms

Verification, Algorithms, Experimentation

Keywords

Automated test generation; test reuse; web applications

1. INTRODUCTION

Web applications have become one of the fastest growing types of software systems today. Testing modern web applications is challenging since multiple languages, such as HTML, JavaScript, CSS, and server-side code, interact with each other to create the application. The final result of all these interactions at runtime is manifested through the Document Object Model (DOM) and presented to the end-user in the browser. To avoid dealing with all these complex interactions separately, many developers treat the web application as a black-box and test it via its manifested DOM, using testing frameworks such as SELENIUM [6]. These DOM-based test cases are written manually, which is a tedious process with an incomplete result.

On the other hand, many automated testing techniques [13, 19, 28, 31] are based on crawling to explore the state space of the application. Although crawling-based techniques automate the testing to a great extent, they are limited in three areas:

Input values: Having valid input values is crucial for proper coverage of the state space of the application. Generating these input values automatically is challenging since many web applications require a specific type, value, and combination of inputs to expose the hidden states behind input fields and forms.

Paths to explore: Industrial web applications have a huge state space. Covering the whole space is infeasible in practice. To avoid unbounded exploration, which could result in state explosion, users define constraints on the depth of the path, exploration time or number of states. Not knowing which paths are important to explore results in obtaining a partial coverage of a specific region of the application.

Assertions: Any generated test case needs to assert the application behaviour. However, generating proper assertions automatically without human knowledge is known to be challenging. As a result, many web testing techniques rely on generic invariants [19] or standard validators [11] to avoid this problem.

These two approaches work at the two extreme ends of the spectrum, namely, fully manual or fully automatic. We believe combining the two can be advantageous. In particular, humans may have the domain knowledge to see which interactions are more likely or important to cover than others; they may be able to use domain knowledge to enter valid data into forms; and, they might know what elements on the page need to be asserted and how. This knowledge is typically manifested in manually-written test cases.

In this paper, we propose to (1) mine the human knowledge existing in manually-written test cases, (2) combine that inferred knowledge with the power of automated crawling, and (3) extend the test suite for uncovered/unchecked portions of the web application under test. We present our technique and tool called TESTILIZER, which given a set of SELENIUM test cases TC and the URL of the application, automatically infers a model from TC, feeds that model to a

ASE'14, September 15 – 19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642991.

crawler to expand by exploring uncovered paths and states, generates assertions for newly detected states based on the patterns learned from TC, and finally generates new test cases.

To the best of our knowledge, this work is the first to propose an approach for extending a web application test suite by leveraging existing test cases. The main contributions of our work include:

- A novel technique to address limitations of automated test generation techniques by leveraging human knowledge from existing test cases.
- An algorithm for mining existing test cases to infer a model that includes (1) input data, (2) event sequences, (3) and assertions, and feeding and expanding that model through automated crawling.
- An algorithm for reusing human-written assertions in existing test cases by exact/partial assertion matching as well as through a learning-based mechanism for finding similar assertions.
- An implementation of our technique in an open source tool, called TESTILIZER [7].
- An empirical evaluation of the efficacy of the generated test cases on four web applications. On average, TESTILIZER can generate test suites with improvements of up to 150% on the fault detection rate and up to 30% on the code coverage, compared to the original test suite.

2. BACKGROUND AND MOTIVATION

In practice, web applications are largely tested through their DOM using frameworks such as SELENIUM. The DOM is a dynamic tree-like structure representing user interface elements in the web application, which can be dynamically updated through client-side JavaScript interactions or server-side state changes propagated to the client-side. DOM-based testing aims at bringing the application to a particular DOM state through a sequence of actions, such as filling a form and clicking on an element, and subsequently verifying the existence or properties (e.g., text, visibility, structure) of particular DOM elements in that state. Figure 1 depicts a snapshot of a web application and Figure 2 shows a simple DOM-based (SELENIUM) test case for that application.

For this paper, a DOM state is formally defined as:

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.* □

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 subsection 3.1 and section 4.

Motivation. Overall, our work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be exploited for tackling some of the challenges in automated web application test generation. Another motivation behind our work is that manually written test cases typically correspond to the most common *happy-paths* of the application that are covered. Automated analysis can subsequently expand these to cover unexplored *bad-weather* application behaviour.

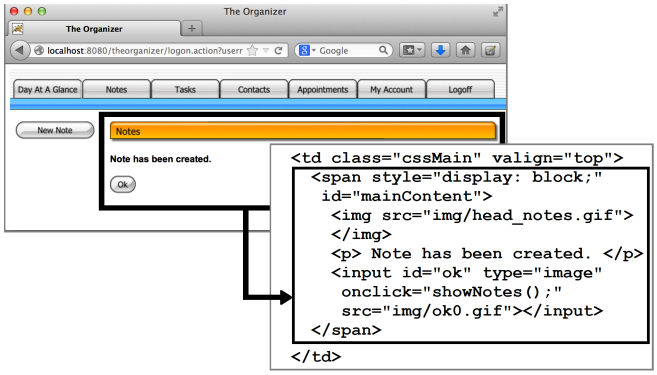

Figure 1: A snapshot of the running example and its partial DOM structure.

```
1  @Test
2  public void testAddNote(){
3   get("http://localhost:8080/theorganizer/");
4   findElement(By.id("logon_username")).sendKeys("↩
        user");
5   findElement(By.id("logon_password")).sendKeys("↩
        pswd");
6   findElement(By.cssSelector("input type="image"")).↩
        click();
7   assertEquals("Welcome to The Organizer!", ↩
        closeAlertAndGetItsText());
8   findElement(By.id("newNote")).click();
9   findElement(By.id("noteCreateShow_subject")).↩
        sendKeys("Running Example");
10  findElement(By.id("noteCreateShow_text")).sendKeys↩
        ("Create a simple running example");
11  findElement(By.cssSelector("input type="image"")).↩
        click();
12  assertEquals("Note has been created.", driver.↩
        findElement(By.id("mainContent")).getText());
13  findElement(By.id("logoff")).click();
14 }
```

Figure 2: A human-written DOM-based (Selenium) test case for the Organizer.

Running example. Figure 1 depicts a snapshot of the Organizer [4], a web application for managing notes, contacts, tasks, and appointments, which we use as a running example to show how input data, event paths, and assertions can be leveraged from the existing test cases to generate effective test cases.

Suppose we have a small test suite that verifies the application's functionality for "adding a new note" and "adding a new contact". Due to space constraints, we only show the `testAddNote` test case in Figure 2. The test case contains valuable information regarding how to log onto the Organizer (Lines 4–5), what data to insert (Lines 9–10), where to click (Lines 6, 8, 11, 13), and what to assert (Lines 7, 12).

