Diversity-Based Web Test Generation

Matteo Biagiola
Fondazione Bruno Kessler
Trento, Italy
biagiola@fbk.eu

Andrea Stocco
Università della Svizzera Italiana
Lugano, Switzerland
andrea.stocco@usi.ch

Filippo Ricca
Università degli Studi di Genova
Genoa, Italy
filippo.ricca@unige.it

Paolo Tonella
Università della Svizzera Italiana
Lugano, Switzerland
paolo.tonella@usi.ch

ABSTRACT

Existing web test generators derive test paths from a navigational model of the web application, completed with either manually or randomly generated input values. However, manual test data selection is costly, while random generation often results in infeasible input sequences, which are rejected by the application under test. Random and search-based generation can achieve the desired level of model coverage only after a large number of test execution attempts, each slowed down by the need to interact with the browser during test execution. In this work, we present a novel web test generation algorithm that pre-selects the most promising candidate test cases based on their diversity from previously generated tests. As such, only the test cases that explore diverse behaviours of the application are considered for in-browser execution. We have implemented our approach in a tool called DIG. Our empirical evaluation on six real-world web applications shows that DIG achieves higher coverage and fault detection rates significantly earlier than crawling-based and search-based web test generators.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

web testing, test generation, diversity, page object

ACM Reference Format:
Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-Based Web Test Generation. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338970

1 INTRODUCTION

Modern web applications available nowadays provide the same high level of user interaction as native desktop applications, while eliminating the need for in-site deployment, installation, and update. For instance, single-page web applications (SPA) achieve high responsiveness and user friendliness by dynamically updating the Document Object Model (DOM) of a single web page by means of JavaScript functions that react asynchronously to user events.

To test such complex software systems, engineers typically adopt test automation frameworks such as Selenium [46]. In this context, the tester verifies the correct functioning of the application under test (AUT) using test scripts that automate the set of manual operations that the end user would perform on the web application's graphical user interface (GUI), such as delivering events with clicks, or filling in forms [5–7, 14, 19, 24–26, 51]. Testers implement business-focused test scenarios within such test scripts, along with the necessary input data. Each test case, hence, exercises a specific test path along the navigational graph of the web application.

Despite the research advancements in automated test generation [16, 18, 40, 42, 47], only a few tools have been proposed in the web domain [5, 32, 36], whose applicability to the complex systems available nowadays is still unfortunately quite limited. Consequently, to date the development of system-level test cases, such as Selenium's, is still performed mostly *manually*.

Generating web test cases automatically is indeed quite challenging, because specific sequences of events and associated input data, must be generated so as to ensure an adequate level of coverage. Existing approaches rely on crawling to obtain a navigational model of the AUT in the form of a graph of the dynamic web pages and the event-based interactions between them. Then, test paths (also called abstract test cases) are derived from the model in order to ensure full coverage of the navigational graph using *e.g.*, breadth-first visit [36] or semantically interacting events [32]. To generate executable tests, test paths require specifying proper input data, which can be produced either manually or by random generation.

However, these approaches have several drawbacks that limit their applicability in practice. First, random input generation makes existing approaches quite ineffective, because they may either (1) exercise the same portion of the AUT repeatedly, without exploring new behaviours that are likely to expose previously unknown faults, or (2) produce infeasible tests, which violate constraints and preconditions of the AUT. The drawback is that a huge number of random test cases needs to be generated before covering the navigational

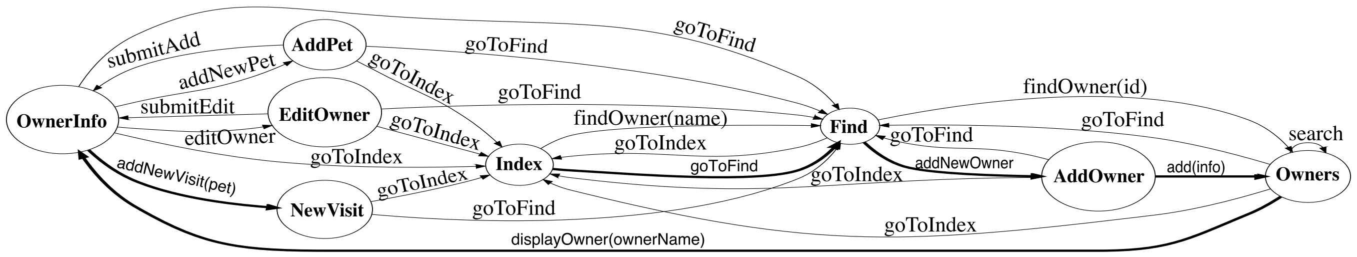

Figure 1: Simplified navigational model of PetClinic web application. Transitions between states are labelled with the corresponding actions, which can be parameterized. The path in bold represents an instance of coverage-maximizing test path, which are those our diversity-based technique aims to generate.

graph adequately, which is highly inefficient. Additionally, web tests are characterized by slow execution times due to the interaction with the browser, which further hinders the applicability of existing test generation techniques when strict timing constraints apply. As such, the mentioned techniques resort mostly to manual input generation, which is however costly, and limited by the testing resources (*e.g.*, time) available to testers.

To overcome these limitations, in this paper, we propose a novel solution for generating system-level web test cases. The goal of our approach is to generate a set of test cases that *diversify the coverage of the navigational graph and the use of input data*, in order to escape infeasible scenarios that are not executable in practice, or that do not produce any coverage improvement.

The key insight of our approach consists in assessing the quality of newly generated test cases *without executing them*, just by evaluating their *diversity* with respect to previously executed candidates. Diversity-based test generation allows escaping local optimum regions of the input space, where similar test cases might all cover the same code and navigations, and only the most diverse candidates are selected for in-browser execution.

We implemented our approach in a tool named DIG (**DI**versity-based **G**enerator), which automatically extracts a navigational model of the web application through crawling and creates a code-based abstraction of the possible actions executable in each web page, following the page object design pattern [15]. Then, candidate test cases (*i.e.*, possible paths and inputs) are generated using a diversity-based heuristics so as to cover the model adequately and efficiently.

