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. 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 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 or standard validators 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
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, $<D,Q,\Omega,\delta>$, where $D$ is the set of vertices, $Q$ is the set of directed edges, $\alpha \in D$ is the root vertex, $\Omega$ is a finite set of labels and $\delta : D \rightarrow \Omega$ is a labelling function that assigns a label from $\Omega$ 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 $\delta$, 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.](image)

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

**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-
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 that 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 (Index, s1, s2, s10, s4, s5) 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 in 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 ∈ T.

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 ∈ v is an element in the DOM tree in state v, whose existence, value, or attributes are checked in an assertion of a test case t ∈ 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 3 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 css-selector) as well as the access values and innerHTML text.
Algorithm 1: State-Flow Graph Inference

\begin{algorithm}
\SetAlgoLined
\DontPrintSemicolon
\KwIn{A Web application \texttt{URL}, a DOM-based test suite \texttt{TS}, crawling constraints \texttt{CC}}
\KwOut{A state-flow graph \texttt{SFG}}
\SetKwProg{Fn}{Procedure}{}{end}
\Fn{InferSFG(\texttt{URL}, \texttt{TS}, \texttt{CC})}{\begin{itemize}
\item \texttt{TS}_{init} \leftarrow \text{Instrument(\texttt{TS})} \\
\item \texttt{TOP} \leftarrow \text{ReadTestOperationDataset()}
\item \texttt{dom} \leftarrow \text{browser.GetDOM()}
\item \texttt{SFG}_{init} \leftarrow \phi
\item for top \in \texttt{TOP} do
\item \hspace{1em} \texttt{C} \leftarrow \text{GetClickables(top)}
\item \hspace{1em} for \texttt{c} \in \texttt{C} do
\item \hspace{1em} \hspace{1em} \texttt{assertion} \leftarrow \text{GetAssertion(top)}
\item \hspace{1em} \hspace{1em} \texttt{dom} \leftarrow \text{browser.GetDOM()}
\item \hspace{1em} \hspace{1em} \texttt{robot.FireEvent(c)}
\item \hspace{1em} \hspace{1em} new\_\texttt{dom} \leftarrow \text{browser.GetDOM()}
\item \hspace{1em} if \texttt{dom.HasChanged(new\_\texttt{dom})} then
\item \hspace{1em} \hspace{1em} \texttt{SFG}_{init}.\text{Update}(c, new\_\texttt{dom}, \text{assertion})
\item \hspace{1em} \hspace{1em} \texttt{browser.GoTo(URL)}
\item \hspace{1em} \hspace{1em} \texttt{SFG}_{init} \leftarrow \text{InferSFG(\texttt{URL}, \texttt{TS}, \texttt{CC})}
\item \texttt{SFG}_{ext} \leftarrow \text{InferSFG(\texttt{TS}, \texttt{CC})}
\item \texttt{ExploreAlternativePaths(\texttt{SFG}_{init}, \texttt{CC})}
\item \texttt{Procedure ExploreAlternativePaths(\texttt{SFG}, \texttt{CC})}{\begin{itemize}
\item if \text{ConstraintSatisfied(\texttt{CC})} do
\item \hspace{1em} \texttt{S} \leftarrow \text{GetNextToExploreState(\texttt{SFG}, \texttt{CC})}
\item \hspace{1em} \texttt{C} \leftarrow \text{GetCandidateClickables(\texttt{s})}
\item \hspace{1em} for \texttt{c} \in \texttt{C} do
\item \hspace{1em} \hspace{1em} \texttt{browser.GoTo(SFG.GetPath(\texttt{s}))}
\item \hspace{1em} \hspace{1em} \texttt{dom} \leftarrow \text{browser.GetDOM()}
\item \hspace{1em} \hspace{1em} \texttt{robot.FireEvent(c)}
\item \hspace{1em} \hspace{1em} \texttt{new\_\texttt{dom}} \leftarrow \text{browser.GetDOM()}
\item \hspace{1em} if \texttt{dom.HasChanged(new\_\texttt{dom})} then
\item \hspace{1em} \hspace{1em} \texttt{SFG}.\text{Update}(c, new\_\texttt{dom}, \text{assertion})
\item \hspace{1em} \hspace{1em} \texttt{ExploreAlternativePaths(SFG, \texttt{CC})}
\end{itemize}}
\end{itemize}}
\end{algorithm}

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) \cite{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 \texttt{SFG} for a web application \texttt{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 \(W\) 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 \(DS\) of \(W\), with a labeling function \(\Phi : \mathcal{V} \rightarrow A\) that assigns a label from \(A\) to each vertex in \(\mathcal{V}\), where \(A\) 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 labeling function that assigns a label, from a set of event types and DOM element properties, to each edge.
5. \texttt{SFG} can have multi-edges and be cyclic. \(\square\)

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) \cite{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 \texttt{init} 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).