We believe this information can be extracted and leveraged in automated test generation. For example, the paths (i.e., sequence of actions) corresponding to these covered functionalities can be used to create an abstract model of the application, shown in thick solid lines in Figure 3. By feeding this model that contains the event sequences and input data leveraged from the test case to a crawler, we can explore *alternative paths* for testing, shown as thin lines in Figure 3; alternative paths for deleting/updating a note/-contact that result in newly detected states (i.e., s10 and s11) are highlighted as dashed lines.

Further, the assertions in the test case can be used as guidelines for generating new assertions on the newly de-

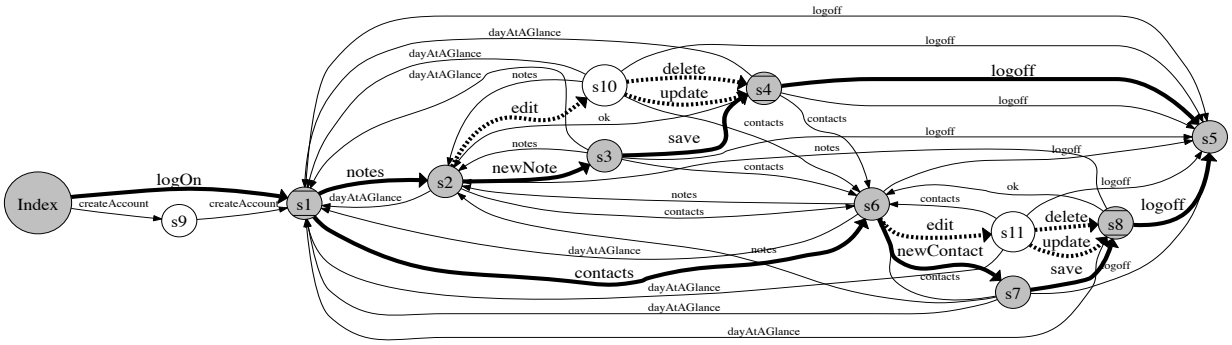

Figure 3: Partial view of the running example application's state-flow graph.

tected states along the alternative paths. These original assertions can be seen as parallel lines inside the nodes on the graph of Figure 3. For instance, line 12 of Figure 2 verifies the existence of the text "Note has been created" for an element (`span`) with `id="mainContent"`, which can be assigned to the DOM state $s4$ in Figure 3.

By exploring alternative paths around existing paths and learning assertions from existing assertions, new test cases can be generated. For example the events corresponding to states $\langle Index, s1, s2, s10, s4, s5\rangle$ can be turned into a new test method `testUpdateNote()`, which on state $s4$, verifies the existence of a `<span>` element with `id="mainContent"`. Further, patterns found in existing assertions can guide us to generate similar assertions for newly detected states (e.g., $s9$, $s10$, $s11$) that have no assertions.

3. APPROACH

Figure 4 depicts an overview of our approach. At a high level, given the URL of a web application and its human-written test suite, our approach mines the existing test suite to infer a model of the covered DOM states and event-based transitions including input values and assertions (blocks 1, 2, and 3). Using the inferred model as input, it explores alternative paths leading to new DOM states, thus expanding the model further (blocks 3 and 4). Next it regenerates assertions for the new states, based on the patterns found in the assertions of the existing test suite (block 5), and finally generates a new test suite from the extended model, which is a superset of the original human-written test suite (block 6). We discuss each of these steps in more details in the following subsections.

3.1 Mining Human-Written Test Cases

To infer an initial model, in the first step, we (1) instrument and execute the human-written test suite T to mine an intermediate dataset of test operations. Using this dataset, we (2) run the test operations to infer a state-flow graph (3) by analyzing DOM changes in the browser after the execution of each test operation.

Instrumenting and executing the test suite. We instrument the test suite (block 1 Figure 4) to collect information about DOM interactions such as elements accessed in actions (e.g., clicks) and assertions as well as the structure of the DOM states covered.

Definition 2 (Manual-test Path). *A manual-test path is the sequence of event-based actions performed while executing a human-written test case $t \in T$.* □

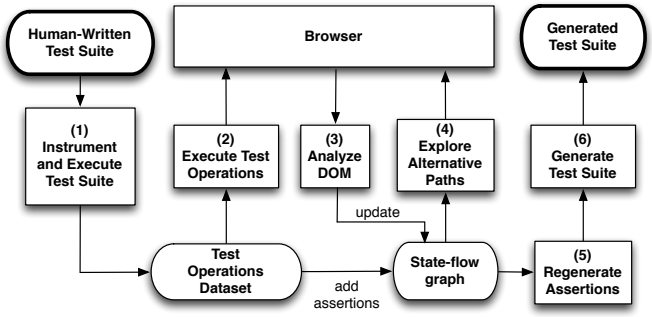

Figure 4: Processing view of our approach.

Definition 3 (Manual-test State). *A manual-test state is a DOM state located on a manual-test path.* □

The instrumentation hooks into any code that interacts with the DOM in any part of the test case, such as test setup, helper methods, and assertions. Note that this instrumentation does not affect the functionality of the test cases (more details in Section 4). By executing the instrumented test suite, we store all observed manual-test paths as an intermediate dataset of test operations:

Definition 4 (Test Operation). *A test operation is a triple <action, target, input>, where action specifies an event-based action (e.g., a click), or an assertion (e.g., verifying a text), target pertains to the DOM element to perform the action on, and input specifies input values (e.g., data for filling a form).* □

The sequence of these test operations forms a dataset that is used to infer the initial model. For a test operation with an assertion as its action, we refer to the target DOM element as a *checked element*, defined as follows:

Definition 5 (Checked Element). *A checked element $ce \in v_i$ is an element in the DOM tree in state v_i, whose existence, value, or attributes are checked in an assertion of a test case $t \in T$.* □

For example in line 12 of the test case in Figure 2, the text value of the element with ID `"mainContent"` is asserted and thus that element is a checked element. Part of the DOM structure at this state is shown in Figure 1, which depicts the checked element `<span id="mainContent">`.

For each checked element we record the element location strategy used (e.g., XPath, ID, tagname, linktext, or csss-elector) as well as the access values and innerHTML text.

Algorithm 1: State-Flow Graph Inference

```
   input : A Web application url URL, a DOM-based test
           suite TS, crawling constraints CC
   output: A state-flow graph SFG
   Procedure InferSFG(URL,TS,CC)
   begin
1      TS_inst ← Instrument(TS)
2      Execute(TS_inst)
3      TOP ← ReadTestOperationDataset()
4      SFG_init ← ∅
5      browser.Goto(URL)
6      dom ← browser.GetDOM()
7      SFG_init.AddInitialState(dom)
8      for top ∈ TOP do
9          C ← GetClickables(top)
10         for c ∈ C do
11             assertion ← GetAssertion(top)
12             dom ← browser.GetDOM()
13             robot.FireEvent(c)
14             new_dom ← browser.GetDOM()
15             if dom.HasChanged(new_dom) then
16                 SFG_init.Update(c, new_dom, assertion)
17         browser.Goto(URL)
18     SFG_ext ← SFG_init
19     ExploreAlterntivePaths(SFG_ext,CC)
20     return SFG_ext
   Procedure ExploreAlterntivePaths(SFG,CC) begin
21     while ConstraintSatisfied(CC) do
22         s ← GetNextToExploreState(SFG)
23         C ← GetCandidateClickables(s)
24         for c ∈ C do
25             browser.Goto(SFG.GetPath(s))
26             dom ← browser.GetDOM()
27             robot.FireEvent(c)
28             new_dom ← browser.GetDOM()
29             if dom.HasChanged(new_dom) then
30                 SFG.Update(c, new_dom)
31                 ExploreAlterntivePaths(SFG,CC)
```

This information is later used in the assertion generation process (in Section 3.3).