Our work makes the following main contributions:

- A novel diversity-based system-level test case generator for web applications.
- An algorithm for identifying coverage- and input-maximizing candidate test cases. Our algorithm uses a novel diversity-aware distance metric between test cases so as to minimize the need for in-browser executions.
- An implementation of our approach in a tool called DIG, which automatically extracts a navigational model composed of page object abstractions, and generates test cases maximizing both navigation sequence and input data diversity [52].
- An empirical evaluation of DIG in generating test suites for six open-source web applications. DIG generated test suites having higher coverage and fault detection rates significantly faster than crawling-based and search-based test generators.

2 BACKGROUND

We present the web testing concepts that are needed to understand the remainder of the paper. We provide background information on the navigational model of a web application, its characteristics and properties, and we explain how it can be used to enable automated test case generation.

2.1 Navigational Model

The navigational model of a web application can be represented as a State-Flow Graph (SFG), where nodes are the dynamic DOM states of the web pages, and the edges the event-based transitions between them [35].

Definition 1 (**State-Flow Graph**). *A state-flow graph SFG for a web application $\mathcal{W}$ is a labeled, directed graph, denoted by a 4-tuple $\langle r, \mathcal{V}, \mathcal{E}, \mathcal{G} \rangle$ where:*

1. *r is the root node (called Index) representing the initial DOM state when $\mathcal{W}$ has been fully loaded into the browser.*
2. *$\mathcal{V}$ is a set of vertices representing the nodes. Each $v \in \mathcal{V}$ represents an abstract DOM state of $\mathcal{W}$.*
3. *$\mathcal{E}$ is a set of directed edges between vertices, which we call actions. Each $(v_1, v_2)_{[g]a} \in \mathcal{E}$ represents a possible transition between two nodes v_1, v_2 if and only if node v_2 is reached by executing the action a in node v_1 and the guard g is satisfied.*
4. *$\mathcal{G}$ is a set of guards on edges $(v_1, v_2)_{[g]a} \in \mathcal{E}$.*
5. *SFG can have multi-edges and be cyclic.* □

2.2 Example of Navigational Model

Figure 1 shows a simplified navigational model of *PetClinic* [43], a web app allowing veterinarians to manage information about their clients and their pets, which is among the experimental subjects used in our evaluation. The graph consists of eight abstract DOM states, each having several possible actions that can trigger transitions between them.

Next, we define the notion of feasibility, and its impact on automated test case generation on our running example.

2.3 Feasibility

Given an abstract DOM state *n*, the set of possible actions $\mathcal{E}$ may be constrained by one or more guards (*a.k.a.*, preconditions).

Definition 2 (**Guard**). *A guard $g(s, a, i)$ is a boolean condition over the possible input values $i \in I$ of an action a in a specific application state s.*

The guard needs to be satisfied in order to enable the transition between state n and the target state of the action a (typically another state n', or n itself). The application state s includes the global variable values, the DOM, and any persistent data that remain available across user interactions.

Hence, satisfiability of a guard depends on: (1) the input values $i \in I$ generated for the given action a, as well as, (2) the internal business logic state s of the application, produced by previous user interaction sequences.

For instance, in the `Owners` state of Figure 1, an action `displayOwner` allows to navigate towards the `OwnerInfo` state. A simple guard over this action imposes that the owner identified by the input `ownerName` must be present in the application prior to executing the action. (For readability, we omitted the guards in the figure.) As such, the guard depends not only on the specific value assigned to the input `ownerName` of the `displayOwner` action, but also on the internal business logic state of the application. In fact, the owner to display must have been previously inserted into the application by a dedicated action (*e.g.*, by executing the `add` action in the `AddOwner` state).

To recap, for a given action a, if the input values do satisfy the guards g of a in the given application state s, then we say that this transition is *feasible*; it is *infeasible* otherwise.

2.4 Test Generation Problem

Given a navigational model of a web application, the goal of a web test generator is to automatically extract sequences of actions and generate suitable inputs in order to exercise the application behaviours thoroughly, potentially covering all transitions in the navigational model.

Definition 3 (**Test Path**). *A Test Path is a sequence of test states paired with a corresponding sequence of actions with unspecified input values.*

Then, a test case can be defined by instantiating concrete input values within a test path.

Definition 4 (**Test Case**). *A Test Case is a sequence of concrete values for a corresponding sequence of actions in a specific test path.*

For example, in PetClinic, a simple test path that enables the addition of a new visit for a pet consists of the test state sequence ⟨`Index`, `Find`, `AddOwner`, `Owners`, `OwnerInfo`, `NewVisit`⟩. The corresponding action sequence is given by ⟨`goToFind`, `addNewOwner`, `add`, `displayOwner`, `addNewVisit`⟩.

In this sequence of actions, three input values need to be specified: (1) `info`, the data of the owner to insert, required by action `add` of state `AddOwner`, (2) `ownerName`, the name of the owner, required by action `displayOwner` of state `Owners`, and (3) `pet`, the data of the pet to be visited, required by action `addNewVisit` of state `OwnerInfo`. The action `addNewVisit` is guarded by a precondition that states that the pet identified by `pet` must exist before the execution of the action. Since in the considered test path no pet is added by any action before executing `addNewVisit`, the chosen test path cannot be taken, for any possible values of input `pet`. Therefore, we say that such path is *infeasible*.

Definition 5 (**Test Path Feasibility**). *A Test Path is feasible if there exists an input parameter-value assignment that satisfies all the guards related to the actions in the test path; it is infeasible otherwise.*

2.5 Existing Test Generation Approaches

Different strategies can be adopted to generate tests cases from a navigational model. In this section, we briefly discuss the most notable existing solutions, as well as their limitations, which motivate the need for a novel and more efficient approach. In Section 3, we will then describe our novel diversity-based test generator.