3.3 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 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 \text{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); \text{GetNextToExploreState} 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 \text{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., \texttt{s2}, navigating the application to reach to that state from the \texttt{Index} state, and firing an event on \texttt{s2} resulting in a new state \texttt{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 choose 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-
c is a DOM condition to be checked. It returns A(2) patterns in the page that might be part of a template. otherwise. We say that an assertion False existence of a checked element
\[ s \rightarrow \langle 3 \]
and the manual-test path with the sequence of states (1) reusing the We extend the set of DOM-based assertions in three forms: (2) patterns in the page that might be part of a template.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states (1) reusing the
We extend the set of DOM-based assertions in three forms:
(2) regenerating assertions
with the sequence of states
(3) learning structures from the original assertions to generate similar assertions for other states.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.

\[ A \rightarrow \langle 3 \]
and the manual-test path with the sequence of states
(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.
element in line 12 of Figure 2 (at state s4 in Figure 3), we have $R(e, s4) = \{e, P(e), Ch(e)\}$, where $P(e) = \{\text{class="cssMain" valign="top"}, \text{<img src="img/head_notes.gif"}, \text{<p>, \text{input id="ok"}}\text{src="img/ok0.gif"}$. We define assertions of the form $A(s_j, c(R(e_j, s_j)))$ with a condition $c(R(e_j, s_j))$ for the region $R$ 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 $R(e_j, s_j)$ as $R(e_i, s_i)$, then generate assertion on $R(e_j, s_j)$ 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 $R(e_j, s_j)$ as $R(e_i, s_i)$, then generate assertion on $R(e_j, s_j)$ for checking its tag and attributes values.

3. RegionTagTextMatched: If a DOM state $s_j$ contains an element $e_j$ with exact tag value of $R(e_j, s_j)$ as $R(e_i, s_i)$, then generate assertion on $R(e_j, s_j)$ for checking its tag value.

Note that the assertion conditions are relaxed one after another. In other words, on a DOM state $s$, if $s \models \text{RegionFullMatched}$, then $s \models \text{RegionTagAttMatched}$; and if $s \models \text{RegionTagAttMatched}$, then we have $s \models \text{RegionTagTextMatched}$. Consequently, it suffices to use the most constrained assertion. We use this property for reducing the number of generated assertions in subsection 3.3.3.

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 $\text{assertTrue(isElementRegionFullPresent(parentElement, element, childrenElements))}$, where $\text{parentElement}$, $\text{element}$, and $\text{childrenElements}$ 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 generation 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 element but not exactly the same. For instance, consider Figure 2 line 12 in which a $\text{<span id="mainContent"} element was checked in an assertion. If in another state, a $\text{<div id="centreDiv"}$ element exists, which is similar to the $\text{<span>}$ element in certain aspects such as content and position on the page, we could generate a DOM-based assertion for the $\text{<div>}$ element in the form of $\text{assertTrue(isElementPresent(By.id("centreDiv")))}$. 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 $\mathcal{D} = \{(x_i, y_i) \mid x_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^n$, where each $x_i$ is a $p$-dimensional 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 $\mathcal{F}: \mathbb{R} \rightarrow \mathbb{R}$ maps a feature vector $x_i$ to its class label $y_i$. To do so, we use Support Vector Machine (SVM) [32] to find the maximal 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 $\psi: e \rightarrow x$ maps an element $e$ to its feature set $x$. 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 $e = \text{<span>}$ in Figure 1 then $\psi(e) = \langle 0.5, 0.7, 0.6, 0.5, 1, 0.2, 0, 0.3 \rangle$ 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 $\mathcal{F}(\psi : e \rightarrow x) = 1$ 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
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  with an existing assertion , a new assertion generated for is treated as follows:

\[
\begin{align*}
\text{Discard } A & \quad \text{if } B \implies A \\
\text{Replace } B \text{ with } A & \quad \text{if } A \implies B \land B \notin \text{original assertions} \\
\text{Add } A \text{ to } s & \quad \text{otherwise}
\end{align*}
\]

**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 CrawliX [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?
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 [3] 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 a study of common mistakes made by web developers. Examples include changing the ID/tag name used in getElemtById and getElemtByName 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.

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

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 getElemtById and getElemtByName 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.

Original SFG Coverage (RQ1). Table 3 shows the average results of our experiments. As expected, the number of states, transitions, and generated test cases is 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.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th># States</th>
<th># Transitions</th>
<th># Test Cases</th>
<th># Orig States Coverage</th>
<th># Orig Transition Coverage</th>
<th>Orig Input Data Usage</th>
<th>Orig Assertion Usage</th>
<th>JS Code Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>37</td>
<td>46</td>
<td>15</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>20%</td>
</tr>
<tr>
<td>EXND</td>
<td>54</td>
<td>63</td>
<td>47</td>
<td>98%</td>
<td>96%</td>
<td>100%</td>
<td>100%</td>
<td>26%</td>
</tr>
<tr>
<td>RAND</td>
<td>53</td>
<td>40</td>
<td>25</td>
<td>65%</td>
<td>60%</td>
<td>0%</td>
<td>0%</td>
<td>22%</td>
</tr>
</tbody>
</table>

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 FullRegion-Matches type. On the other hand, learned assertions are less strict (e.g., AttTagRegionMatches) 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 5: Average number of assertions per state, before and after filtering unstable assertions.](image)

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.

![Figure 6: Comparison of average fault detection rate using different test suite generation methods.](image)
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, Sprengle 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 defense analysis of server-side code. Pezze et al. [26] present a technique to generate integration test cases from existing unit test cases. Mirzaaghaeei 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