Constructing the initial model. We model a web application as a *State-Flow Graph* (SFG) [18, 19] that captures the dynamic DOM states as nodes and the event-driven transitions between them as edges.

Definition 6 (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 an abstract DOM state $\mathcal{DS}$ of $\mathcal{W}$, with a labelling function $\Phi : \mathcal{V} \rightarrow \mathcal{A}$ that assigns a label from $\mathcal{A}$ to each vertex in $\mathcal{V}$, where $\mathcal{A}$ is a finite set of DOM-based assertions in a test suite.*
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.* □

An example of such a partial SFG is shown in Figure 3. The abstract DOM state is an abstracted version of the DOM tree of a web application, displayed on the web browser at runtime. This abstraction can be conducted by using a DOM string edit distance, or by disregarding specific aspects of a DOM tree (such as irrelevant attributes, time stamps, or styling issues) [19]. The state abstraction plays an important role in reducing the size of SFG since many subtle DOM differences do not represent a proper state change, e.g., when a row is added to a table.

Algorithm 1 shows how the initial SFG is inferred from the manual-test paths. First the initial *index* state is added as a node to an empty SFG (Algorithm 1, lines 5–7). Next, for each test operation in the mined dataset (TOP), it finds DOM elements using the locator information and applies the corresponding actions. If an action is a DOM-based assertion, the assertion is added to the set of assertions of the corresponding DOM state node (Algorithm 1, lines 8–17). The state comparison to determine a new state (line 15) is carried out via a state abstraction function (more explanation in Section 4).

3.2 Exploring Alternative Paths

At this stage, we have a state-flow graph that represents the covered states and paths from the human-written test suite. In order to further explore the web application to find alternative paths and new states, we seed the graph to an automated crawler (block 4 Figure 4).

The exploration strategy can be conducted in various ways: (1) remaining close to the manual-test paths, (2) diverging [20] from the manual-test paths, or (3) randomly exploring. However, in this work, we have opted for the first option, namely staying close to the manual-test paths. The reason is to maximize the potential for reuse of and learning from existing assertions. Our insight is that if we diverge too much from the manual-test paths and states, the human-written assertions will also be too disparate and thus less useful.

To find alternative paths, events are automatically generated on DOM elements and if as a result the DOM is mutated, the new state and the corresponding event transition are added to the SFG. Note that the state comparison to determine a new state (line 29) is carried out via the same state abstraction function used before (line 15). The procedure `ExploreAlternativePaths` (Algorithm 1, lines 21–31) recursively explores the application until a pre-defined constraint (e.g., maximum time, or number of states) is reached. The algorithm is guided by the manual-test states while exploring alternative paths (Line 22); `GetNexToExploreState` decides which state should be expanded next. It gives the highest priority to the manual-test states and when all manual-test states are fully expanded, the next immediate states found are explored further. More specifically, it randomly selects a manual-test state that contains unexercised candidate clickables and navigates the application further through that state. The `GetCandidateClickable` method (Line 23) returns a set of candidate clickables that can be applied on the selected state. This process is repeated until all manual-test states are fully expanded.

For example, consider the manual-test sates shown in grey circles in Figure 3. The method starts by randomly selecting a state, e.g., $s2$, navigating the application to reach to that state from the *Index* state, and firing an event on $s2$ resulting in an new state $s10$.

3.3 Regenerating Assertions

The next step is to generate assertions for the new DOM states in the extended SFG (block 5 Figure 4). In this work, we propose to leverage existing assertions to regenerate new ones. By analyzing human-written assertions we can infer information regarding (1) portions of the page that are con-

Algorithm 2: Assertion Regeneration

```
   input : An extended state-flow graph SFG = < r, V, E, L >
   Procedure RegenerateAssertions(SFG)
   begin
       /*Learn from DOM elements in the manual-test states*/
1      dataset ←MakeDataset(SFG.GetManualTestStates())
2      Train(dataset)
3      for s_i ∈ V do
4          for ce ∈ s_i.GetCheckedElements() do
5              assert ← ce.GetAssertion()
6              cer ← ce.GetCheckedElementRegion()
7              s_i.AddRegFullAssertion(cer)
8              for s_j ∈ V & s_j ≠ s_i do
9                  dom ← s_j.GetDOM()
                   /*Generate exact element assertion for s_j*/
10                 if ElementFullMatched(ce, dom) then
11                     s_j.ReuseAssertion(ce,assert)
12                 else if ElementTagAttMatched(ce, dom)
                   then
13                     s_j.AddElemTagAttAssertion(ce)
                   /*Generate exact region assertion for s_j*/
14                 if RegionFullMatched(cer, dom) then
15                     s_j.AddRegFullAssertion(cer)
16                 else if RegionTagAttMatched(cer, dom)
                   then
17                     s_j.AddRegTagAttAssertion(cer)
18                 else if RegionTagMatched(cer, dom) then
19                     s_j.AddRegTagAssertion(cer)
               /* Generate similar region assertions for s_i */
20             for be ∈ s_i.GetBlockElements() do
21                 if Predict(be) == 1 then
22                     s_i.AddRegTagAttAssertion(be.GetRegion())
```

sidered important for testing; for example, a banner section or decoration parts of a page might not be as important as an inner content that changes according to a main functionality, (2) patterns in the page that might be part of a template. Therefore, extracting patterns from existing assertions may help us in generating new but similar assertions.

We formally define a DOM-based assertion as a function $\mathcal{A} : (s, c) \rightarrow \{True, False\}$, where s is a DOM state, and c is a DOM condition to be checked. It returns *True* if s matches/satisfies the condition c, denoted by $s \models c$, and *False* otherwise. We say that an assertion A subsumes (implies) assertion B, denoted by $A \Longrightarrow B$, if $A \rightarrow True$, then $B \rightarrow True$. This means that B can be obtained from A by weakening A's condition. In this case, A is more specific/-constrained than B. For instance, an assertion verifying the existence of a checked element `<span id="mainContent">` can be implied by an assertion which verifies both the existence of that element and its attributes/textual values.