2.5.1 Graph Visit Approaches. Breadth-first and depth-first graph visit algorithms have been applied to the navigational model of a web application to derive test paths (also called abstract test cases, *i.e.*, sequences of states and actions lacking concrete input data to become executable). In the tool ATUSA [37], random input data generation has been proposed to fill the gap between test paths and concrete test cases. However, randomly generated inputs have often low chance of producing feasible test cases, hence they typically require a huge number of input generations and corresponding test executions. Moreover, ATUSA generates big test suites even for simple web applications [5, 32], because it relies on a state abstraction function that creates a large number of test states and, correspondingly, a large number of test paths.

2.5.2 Semantic-Based Approaches. Alternatively, test paths can be restricted to those exercising semantically relevant interactions, *i.e.*, event sequences in which the ordering of events affects the state reached at the end of the interactions [57]. Such approach has been successfully applied to web applications [32], but yet it produces a set of abstract test cases (semantically interacting event sequences) that require manual specification of input data to make them executable.

2.5.3 Search-Based Approaches. Search-based techniques iteratively sample the input space, selecting the fittest candidate test cases, and evolving the fittest ones using genetic search operators to create new test cases [29]. Since these algorithms can effectively guide the generation of test cases even for large input spaces, they are suited for system-level testing [58]. Concerning the web domain, an effective fitness function can be defined based on *approximate* information available in the navigational model—specifically the actions' guards. Researchers have shown that this approach can guide the search toward generating test cases unaffected by the path infeasibility problem [5]. However, this approach needs the manual specification of all guards for each action, a task that is time-consuming and laborious for testers. Indeed, such information depends on the web application business logic and intended behaviour, and thus cannot be generated fully automatically. Additionally, the evaluation of the fitness function is costly, because it requires a large number of candidates to be generated and executed in the browser before converging to an adequate set of tests.

2.5.4 Summary. While all discussed approaches provide theoretical warranties of asymptotic convergence to the desired input values, they exhibit poor execution time performance when applied to web applications, as compared, for instance, to standard

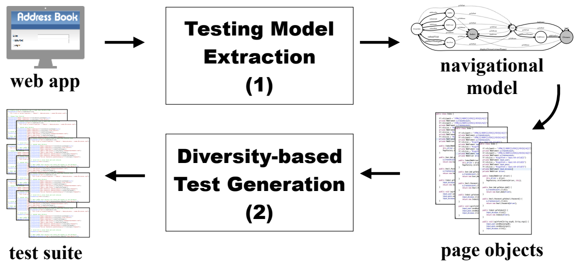
web app
Testing Model Extraction (1)
navigational model
page objects
Diversity-based Test Generation (2)
test suite

Figure 2: System-level web test generation approach

Java desktop applications [17]. The reason why both random and search-based approaches are computationally expensive is that they always need to execute the candidate test cases to assess their feasibility and, in the latter case, the value of the fitness function. This disadvantage is amplified in web testing because (1) the overhead imposed by browser's interaction makes evaluating these tests very slow, and (2) the input space may contain local optimum regions, in which all candidate test cases are likely to be behaviourally equivalent (*e.g.*, all equally infeasible), hence increasing the run time of the test generation algorithm without benefiting the overall coverage [5]. The limitations discussed in this section (test suite's size, high computational cost, need for manual guard specification) justify the investigation of alternative, possibly cheaper and more automated algorithms.

3 DIVERSITY-BASED WEB TEST GENERATION

In a nutshell, our goal is to efficiently generate system-level web test cases that exercise the application behaviours adequately. Figure 2 depicts the main steps of our approach, which is based on the inference of testing model as follows. First, a navigational model of the dynamic web pages is extracted ❶. Second, each web page is modelled in form of page objects [15], object-oriented classes that expose the actions executable in each web page as methods. Finally, our test generator uses a novel metric distance between test actions and input data to generate a test suite that exercises the web application thoroughly ❷. Our approach does not require specifying the guards of the actions in the navigational model.

We now detail each step of our approach, as well as the computation of the distance between tests, which is exemplified on our running example.

3.1 Testing Model Extraction

In the first step, we obtain a navigational model of the web application. This can be performed manually by testers, through manual exploration of the web app functionalities, or automatically. Our approach adopts this latter option: we use a web crawler to automatically explore the state space of the web application (see Section 3.5 for more details). The output of this phase is a navigational model of the application, in which nodes are the dynamic DOM states and edges are the event-based transitions between them.

The navigational model retrieved by the crawler is used to generate an intermediate code-based abstraction that can conveniently support our diversity-based automated test case generation. We use page objects [15] (hereafter referred to as POs) as a concrete representation for the DOM states in the navigational model and their actions. Following the guidelines provided by this design pattern, each dynamic DOM state is represented as an object-oriented class whose main functionalities are exposed to testers as PO methods. In our running example *PetClinic*, for instance, the `AddOwner` state would be represented by its own class, containing three methods (`add`, `goToIndex` and `goToFind`).

The motivation for the choice of POs as our state representation is twofold: (1) page objects are a well-known and utilized pattern in web test automation, to enhance the development and maintainability of test cases, and (2) they can be created automatically from a given navigational model with a good degree of accuracy [50].

3.2 Diversity-Based Path and Input Generation

Inspired by adaptive random testing [9], which makes the assumption of contiguous failure regions, we conjecture that web applications might also have **contiguous infeasibility regions**. Correspondingly, in the case of web apps the *main advantage* of test case diversity would be the possibility of exploring the search space at large, diversifying the region where navigation sequences are sampled. This is expected to help escaping local solutions, as well as avoiding generating infeasible test paths and ensuring a more effective exploration of alternative (*i.e., diverse*) behaviours. Indeed, research has shown that diversity is especially beneficial to fault detection. Diverse inputs are necessary to expose different failures, whereas inputs from contiguous areas of the input space are likely to expose the same program failure [8].