Algorithm 2 shows our assertion regeneration procedure. We consider each manual-test state s_i (Definition 3) in the SFG and try to reuse existing associated assertions in s_i or generate new ones based on them for another state s_j. We extend the set of DOM-based assertions in three forms: (1) reusing the *same* assertions from manual-test states for states without such assertions, (2) regenerating assertions with the *exact* assertion pattern structure as the original assertions but adapted for another state, and (3) learning structures from the original assertions to generate *similar* assertions for other states.

3.3.1 Assertion Reuse

As an example for the assertion reuse, consider Figure 3 and the manual-test path with the sequence of states $\langle Index, s1, s2, s3, s4, s5\rangle$ for adding a note. Assertions in Figure 2 line 7 and 12 are associated to states $s1$ and $s4$, respectively. Suppose that we explore an alternative path for deleting a note with the sequence $\langle Index, s1, s2, s10, s4, s5\rangle$, which was not originally considered by the developer. Since the two test paths share a common path from $Index$ to $s1$, the assertion on $s1$ can be reused for the new test case (note deletion) as well. This is a simple form of assertion reuse on new test paths.

Table 1: Summary of the assertion reuse/regeneration conditions for an element e_j on a DOM state s_j, given a checked element e_i on state s_i.

Condition	Description
ElementFullMatched	$Tag(e_i)$=$Tag(e_j) \wedge Att(e_i)$=$Att(e_j) \wedge$ $Txt(e_i)$=$Txt(e_j)$
ElementTagAttMatched	$Tag(e_i)$=$Tag(e_j) \wedge Att(e_i)$=$Att(e_j)$
RegionFullMatched	$Tag(R(e_i, s_i))$=$Tag(R(e_j, s_j)) \wedge$ $Att(R(e_i, s_i))$=$Att(R(e_j, s_j)) \wedge$ $Txt(R(e_i, s_i))$=$Txt(R(e_j, s_j))$
RegionTagAttMatched	$Tag(R(e_i, s_i))$=$Tag(R(e_j, s_j)) \wedge$ $Att(R(e_i, s_i))$=$Att(R(e_j, s_j))$
RegionTagMatched	$Tag(R(e_i, s_i))$=$Tag(R(e_j, s_j))$

3.3.2 Assertion Regeneration

We regenerate two types of precondition assertions namely *exact element-based assertions*, and *exact region-based assertions*. By "exact" we mean repetition of the same structure of an original assertion on a checked element.

The rationale behind our technique is to use the location and properties of checked elements and their close-by neighbourhood in the DOM tree to regenerate assertions, which focus on the exact repeated structures and patterns in other DOM states. This approach is based on our intuition that checking the close-by neighbour of checked elements is just as important.

Exact element assertion generation. We define assertions of the form $\mathcal{A}(s_j, c(e_j))$ with a condition $c(e_j)$ for element e_j on state s_j. Given an existing checked element (Definition 5) e_i on a DOM state s_i, we consider 2 conditions as follows:

1. *ElementFullMatched*: If a DOM state s_j contains an element with exact tag, attributes, and text value as e_i, then *reuse* assertion on e_i for checking e_j on s_j.
2. *ElementTagAttMatched*: If a DOM state s_j contains an element e_j with exact tag and attributes, but different text value as e_i, then *generate* assertion on e_j for checking its tag and attributes.

Table 1 summarizes these conditions. An example of a *generated* assertion is `assertTrue(isElementPresent(By.id("mainContent")))` which checks the existence of a checked element with ID `"mainContent"`. Such an assertion can be evaluated in any state in the SFG that contains that DOM element (and thus meets the precondition). Note that we could also propose assertions in case of mere tag matches, however, such assertions are not generally considered useful as they are too generic.

Exact region assertion generation. We define the term *checked element region* to refer to a close-by area around a checked element:

Definition 7 (Checked Element Region). *For a checked element e on state s, a checked element region $\mathcal{R}(e, s)$, is a function $\mathcal{R} : (e, s) \rightarrow \{e, \mathcal{P}(e), \mathcal{C}h(e)\}$, where $\mathcal{P}(e)$ and $\mathcal{C}h(e)$ are the parent node, and children nodes of e respectively.* □

For example, for the element e = `<span id="mainContent">` (Figure 1), which is in fact a checked

element in line 12 of Figure 2 (at state $s4$ in Figure 3), we have $\mathcal{R}(e, s4) = \{e, \mathcal{P}(e), \mathcal{C}h(e)\}$, where $\mathcal{P}(e)$=`<td class="cssMain" valign="top">`, and $\mathcal{C}h(e)$={`<img src="img/head_notes.gif">`, `<p>`, `<input id="ok" src="img/ok0.gif">`}.

We define assertions of the form $\mathcal{A}(s_j, c(\mathcal{R}(e_j, s_j)))$ with a condition $c(\mathcal{R}(e_j, s_j))$ for the region $\mathcal{R}$ of an element e_j on state s_j. Given an existing checked element e_i on a DOM state s_i, we consider 3 conditions as follows:

1. *RegionFullMatched*: If a DOM state s_j contains an element e_j with exact tag, attributes, and text values of $\mathcal{R}(e_j, s_j)$ as $\mathcal{R}(e_i, s_i)$, then *generate* assertion on $\mathcal{R}(e_j, s_j)$ for checking its tag, attributes, and text values.
2. *RegionTagAttMatched*: If a DOM state s_j contains an element e_j with exact tag, and attributes values of $\mathcal{R}(e_j, s_j)$ as $\mathcal{R}(e_i, s_i)$, then *generate* assertion on $\mathcal{R}(e_j, s_j)$ for checking its tag and attributes values.
3. *RegionTagMatched*: If a DOM state s_j contains an element e_j with exact tag value of $\mathcal{R}(e_j, s_j)$ as $\mathcal{R}(e_i, s_i)$, then *generate* assertion on $\mathcal{R}(e_j, s_j)$ for checking its tag value.

Note that the assertion conditions are relaxed one after another. In other words, on a DOM state s, if $s \models RegionFullMatched$, then $s \models RegionTagAttMatched$; and if $s \models RegionTagAttMatched$, then we have $s \models RegionTagMatched$. Consequently it suffices to use the most constrained assertion. We use this property for reducing the number of generated assertions in subsubsection 3.3.4.

Table 1 summarizes these conditions. Assertions that we generate for a checked element region, are targeted around a checked element. For instance, to check if a DOM state contains a checked element region with its tag, attributes, and text values, an assertion will be *generated* in the form of `assertTrue(isElementRegionFullPresent(parentElement, element, childrenElements))`, where `parentElement`, `element`, and `childrenElements` are objects reflecting information about that region on the DOM.

For each checked element *ce* on s_i, we also generate a `RegionFull` type of assertion for checking its region, i.e., verifying *RegionFullMatched* condition on s_i (Algorithm 2 line 5). Lines 10–13 perform exact element assertion generation. The original assertion can be reused in case of *ElementFullMatched* (line 11). Lines 14–19 apply exact region assertion generation based on the observed matching. Notice the hierarchical selection which guarantees generation of more specific assertions.