A *second advantage* of diversity-based test case generation is its higher efficiency with respect to existing random and search-based approaches. In fact, the quality of a candidate test case is evaluated by measuring its *diversity* with respect to previously generated test cases and using such metric to assess its potential at increasing the exploration of diverse behaviours.

Interestingly, such assessment can be performed without actually executing the candidate test cases, as described in Section 3.3.

Algorithm 1 describes our overall procedure for diversity-based path and input generation. The test generation starts from an empty set of tests, and therefore the first generated test is random. The algorithm generates a set C of *candidate* test cases, instantiating candidate test paths along with concrete input vectors (Lines 8–10). To select the most promising candidates, the distance between each candidate test case and the current set of already executed test cases T_{exec} is computed (Lines 11–15) and only the farthest test case t is executed (Line 16). Test case t is restricted to its feasible prefix in case it includes a divergence (Line 17–18). A *divergence* occurs whenever the test path of a test case t differs from the execution trace obtained by running t. Then, the test case is added to the final test suite TS_{gen} only if it contributes to increase coverage of the navigational model M (Lines 19–20).

Algorithm 2 shows how candidate test paths are created. In our notation, V represents a state sequence, A indicates a method sequence having X as a input vector sequence. V, A and X are incrementally created within the main loop (Lines 5–11) by choosing an edge $\langle v, v' \rangle_{[g]a}$ randomly, with uniform probability, among those available from state v according to the navigational model M.

Algorithm 1: Diversity-based Test Case Generation

```
Input   : $\mathcal{M}$: navigational model, $k$: number of candidates
Output  : $TS_{gen}$: test suite that optimizes coverage of $\mathcal{M}$
1  $T_{exec} \leftarrow \emptyset$                                        ▷ Set of executed test cases
2  $TS_{gen} \leftarrow \emptyset$                                        ▷ Set of generated test suite
3  generate randomly a test case $t$, add $t$ to $T_{exec}$, and execute it
4  while $\mathcal{M}$ is not adequately covered by $TS_{gen}$ or timeout is not reached do
5      $m \leftarrow$ getRandomUncoveredMethod($\mathcal{M}$)
6      $D_{max} \leftarrow 0$
7      $C \leftarrow \emptyset$                                           ▷ Set of candidate test cases
8      for $i \leftarrow 1$ to $k$ do
9          $C \leftarrow C \cup$ generateCandidateTest($\mathcal{M}$, $m$)   ▷ see Algo 2
10     end
11     for $c_i \in C$ do
12         $d_i \leftarrow min(\rho(c_i, t_j)) \quad \forall t_j \in T_{exec}$   ▷ see Eq 1
13         if $d_i > D_{max}$ then
14             $D_{max} \leftarrow d_i$
15             $t \leftarrow c_i$
16         end
17     end
18     add $t$ to $T_{exec}$ and execute it
19     if $t$ is divergent then
20         $t \leftarrow$ getFeasiblePrefix($t$)
21     end
22     if $t$ increases coverage of $\mathcal{M}$ w.r.t. $TS_{gen}$ then
23         $TS_{gen} \leftarrow TS_{gen} \cup \{t\}$
24     end
25 end
26 return $TS_{gen}$
```

Algorithm 2: Candidate Test Case Generation

```
Input   : $\mathcal{M}$: navigational model, $m$: target method
Output  : $\langle \mathcal{V}, \mathcal{A}, \mathcal{X} \rangle$: candidate test case reaching $m$
1  $\mathcal{A} \leftarrow \langle\rangle$
2  $\mathcal{X} \leftarrow \langle\rangle$
3  $v \leftarrow$ getRootNode($\mathcal{M}$)
4  $\mathcal{V} \leftarrow \langle v \rangle$
5  while $m \notin \mathcal{A}$ do
6      $\langle v, v' \rangle_{[g]a} \leftarrow$ getRandomEdge($\mathcal{M}$, $v$, $m$)   ▷ must ensure $v' \rightsquigarrow m$
7      $v \leftarrow v'$
8      $x \leftarrow$ getRandomInputVector($a$)
9      $\mathcal{V} \leftarrow \mathcal{V}.add(v)$
10     $\mathcal{A} \leftarrow \mathcal{A}.add(a)$
11     $\mathcal{X} \leftarrow \mathcal{X}.add(x)$
12 end
13 return $\langle \mathcal{V}, \mathcal{A}, \mathcal{X} \rangle$
```

The selection is constrained by the fact that the target method m must remain reachable from v'. At last, a vector of input values as parameters to a is randomly chosen (Line 8).

3.3 Distance Between Test Cases

Algorithm 1 requires a distance metric to assess the diversity between test cases. Differently from object-oriented [10, 28] or numerical applications testing [8], in our setting we cannot rely only on the input values, as the distance function ρ must also take into account the sequence of actions composing a test case. As such, we devised a novel distance metric between two test cases taking into account the *diversity of the respective sequences of actions*, and, in cases in which this is not discriminative, privileging the test case having the farthermost diversity in terms of concrete input values. By diversifying the sequence of actions, as well as the associated input values, we conjecture that we can escape infeasible test path regions and we can diversify the app behaviours.

3.3.1 Distance Formula. Intuitively, our distance formula comprises two terms, the first measuring the distance between the action sequences in the two test cases being compared and the second measuring the distance between the input values used by matching actions in the two sequences. To compute the first term, the distance function α reports the number of non-matching actions in the two sequences. To compute the second term, we rely on the longest common subsequence to obtain the matching actions; upon each matched actions, we compute the normalized distance β between their parameter values.