3.3.3 Learning Assertions for Similar Regions

The described exact element/region assertion regeneration techniques only consider the exact repetition of a checked element/region. However, there might be many other DOM elements that are similar to the checked elements but not exactly the same. For instance, consider Figure 2 line 12 in which a `<span id="mainContent">` element was checked in an assertion. If in another state, a `<div id="centreDiv">` element exists, which is similar to the `<span>` element in certain aspects such as content and position on the page, we could generate a DOM-based assertion for the `<div>` element in the form of `assertTrue(isElementPresent(By.id("centreDiv")));`.

We view the problem of generating similar assertions as a classification problem which decides whether a block level DOM element is *important to be checked* by an assertion or not. To this end, we apply machine learning to train a classifier based on the features of the checked elements in existing assertions. More specifically, given a training dataset $\mathcal{D}$ of n DOM elements in the form $\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^{n}$, where each $\mathbf{x}_i$ is a p-dimensional real vector representing the features of a DOM element e_i, and y_i indicates whether e_i is a checked element (+1) or not (−1), the classification function $\mathcal{F} : \mathbf{x_j} \rightarrow y_i$ maps a feature vector $\mathbf{x}_j$ to its class label y_j. To do so, we use Support Vector Machine (SVM) [32] to find the max-margin hyperplane that divides the elements with $y_i = 1$ from those with $y_i = -1$. In the rest of this subsection, we describe our used features, how to label the feature vectors, and how to generate similar region DOM-based assertions.

DOM element features. We present a set of features for a DOM element to be used in our classification task. A feature extraction function $\psi : e \rightarrow \mathbf{x}$ maps an element e to its feature set $\mathbf{x}$. Many of these features are based on and adapted from the work in [29], which performs page segmentation ranking for adaptation purpose. The work presented a number of spatial and content features that capture the importance of a webpage segment based on a comprehensive user study. Although they targeted a different problem than ours, we gained insight from their empirical work and use that to reason about the importance of a page segment for testing purposes. Our proposed DOM features are presented in Table 2. We normalize feature values between [0–1] as explained in Table 2, to be used in the learning phase. For example, consider the element e = `<span>` in Figure 1, then $\psi(e)$ = <0.5, 0.7, 0.6, 0.5, 1, 0.2, 0, 0.3> corresponding to features `BlockCenterX`, `BlockCenterY`, `BlockWidth`, `BlockHeight`, `TextImportance`, `InnerHtmlLength`, `LinkNum`, and `ChildrenNum`, respectively.

Labelling the feature vectors. For the training phase, we need a dataset of feature vectors for DOM elements annotated with +1 (important to be checked in assertion) and -1 (not important for testing) labels. After generating a feature vector for each "checked DOM element", we label it by +1. For some elements with label -1, we consider those with "most frequent features" over all the manual-test states. Unlike previous work that focuses on DOM invariants [25], our insight is that DOM subtrees that are invariant across manual-test states, are less important to be checked in assertions. In fact, most modern web applications execute a significant amount of client-side code in the browser to mutate the DOM at runtime; hence DOM elements that remain unchanged across application execution are more likely to be related to fixed (server-side) HTML templates. Consequently, such elements are less likely to contain functionality errors. Thus, for our feature vectors we consider all block elements (such as `div`, `span`, `table`) on the manual-test states and rank them in a decreasing order based on their occurrences. In order to have a balanced dataset of items belonging to {-1,+1}, we select the k-top ranked (i.e., k most frequent) elements with label -1, were k equals the number of label +1 samples.

Predicting new DOM elements. Once the SVM is trained on the dataset, it is used to predict whether a given DOM element should be checked in an assertion (algorithm 2, Lines 20–23). If the condition $\mathcal{F}(\psi : e \rightarrow \mathbf{x})$=1 holds, we generate a `RegionTagAtt` type assertion (i.e., checking tag and attributes of a region). We do not consider a `RegionFull` (i.e., checking tag, attributes, and text of a region) assertion type in this case because we are dealing with a *similar* detected region, not an exact one. Also, we do not generate a `RegionTag` assertion type because a

Table 2: DOM element features used to train a classifier.

Feature Name	Definition	Rationale
ElementCenterX, ElementCenterY	The (x,y) coordinates of the centre of a DOM element. BlockCenterX and BlockCenterY are normalized by dividing by PageWidth and PageHeight (i.e., the width and height of the whole page) respectively.	Web designers typically put the most important information (main content) in the centre of the page, the navigation bar on the header or on the left side, and the copyright on the footer [29]. Thus, if the (x,y) coordinate of the centre of a DOM block is close to the (x,y) coordinate of the web page centre, that block is more likely to be part of the main content.
ElementWidth, ElementHeight	These are the width and height of the DOM element, which are also normalized by dividing by PageWidth and PageHeight, respectively.	The width and height of an element can be an indication for an important segment. Intuitively, large blocks typically contain much irrelevant noisy content [29].
TextImportance	This binary value feature indicates whether the block element contains any visually important text.	Text in bold/italic style, or header elements (such as h1, h2,..., h5) to highlight and emphasize textual content usually imply importance in that region.
InnerHtmlLength	The innerHtmlLength is the length of all HTML code string (without whitespace) in the element block. We normalize this value by dividing it by InnerHtmlLength of the whole page.	The normalized feature value can indicate the block content size. Intuitively, blocks with many sub-blocks and elements are considered to be less important than those with fewer but more specific content [29].
LinkNum	The LinkNum is the number of anchor (hyperlink) elements inside the DOM element and is normalized by the link number of the whole page.	If a DOM region contains clickables, it is likely part of a navigational structure (menu) and not part of the main content [29].
ChildrenNum	The ChildrenNum is the number of child nodes under a DOM node. We normalize this value by dividing it by a constant number (10 in our implementation) and setting the normalized value to 1 if it exceeds 1.	We have observed in many DOM-based test cases that checked elements do not have a large number of children nodes. Therefore, this feature can be used to discourage elements with many children to be selected for a region assertion, to enhance test readability.

higher priority should be given to the similar region-based assertions.

3.3.4 Assertion Minimization

The proposed assertion regeneration technique can generate many DOM-based assertions per state, which in turn can make the generated test method hard to comprehend and maintain. Therefore, we (1) avoid generating redundant assertions, and (2) prioritize assertions based on their constraints and effectiveness.

Avoiding redundant assertions. A new reused/generated assertion for a state (Algorithm 2, lines 5, 11, 13, 15, 17, 19, and 22), might already be subsumed by, or may subsume other assertions, in that state. For example an exact element assertion which verifies the existence of a checked element `<span id="mainContent">` can be subsumed by an exact region assertion which has the same `span` element in either its checked element, parent, or its children nodes. Assertions that are subsumed by other assertions are redundant and safely eliminated to reduce the overhead in testing time and increase the readability and maintainability of test cases. For a given state s with an existing assertion B, a new assertion A generated for s is treated as follows:

$$\begin{cases} \text{Discard } A & \text{; if } B \Longrightarrow A \\ \text{Replace } B \text{ with } A & \text{; if } A \Longrightarrow B \wedge B \notin \text{original assertions} \\ \text{Add } A \text{ to } s & \text{; otherwise} \end{cases}$$

Prioritizing assertions. We prioritize the generated assertions such that given a maximum number of assertions to produce per state, the more effective ones are ranked higher and chosen. We prioritize assertions in each state in the following order; the highest priority is given to the original human-written assertions. Next are the reused, the `RegionFull`, the `RegionTagAtt`, the `ElementTagAtt`, and the `RegionAtt` assertions. This ordering gives higher priorities to more specific/constrained assertions first.

3.4 Test Suite Generation

In the final step, we generate a test suite from the extended state-flow graph. Each path from the *Index* node to a sink node (i.e., node without outgoing edges) in the SFG is transformed into a unit test. Loops are included once. Each test case captures the sequence of events as well as any assertions for the target states. To make the test case more readable for the developers, information (such as tag name and attributes) about related DOM elements is generated as code comments.

After generating the extended test suite, we make sure that the reused/regenerated assertions are stable, i.e., do not falsely fail, when running the test suite on an unmodified version of the web application. Some of these assertions are not only DOM related but also depend on the specific path through which the DOM state is reached. Our technique automatically identifies and filters these false positive cases from the generated test suite. This is done through executing the generated test suite and eliminating failing assertions form the test cases iteratively, until all tests pass successfully.

4. IMPLEMENTATION

The approach is implemented in a tool, called Testilizer, which is publicly available [7]. The state exploration component is built on top of Crawljax [18]. Testilizer requires as input the source code of the human-written test suite and the URL of the web application. Testilizer currently supports Selenium tests, however, our approach can be easily applied to other DOM-based tests as well.

To instrument the test cases, we use JavaParser [2] to get a abstract syntax tree. We instrument all DOM related method calls and calls with arguments that have DOM element locaters. We also log the DOM state after every event in the tests, capable of changing the DOM. For the state abstraction function (as defined in Definition 1), we generate an abstract DOM state by ignoring recurring structures (patterns such as table rows and list items), textual content (such as ignoring the text node "Note has been created" in the partial DOM shown in Figure 1), and contents in the `<script>` tags. For the classification step, we use LIBSVM [12], which is a popular library for support vector machines.

5. EMPIRICAL EVALUATION

To assess the efficacy of our proposed technique, we have conducted a controlled experiment to address the following research questions:

RQ1 How much of the information (input data, event sequences, and assertions) in the original human-written test suite is leveraged by Testilizer?

RQ2 How successful is Testilizer in regenerating effective assertions?

Table 3: Experimental objects.

Name	SLOC	#Test Methods	#Assertions
Claroline e-learning (1.11.7)	PHP (295K) JS (36K)	23	35
PhotoGallery (3.31)	PHP (5.6K) JS (1.5K)	7	18
WolfCMS (0.7.8)	PHP (35K) JS (1.3K)	12	42
EnterpriseStore (1.0.0)	Java (3K) JS (57K)	19	17

RQ3 Does TESTILIZER improve coverage?

Our experimental data along with the implementation of TESTILIZER are available for download [7].

5.1 Experimental Objects

We selected four open source web applications that make extensive use of client-side JavaScript, fall under different application domains, and have Selenium test cases. The experimental objects and their properties are shown in Table 3. *Claroline* [1] is a collaborative e-learning environment, which allows instructors to create and administer courses. *Phormer* [5] is a photo gallery equipped with upload, comment, rate, and slideshow functionalities. *WolfCMS* [8] is a content management system. *EnterpriseStore* [9] is an enterprise asset management web application.

5.2 Experimental Setup

Our experiments are performed on Mac OS X, running on a 2.3GHz Intel Core i7 CPU with 8 GB memory, and FireFox 28.0.

5.2.1 Independent Variables

We compare the original human-written test suites with the test suites generated by TESTILIZER.

Test suite generation method. We evaluate different test suite generation methods for each application as presented in Table 4. We compare TESTILIZER (EXND+AR) with three baselines, (1) ORIG: original human-written test suite, (2) EXND+RND: test suite generated by traversing the extended SFG, equipped with random assertion generation, and (3) RAND+RND: random exploration and random assertion generation. In random assertion generation, for each state we generate element/region assertions by randomly selecting from a pool of DOM-based assertions. These random assertions are based on the existence of an element/region in a DOM state. Such assertions are expected to pass as long as the application is not modified. However, due to our state abstraction this can result in unstable assertions, which are also automatically eliminated following the approach explained in subsection 3.4.

We further evaluate various instantiations of our assertion generation in EXND+AR, i.e., using only (a) original assertions, (b) reused assertions (Section 3.3.1), (c) exact generated (Section 3.3.2), (d) similar region generated (Section 3.3.3), and (e) a combination of all these types.

Exploration constraints. We confine the exploration time to five minutes in all the experiments, which should be acceptable in most testing environments. Suppose in the EXND approach, TESTILIZER spends time t to generating the initial SFG for an application. To make a fair comparison, we add this time t to the five minutes for the RAND exploration approach. We set no limits on the crawling depth nor the maximum number of states to be discovered while looking for alternative paths. Note that for both EXND and RAND crawling, after a clickable element on a state was exercised, the crawler resets to the index page and continues crawling from another chosen state.

Table 4: Test suite generation methods evaluated.

Test Suite Generation Method	Action Sequence Generation Method	Assertion Generation Method
ORIG	Manual	Manual
TESTILIZER (EXND+AR)	Traversing paths in the extended SFG generated from the original tests	Assertion regeneration
EXND+RND	Traversing paths in the extended SFG generated from the original tests	Random
RAND+RND	Traversing paths in the SFG generated by random crawling	Random

Maximum number of generated assertions. We constrain the maximum number of generated assertions for each state to five. To have a fair comparison, for the EXND+RND and RAND+RND methods, we perform the same assertion prioritization used in TESTILIZER and select the top ranked.

Learning parameters. We set the SVM's kernel function to the Gaussian RBF, and use 5-fold cross-validation for tuning the model and feature selection.

5.2.2 Dependent Variables

Original coverage. To assess how much of the information including input data, event sequences, and assertions of the original test suite is leveraged (RQ1), we measure the state and transition coverage of the initial SFG (i.e., SFG mined from the original test cases). We also measure how much of the unique assertions and unique input data in the original test cases has been utilized.

Fault detection rate. To answer RQ2 (assertions effectiveness), we evaluate the DOM-based fault detection capability of TESTILIZER through automated first-order mutation analysis. The test suites are evaluated based on the number of detected mutants by test assertions.