Given two test cases (t_i, t_j), where $t_i = \langle V_i, A_i, X_i \rangle$ and $t_j = \langle V_j, A_j, X_j \rangle$, the distance $\rho(t_i, t_j)$ is given by Equations (1), (2):

$$\rho(t_i, t_j) = \alpha(V_i :: A_i, V_j :: A_j) + \sum_{\langle k_1, k_2 \rangle \in LCS_{i,j}} \beta(X_i[k_1], X_j[k_2]) \quad (1)$$

$$\beta(x, y) = \frac{1}{|x|} \sum_{k \in [1 \ldots |x|]} \delta(x[k], y[k]) \quad (2)$$

In Equation 1, the notation $V_x :: A_y$ indicates that the two sequences of actions are identified both by their actions A_i, A_j and the states V_i, V_j in which they are applicable. This helps disambiguate cases in which an action A_y is present with the same name in different states.

Function α represents the *sequence edit distance* [11, 27], which determines the number of non-matching elements in two sequences (*e.g.*, $4 = |\langle a, b \rangle| + |\langle e, f \rangle|$, when comparing $\langle a, b, b, c \rangle$ to $\langle e, b, c, f \rangle$). *LCS* is the set of matching indexes in the *longest common subsequence* [11, 27] (*e.g.*, $\{(2, 2), (4, 3)\}$, when comparing $\langle a, b, b, c \rangle$ to $\langle e, b, c, f \rangle$). Function β computes the distance between the two parameter value sequences $X_i[k_1], X_j[k_2]$ of two matching actions (indexed by k_1 and k_2 respectively in $V_i :: A_i$ and $V_j :: A_j$). At last, function δ computes the normalized distance between two primitive input values of the same type (see Table 1).

The second term of Equation 1 matches the actions $A_i[k_1], A_j[k_2]$ in the two sequences. Correspondingly, the two input vectors $X_i[k_1]$ and $X_j[k_2]$ have the same cardinality. Function β computes the average distance between the parameter values x and y of two matching actions. It resorts to function δ to compute the normalized distance between pairs of primitive input values $x[k]$, $y[k]$. The computation of function δ varies depending on the type of parameters occurring into the actions (see Table 1). The input distance δ is normalized between 0 and 1. For types `string` or `number`, normalization is achieved using function $\eta(x) = \frac{x}{x+1}$ as proposed by Arcuri *et al.* [2], that maps a value x onto the range $[0, 1]$.

Equation 1 resembles the computation of the *fitness function* for branch coverage, commonly adopted in search-based testing [34].

Table 1: Input distance computation

Type	**Input Distance** $\delta(w, z)$
`boolean`	$\delta(w, z) = \{0 \text{ if } w = z; 1 \text{ otherwise }\}$
`enum`	$\delta(w, z) = \{0 \text{ if } w = z; 1 \text{ otherwise }\}$
`string`	$\delta(w, z) = \alpha(w, z)/(1 + \alpha(w, z))$
`number`	$\delta(w, z) = \lvert w - z \rvert / (1 + \lvert w - z \rvert)$

In such a case, the two terms of the fitness function are *approach level* and normalized *branch distance*, respectively. We share with that definition the idea of making the first term (action sequence distance) dominant, while resorting to the second term (input value distance) only when the first term is not discriminative enough. In fact, the range of α is $\mathbb{N}$, so the minimum non-zero distance is 1, whereas β ranges between 0 and 1, such that the contribution to ρ of each matching pair of actions cannot be greater than 1. The intuition is that the major contribution to diversity comes from the action sequence distance, while the input value distance for each matching action pair contributes at most as the minimum non-zero action sequence distance (*i.e.*, 1). However, in case of a large number of actions being matched, the aggregate value of the input value distance might become dominant.

3.4 Example of Distance Computation

We present how the distance between test cases is computed using a simplified notation in which only the actions are reported while omitting the states (*e.g.*, `Index::findOwner` becomes `findOwner`). In fact, in all our examples the state sequence associated with a given action sequence will be unique (this is not true in the general case). Moreover, parameter values are indicated in brackets rather than separate input vectors (*e.g.*, an action sequence $A = \langle$`findOwner`$\rangle$ and the corresponding input sequence $X = \langle$(“`John Doo`”)$\rangle$ become $\langle$`findOwner(“John Doo”)`$\rangle$).

Let us consider three simple test cases t_1, t_2, t_3 for the running example PetClinic (Section 2), defined as follows:

$t_1 = \langle$`findOwner("John Doo"), find(101)`$\rangle$
$t_2 = \langle$`findOwner("Johnny Boo"),addNewOwner,add("Johnny Boo")`$\rangle$
$t_3 = \langle$`findOwner("John Doo"), find(105)`$\rangle$

Suppose that t_1 is already in the list of generated test cases T_{gen}, whereas t_2 or t_3 are in the set of generated tests C. Our algorithm must decide which test to execute next. The sequence edit distance between t_1 and t_2 is $\alpha = 3$, because there are three non-matching actions in t_1, t_2 (`find` in t_1, and `addNewOwner`, `add` in t_2).

Concerning the matching actions (`findOwner` in both t_1 and t_2), the input distance is computed by function β, which in turn resorts to function δ for the distance between primitive values. In our running example, β((`"John Doo"`), (`"Johnny Boo"`)) = δ(`"John Doo"`, `"Johnny Boo"`) = 3/4 = 0.75, because α(`"John Doo"`, `"Johnny Boo"`) = 3 (the three non matching characters in `"Johnny Boo"` being ‘`n`’, ‘`y`’, ‘`B`’). Therefore, $\rho(t_1, t_2) = 3 + 0.75 = 3.75$.

Conversely, the sequence edit distance between t_1 and t_3 is $\alpha = 0$, since there are no unmatched actions in the two sequences. Thus, our distance formula relies on the input distance β((`"John Doo"`), (`"John Doo"`)) = 0, and β((`101`), (`105`)) = 4/5 (we use the δ function that applies to numbers). Therefore, $\rho(t_1, t_3) = 0 + 4/5 = 0.80$. Thus, the *most diverse* test case t_2 is selected and executed.