We apply the DOM, jQuery, and XHR mutation operators at the JavaScript code level as described in [21], which are based on a study of common mistakes made by web developers. Examples include changing the ID/tag name used in `getElementById` and `getElementByTagName` methods, changing the attribute name/value in `setAttribute`, `getAttribute` and `removeAttribute` methods, removing the $ sign that returns a jQuery object, changing the name of the property/class/element in the `addClass`, `removeClass`, `removeAttr`, `remove`, `attr`, and `css` methods in jQuery, swapping `innerHTML` and `innerText` properties, and modifying the XHR type (`Get`/`Post`). On average we generate 36 mutant versions for each application.

Code coverage. Code coverage has been commonly used as an indicator of the quality of a test suite by identifying under-tested parts, while it does not directly imply the effectiveness of a test suite [16]. Although TESTILIZER does not target code coverage maximization, to address RQ3, we compare the JavaScript code coverage of the different test suites using JSCover [3].

5.3 Results

Original SFG Coverage (RQ1). Table 5 shows the average results of our experiments. As expected, the number of states, transitions, and generated test cases are higher in TESTILIZER. The random exploration (RAND) on average generates fewer states and transitions, but more test cases

Table 5: Results showing statistics of the test models and original test suite information usage, average over experimental objects.

Test Suite	# States	# Transitions	# Test Cases	Orig State Coverage	Orig Transition Coverage	Orig Input Data Usage	Orig Assertion Usage	JS Code Coverages
ORIG	37	46	15	100%	100%	100%	100%	20%
EXND	54	63	47	98%	96%	100%	100%	26%
RAND	33	40	25	65%	60%	0%	0%	22%

compared to the original test suite. This is mainly due to the fact that in the SFG generated by RAND, there are more paths from `Index` to the sink nodes than in the SFG mined from the original test suite.

Regarding the usage of original test suite information (RQ1), as expected TESTILIZER, which leverages the event sequences and inputs of the original test suite, has almost full state (98%) and transition (96%) coverage of the initial model. The few cases missed are due to the traversal algorithm we used, which has limitations on dealing with cycles in the graph that do not end with a sink node and thus are not generated. Note that we can select the missing cases from the original manual-written test suite and add them to the generated test suite. By analyzing the generated test suites, we found that on average, TESTILIZER reused 22 input values (in addition to the login data) from the average of 15 original inputs. The RAND exploration approach covered about 60% of the states and transitions, without any usage of input data (apart from the login data, which was provided to RAND manually).

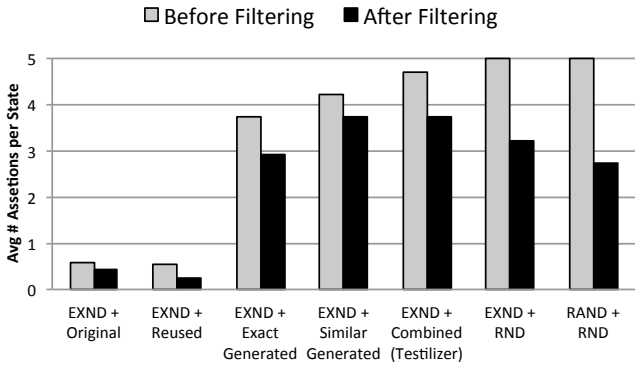

Figure 5: Average number of assertions per state, before and after filtering unstable assertions.

Figure 5 presents the average number of assertions per state before and after filtering the unstable ones. The difference between the number of actual generated assertions and the stable ones reveals that our generated assertions (combined, similar/exact generated) are more stable than the random approach. The reduction percentage is 25%, 49%, 22%, 11%, 20%, 35%, and 45% for the original, reused, exact generated, similar generated, combined (TESTILIZER), EXND+RND and RAND+RND, respectively.

A major source of this instability is the selection of dynamic DOM elements in the generated assertions. For instance, RND (random assertion generation) selects many DOM elements with dynamic time-based attributes. Also the more restricted an assertion is, the less likely it is to remain stable in different paths. This is the case for some of the (1) reused assertions that replicate the original assertions and (2) exact generated ones specially `FullRegionMatchs` type. On the other hand, learned assertions are less strict (e.g., `AttTagRegionMatchs`) and are thus more stable.

Overall, the test suite generated by TESTILIZER, on average, consists of 12% original assertions, 11% reused assertions, 31% exact generated assertions, and 45% of similar learned assertions.

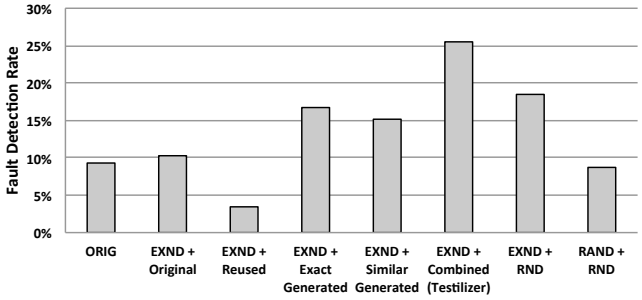

Figure 6: Comparison of average fault detection rate using different test suite generation methods.

Fault detection (RQ2). Figure 6 depicts a comparison of fault detection rates for the different methods. Figure 6 shows that exact and similar generated assertions are more effective than original and reused ones. The effectiveness of each assertion generation technique solely is not more than the random approach. This is mainly due to the fact that the number of random assertions per state is more than the assertions reused/generated by TESTILIZER, since we always select 5 random assertions at each state from a pool of assertions but not always find 5 exact/similar match in a state.

More importantly, the results show that TESTILIZER outperforms fault detection capability of the original test suite by 150% (15% increase) and the random methods by 37% (7% increase). This supports our insight that leveraging input values and assertions from human-written test suites can be helpful in generating more effective test cases.

Code Coverage (RQ3). Although code coverage improvement is not the main goal of TESTILIZER in this work, the generated test suite has a slightly higher code coverage. As shown in Table 5, there is a 30% improvement (6% increase) over the original test suite and 18% improvement (4% increase) over the RAND test suite. Note that the original test suites were already equipped with proper input data, but not many execution paths (thus the slight increase). On the other hand, the random exploration considered more paths in a blind search, but without proper input data.

5.4 Discussion

Test case dependencies. An assumption made in TESTILIZER is that the original test suite does not have any test case dependencies. Generally, test cases should be executable without any special order or dependency on previous tests. However, while conducting our evaluation, we came across multiple test suites that violated this principle. For such cases, although TESTILIZER can generate test cases, failures can occur due to these dependencies.

Effectiveness. The effectiveness of the generated test suite depends on multiple factors. First, the size and the quality of the original test suite is very important; if the original test suite does not contain paths with effective assertions, it is not possible to generate an effective extended test suite. In the future we plan to use other adequacy metrics, such as DOM coverage [22], to measure the quality of a given test suite. Second, the learning-based approach can be tuned in various ways (e.g., selecting other features, changing the SVM parameters, and choosing sample dataset size) to obtain better results. Third, the size of the DOM subtree (region) to be checked can be increased to detect changes more effectively, however, it might come at the cost of making the test suite more brittle.