3.5 Implementation

We implemented our approach in a Java tool called DIG (**DI**versity-based **G**enerator), which is publicly available [52]. To retrieve the PO testing model, DIG relies to Apogen [50]. Finally, our diversity-based test generator is implemented on top of EvoSuite [17], which we extended to handle the PO model. Our algorithm uses the PO method information to generate and evaluate candidate test cases. At last, only the most diverse candidates are executed through Selenium WebDriver [46] in *headless* execution mode.

4 EMPIRICAL VALIDATION

4.1 Research Questions

We address the following research questions:
RQ$_1$ (Effectiveness): *How do diversity-, search-, and crawling-based random test generation compare in terms of state coverage, code coverage and fault detection?*
RQ$_2$ (Efficiency): *How do diversity- and search-based test generation compare in terms of efficiency over time?*
RQ$_3$ (Distance Computation): *What is the impact of distance computation in the diversity-based test generation process?*
RQ$_4$ (Manual POs): *What is the effect of using manually defined POs within the diversity- and search-based test generation approaches?*

RQ$_1$ and RQ$_2$ aim to compare DIG with two state-of-the-art solutions: a search-based web test generator called SubWeb [5], and ATUSA [37], which is based on a crawling-based random approach. RQ$_3$ and RQ$_4$ aim at assessing the impact of the internal factors of the proposed approach (namely, distance computation and POs generation method) on the final test suites.

4.2 Subject Systems

We overviewed the most popular JavaScript frameworks for developing web applications from *GitHub Collections* [21]. Popularity was measured as the number of stars owned by the framework's GitHub repository at the time of writing (August 2018). We retained five frameworks with more than 15k stars. Second, we selected web applications developed with one of the selected frameworks and are popular (number of stars $\geq$ 50), mature (number of commits $\geq$ 50) and have been maintained recently (year of last commit $\geq$ 2016). Third, from the resulting candidate set, in order to maximize diversity and representativeness, we randomly sampled six applications considering the six most popular Javascript frameworks: dimeshift (Backbone.js), pagekit (Vue.js), splittypie (Ember.js), phoenix-trello (Phoenix/React), retroboard (React), PetClinic (AngularJS).

Table 2 summarizes the main characteristics of our subjects. The size of the selected systems (> 1k client-side JavaScript LOCs, frameworks excluded) is representative of modern web applications [39] (Ocariza *et al.* [39] report an average of 1,689 LOCs for a dataset of web applications developed with the AngularJS framework with at least 50 stars).

Table 2: Experimental subjects

Subject	Framework	LOC (JS)	Stars	Commits	Year
dimeshift [12]	Backbone	5,140	127	194	2018
pagekit [41]	Vue.js	4,214	4,851	4,914	2019
splittypie [48]	Ember.js	2,710	67	331	2017
phoenix [44]	React	2,289	2,233	422	2016
retroboard [45]	React	2,144	390	476	2019
PetClinic [43]	AngularJS	2,939	50	71	2018

Table 3: Effectiveness, Efficiency and Distance Computation results (Automated POs) for RQ_1, RQ_2, and RQ_3 for all subjects and approaches. Values in bold indicate statistically significant differences between DIG and SubWeb. Stars indicate statistically significant differences between DIG_{S+I} and DIG_{S}. NC indicates non-comparable values since the test generation terminates before the given time budget.

	Effectiveness											Efficiency									Distance Computation						
	Structural Coverage							**Faults**				**Structural Coverage**						**Faults**			**Tests**					**Distance**	
	State Cov. (%)			*Branch Cov. (%)*				*Avg Unique (#)*				*State AUC (%)*			*Branch AUC (%)*			*AUC (%)*			*Gen.*			*Exec.*		*#*	
	DIG_{S+I}	DIG_{S}	SubWeb	DIG_{S+I}	DIG_{S}	SubWeb	ATUSA	DIG_{S+I}	DIG_{S}	SubWeb	ATUSA	DIG_{S+I}	DIG_{S}	SubWeb	DIG_{S+I}	DIG_{S}	SubWeb	DIG_{S+I}	DIG_{S}	SubWeb	DIG_{S+I}	DIG_{S}	SubWeb	DIG_{S+I}	DIG_{S}	DIG_{S+I}	DIG_{S}
dimeshift	99.4	**100.0**	98.4	40.5	40.4	40.3	18.8	2.7	**3.3**	2.5	1.0	**97.2**	**97.2**	95.1	**36.2**	35.9	34.9	**50.0**	**56.2**	35.2	NC	NC	NC	NC	NC	NC	NC
pagekit	96.3	97.6	96.3	27.9	27.2	28.0	24.8	1.0	1.1	1.1	0.0	**94.5**	**96.2**	93.3	24.6	24.6	26.6	33.0	34.6	41.7	69.5*k*	7.9*k*	218	158	218	487*k*	628*k*
splittypie	94.0	93.8	91.0	51.9	51.7	50.1	18.6	5.6	6.0	5.6	1.0	**91.0**	**90.7**	86.3	**47.3**	**45.8**	44.3	84.5	89.6	86.2	11*k*	11.8*k*	329	221	236	1,227*k*	1,398*k*
phoenix	99.7	99.7	99.2	64.1	62.8	62.5	34.2	3.1	**3.5**	3.1	0.0	**98.1**	**98.1**	94.8	**58.0**	**58.7**	55.4	**58.1**	**66.2**	47.8	NC	NC	NC	NC	NC	NC	NC
retroboard	100.0	100.0	100.0	71.3	71.7	69.8	51.4	0.0	0.0	0.0	0.0	**97.9***	96.1	96.3	**69.1**	**68.7**	67.4	0.0	0.0	0.0	NC	NC	NC	NC	NC	NC	NC
PetClinic	100.0	100.0	100.0	85.0	85.0	85.0	0.0	2.5	2.9	2.1	0.0	96.4	95.8	97.3	**79.1**	75.7	68.7	41.0	**45.6**	32.5	NC	NC	NC	NC	NC	NC	NC
Average	99.7	98.5	97.5	56.8	56.5	56.0	24.6	2.5	2.8	2.4	0.3	95.9	95.7	93.9	52.4	51.6	49.6	44.4	48.7	40.6	40.2*k*	15.3*k*	274	180	197	857*k*	1,000*k*

4.3 Procedure and Metrics

Effectiveness (RQ_1). We ran DIG, SubWeb and ATUSA on each subject system. For DIG, we set the number of candidate test cases generated at each step of the algorithm to 50. As SubWeb requires, we manually specified the guards for the POs methods. We granted each tool the same time budget of 30 minutes because, in our exploratory experiments, we empirically observed convergence of state coverage to a plateau within half an hour. Additionally, we repeated each experiment 15 times, and computed the average across all executions to cope with non-deterministic behaviours.