Efficiency. The larger a test suite, the more time it takes to test an application. Since in many testing environments time is limited, not all possible paths of events should be generated in the extended test suite. The challenge is finding a balance between effectiveness and efficiency of the test cases. The current graph traversal method in TESTILIZER may produce test cases that share common paths, which do not contribute much to fault detection or code coverage. An optimization could be realized by guiding the test generation algorithm towards states that have more constrained DOM-based assertions.

Threats to validity. Although SELENIUM is widely used in industry for testing commercial web applications, unfortunately, very few open source web applications are publicly available that have (working) SELENIUM test suites. Therefore, we were able to include a limited number of applications in our study. A threat to the external validity of our experiment is with regard to the generalization of the results to other web applications. To mitigate this threat, however, we selected our experimental objects from different domains with variations in functionality and structure. With respect to reproducibility of our results, TESTILIZER, the test suites, and the experimental objects are publicly available, making the experiment reproducible.

6. RELATED WORK

Elbaum et al. [15] leverage user-sessions for web application test generation. Based on this work, Sprenkle et al. [30] propose a tool to generate additional test cases based on the captured user-session data. McAllister et al. [17] leverage user interactions for web testing. Their method relies on prerecorded traces of user interactions and requires instrumenting one specific web application framework. None of these techniques considers leveraging knowledge from existing test cases as TESTILIZER does.

Xie and Notkin [34] infer a model of the application under test by executing the existing test cases. Dallmeier et al. [14] mine a specification of desktop systems by executing the test cases. Schur et al. [28] infer behaviour models from enterprise web applications via crawling. Their tool generates test cases simulating possible user inputs. Similarly, Xu et al. [35] mine executable specifications of web applications from SELENIUM test cases to create an abstraction of the system. Yuan and Memon [39] propose an approach to iteratively rerun automatically generated test cases for generating alternating test cases. This is inline with feedback-directed testing [24], which leverages dynamic data produced by executing the program using previously generated test cases. For instance, Artemis [11] is a feedback-directed tool for automated testing of JavaScript applications that uses generic oracles such as HTML validation. Our previous work, FeedEx [20], applies a feedback-directed exploration technique to guide the exploration at runtime towards more coverage and higher navigational and structural diversity. These approaches, however, do not use information in existing test cases, and they do not address the problem of test oracle generation.

Yoo and Harman [37] propose a search-based approach to reuse and regenerate existing test data for primitive data types. They show that the knowledge of existing test data can help to improve the quality of new generated test data. Alshahwan and Harman [10] generate new sequences of HTTP requests through a def-use analysis of server-side code. Pezze et al. [26] present a technique to generate integration test cases from existing unit test cases. Mirzaaghaei et al. [23] use test adaptation patterns in existing test cases to support test suite evolution.

This work is also related to test suite augmentation techniques [36, 27] used in regression testing. In test suite augmentation the goal is to generate new test cases for the changed parts of the application. More related to our work is [33], which aggregates tests generated by different approaches using a unified test case language. They propose a test advice framework that extracts information in the existing tests to help improve other tests or test generation techniques.

A generic approach used often as a test oracle is checking for thrown exceptions and application crashes [38]. This is, however, not very helpful for web applications as they do not crash easily and the browser continues the execution even after exceptions. Current web testing techniques simplify the test oracle problem in the generated test cases by using soft oracles, such as generic user-defined oracles, and HTML validation [19, 11].

Our work is different from these approaches in that we (1) reuse knowledge in existing human-written test cases in the context of web application testing, (2) reuse input values and event sequences in test cases to explore alternative paths and news states of web application, and (3) reuse oracles of the test cases for regenerating assertions to improve the fault finding capability of the test suite.

7. CONCLUSIONS AND FUTURE WORK

This work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be used to tackle some of the challenges in automated web application test generation. Given a web application and its DOM-based (such as SELENIUM) test suite, our tool, called TESTILIZER, utilizes the given test suite to generate effective test cases by exploring alternative paths of the application, and regenerating assertions for new detected states. Our empirical results on four real-world applications show that TESTILIZER easily outperforms a random test generation technique, provides substantial improvements in the fault detection rate compared with the original test suite, while slightly increasing code coverage too.

For future work, we plan to evaluate the effectiveness of other state space exploring strategies, e.g., diversification of test-paths, and investigate correlations between the effectiveness of the original test suite and the generated test suite.

8. ACKNOWLEDGMENTS

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, and Swiss National Science Foundation (PBTIP2145663).

9. REFERENCES

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

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

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

[4] Organizer. http://www.apress.com/9781590596951.

[5] Phormer Photogallery. http://sourceforge.net/projects/rephormer/.

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

[7] Testilizer. http://salt.ece.ubc.ca/software/testilizer.

[8] WolfCMS. https://github.com/wolfcms/wolfcms.

[9] WSO2 EnterpriseStore. https://github.com/wso2/enterprise-store.

[10] N. Alshahwan and M. Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 45–55, 2012.

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

[12] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[13] 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.

[14] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96, 2010.

[15] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

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

[17] S. McAllister, E. Kirda, and C. Kruegel. Leveraging user interactions for in-depth testing of web applications. In *Recent Advances in Intrusion Detection*, volume 5230 of *LNCS*, pages 191–210. Springer, 2008.

[18] 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.

[19] 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.

[20] A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 278–287. IEEE Computer Society, 2013.

[21] 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.

[22] M. Mirzaaghaei and A. Mesbah. DOM-based test adequacy criteria for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–81. ACM, 2014.

[23] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240. IEEE Computer Society, 2012.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE Computer Society, 2007.

[25] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for web 2.0 application robustness testing. In *Proceedings of the International Symposium on Sw. Reliability Eng. (ISSRE)*, pages 191–200. IEEE Computer Society, 2010.

[26] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 11–20. IEEE, 2013.

[27] K. Rubinov and J. Wuttke. Augmenting test suites automatically. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1433–1434, Piscataway, NJ, USA, 2012. IEEE Press.

[28] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Proceedings of the Foundations of Software Engineering (ESEC/FSE), pages 422–432. ACM, 2013.

[29] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning important models for web page blocks based on layout and content analysis. *ACM SIGKDD Explorations Newsletter*, 6(2):14–23, 2004.

[30] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 253–262. ACM, 2005.

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

[32] V. Vapnik. *The nature of statistical learning theory*. springer, 2000.

[33] Y. Wang, S. Person, S. Elbaum, and M. B. Dwyer. A framework to advise tests using tests. In *Proc. of ICSE NIER*. ACM, 2014.

[34] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Formal Approaches to Software Testing*, pages 60–69. Springer, 2004.

[35] D. Xu, W. Xu, B. K. Bavikati, and W. E. Wong. Mining executable specifications of web applications from selenium ide tests. In *Software Security and*

Reliability (SERE), 2012 IEEE Sixth International Conference on, pages 263–272. IEEE, 2012.

[36] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 257–266. ACM, 2010.

[37] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012.

[38] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.

[39] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559–575, 2010.