We considered three metrics of effectiveness. First, we measured the *state coverage* of the navigational model according to the transition coverage adequacy criterion. Second, we measured *branch coverage* of the JavaScript code of our subject systems. We instrumented the client-side of each web application (libraries and framework excluded) with the tool `Istanbul` [20], and executed the generated test suites against the instrumented applications. Third, we studied the *fault detection* capability, by counting the number of unique faults (*i.e.*, unique JavaScript exceptions and errors) reported in the JavaScript console upon test suite execution.

Efficiency (RQ_2). To assess *efficiency*, we measured how the competing algorithms perform over time in terms of state coverage, branch coverage, and fault detection. To this aim, we computed the area under the curve (AUC) of each metric as a function of the test generation time, with higher values of AUC denoting a superior efficiency of the algorithm within the given time budget (30 minutes).

Concerning state coverage, we computed the AUC accurately, because EvoSuite outputs the value of such metric at every new test case generation. Differently, to measure AUC for branch coverage and fault detection, we had to execute each intermediate test suite produced during the test generation process. Given the huge number of test suites to be considered (in the order of dozens of thousands considering all applications, approaches and repetitions), for each subject, we sampled three time intervals: the point at which the state coverage difference between DIG and SubWeb is maximal, the point at which is 50% of the maximum, and 30 minutes. In fact, by graphically plotting the two state coverage functions, we observed that state coverage difference has a steep peak followed by a smooth decline. Hence, in order to accurately estimate the AUC with only a few data points (time intervals), it makes sense to sample them where the difference is the largest, and the next one, when it is halved.

Distance Computation (RQ_3). Distance computation is expensive because it grows quadratically with the number of test cases: at each test generation step, the number of distance computations is given by the number of previously executed test cases multiplied by the number of candidates. The input distance term of Equation 1 further increases the cost for distance computation. To assess such impact, we ran DIG disabling the input distance computation, thus computing the distance as just the sequence edit distance. We refer to these two variants of our approach as DIG_{S} and DIG_{S+I}, respectively, and computed the same effectiveness/efficiency metrics used for RQ_1 and RQ_2. Additionally, we assessed the overhead of distance computation on the number of test executions by reporting the number of tests generated and executed by each tool/configuration.

Manual POs (RQ_4). A developer may develop more accurate POs than those produced by an automatic technique. Thus, we compared the effectiveness and efficiency of DIG and SubWeb when manually defined POs are utilized.

Unfortunately, none of our subjects was equipped with PO-based test suites. Thus, we had to manually create the POs for each subject application. To minimize any subjectivity/bias, we developed the POs prior to running Apogen, by adopting a rigorous procedure. Specifically, we adhered to the guidelines given by Van Deursen [53, 54] on the design of PO-based web test suites. Each PO represents a test state, with explicit responsibilities for state navigation and state inspection. Thus, we represented each action of a test state as a *trigger method* in the page object. Instances of such methods are, for instance, clicks and data-submitting forms that bring the browser to a new state. *Inspection methods* have been used to retrieve the value of key/unique elements displayed in the browser when it is in a given state, such as a username.

In light of these design considerations, we modelled the web applications into POs as follows. Starting from the root (*i.e.*, the initial web page, typically the login page), we modelled it as a PO. Following the navigations that were possible from the initial state, new test states were discovered (e.g., for user registration), which in turn were modelled as POs. Then, we proceeded following the actions that were possible in each newly discovered state, building page objects iteratively and incrementally, until all the pages were accounted for. Additionally, states having common behaviours (e.g., menu bars) were organized into reusable components [53, 54].

To compare automatically *vs* manually generated POs, we computed the same effectiveness and efficiency metrics used to answer RQ_1 and RQ_2, as described above.

4.4 Results

Effectiveness (RQ_1). Table 3 (*Effectiveness*) compares DIG_{S+I}, DIG_S, and SubWeb in terms of state coverage, code coverage and fault detection. Statistical significance of the difference was assessed by applying the non-parametric Mann-Whitney U test [22], with a confidence threshold $\alpha = 0.05$.

Concerning ATUSA, we do not report comparisons in terms of state coverage, because the navigational model retrieved by the crawler is different (usually, much larger) from those based on the PO abstraction that DIG generates. To compare them accurately, one would need to find the isomorphism between the two SFGs, which requires mapping each state and each transition from one model onto the other—a manual and expensive process. Thus, for ATUSA, we only compare code coverage and fault detection metrics.

Looking at the results, both DIG and SubWeb outperform ATUSA substantially. In all cases the differences are statistically significant (not reported in Table 3).

As far as effectiveness is concerned, when automated POs are utilized, DIG and SubWeb can be considered comparable test generators, with minimal performance variations.

Despite both tools cover almost all navigational models within the given time budget of 30 minutes, there is however a remarkable difference. DIG is totally automated, and achieved these results by relying only its diversity heuristic. SubWeb, on the contrary, is semi-automatic, as it takes advantage of manually-defined preconditions to guide the search and avoid path infeasibility.

RQ_1: *Diversity- and search-based approaches achieve substantially higher state coverage, code coverage and fault detection than the crawling-based random approach. Despite being comparably effective, the diversity-based approach is preferable because it is fully automated.*

Efficiency (RQ_2). Table 3 (*Efficiency*) compares DIG and SubWeb in terms of state coverage, code coverage and fault detection achieved over time (AUC metrics). DIG outperforms SubWeb by a statistically significant amount in 5/6 subjects for state and branch coverage. Regarding fault detection, DIG is significantly better than SubWeb in 3/5 subjects (*retroboard* revealed no faults).

The plot in Figure 3 shows a meaningful example regarding the efficiency difference on state coverage for *phoenix*. DIG reaches the maximum state coverage after nearly one third of the total time

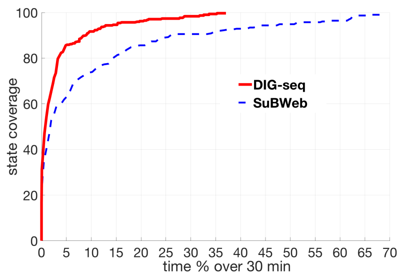

Figure 3: Average efficiency over time in terms of state coverage of the compared approaches on *phoenix*.

budget, whereas SubWeb takes approximately twice as much time. Moreover, for small time intervals, the effectiveness difference of DIG *vs* SubWeb is further amplified. For instance, the maximum difference between the two algorithms is 22% after 2 minutes (≈6% of the time budget). In practice, this means that DIG is preferable if strict testing time constraints apply.

RQ_2: *The diversity-based approach achieves high coverage and fault detection rates substantially faster than the search-based approach.*

Distance Computation (RQ_3). Table 3 (*Distance Computation*) shows the number of test cases generated and executed by DIG and SubWeb and the number of distance computations required by DIG_{S+I} and DIG_S (for SubWeb the number of generated test cases always equals the executed test cases).

On average, DIG computed a large number of distances ($857k$ by DIG_{S+I} and $1,000k$ by DIG_S). Such computations reduces the time available for test case execution. Thus, while SubWeb run on average 274 test cases, DIG_{S+I} and DIG_S run an average of 94 and 77 less test cases, respectively (*dimeshift*, *phoenix*, *retroboard* and *PetClinic* are excluded from the analysis since they always terminate before 30 minutes, which means that the number of test executions is not constrained by the time budget).

The number of generated test cases (not necessarily executed) by DIG is substantially higher than the test cases generated and executed by SubWeb (e.g., $40.2k$ and $15.3k$ *vs* 274). However, despite a lower number test executions, the time spent in distance computation allows DIG to produce test cases that have a higher chance of increasing coverage and fault detection (see results for **RQ_1** and **RQ_2**). This confirms our initial hypothesis that it is possible to assess the quality of web test cases without executing them, while still achieve high coverage and fault detection rates.

Let us now compare DIG_{S+I} and DIG_S. The extra computational cost associated with the input distance reduces on average the number of executed test cases by ≈9%. On the other hand, the increased accuracy of the distance computed by DIG_{S+I} does not bring considerable advantages in terms of coverage or fault detection (see results for **RQ_1** and **RQ_2**).

RQ_3: *The overhead brought by the distance computation is overshadowed by the benefits in efficiency and automation. Moreover, the input component of the proposed distance metric can be discarded with little to no associated penalty.*

Table 4: Effectiveness, Efficiency and Distance Computation results (Manual POs) for RQ$_4$, and RQ$_3$ for all subjects and approaches. Values in bold indicate statistically significant differences between DIG and SubWeb. Stars indicate statistically significant differences between DIG$_{S+I}$ and DIG$_S$.

	Effectiveness											Efficiency			Distance Computation						
	Structural Coverage							Faults				Structural Coverage			Tests					Distance	
	State Cov. (%)			*Branch Cov. (%)*				*Avg Unique (#)*				*State AUC (%)*			*Gen.*		*Exec.*			*#*	
	DIG$_{S+I}$	DIG$_S$	SubWeb	DIG$_{S+I}$	DIG$_S$	SubWeb	ATUSA	DIG$_{S+I}$	DIG$_S$	SubWeb	ATUSA	DIG$_{S+I}$	DIG$_S$	SubWeb	DIG$_{S+I}$	DIG$_S$	SubWeb	DIG$_{S+I}$	DIG$_S$	DIG$_{S+I}$	DIG$_S$
dimeshift	**70.8**	**70.3**	67.2	**41.7**	**41.3**	40.1	18.8	2.7	3.1	2.8	1.0	**67.4**	**66.9**	62.2	31.4*k*	27.0*k*	628	541	490	6,015*k*	7,331*k*
pagekit	**65.0**	**65.1**	61.6	34.5	**35.3**	33.3	24.9	2.7	2.8	2.7	0.0	**60.4**	**60.6**	56.4	9.7*k*	7.3*k*	196	146	142	965*k*	537*k*
splittypie	**74.7**	**74.4**	67.0	**51.7**	**51.4**	50.6	18.7	7.0	6.9	6.7	1.0	**70.8**	**70.6**	61.4	13.7*k*	12.5*k*	275	250	225	1,271*k*	1,569*k*
phoenix	**72.1**	**74.6**	68.8	63.4	**64.4**	61.8	34.2	**2.8**	2.3	1.4	0.0	**69.5**	**72.4*****	65.5	16.5*k*	13.6*k*	331	272	243	1,482*k*	1,856*k*
retroboard	85.9	**88.1**	85.3	82.2	83.1*	82.3	51.5	0.0	0.0	0.0	0.0	**84.0**	**86.0*****	81.5	23.0*k*	19.8*k*	462	396	352	3,106*k*	3,930*k*
PetClinic	**69.2**	**69.8**	65.7	49.1	46.8	44.1	0.0	**3.7**	3.1	3.0	0.0	**66.9**	**67.2**	62.2	12.4*k*	12.6*k*	248	253	356	1,544*k*	1,607*k*
PetClinic	**69.2**	**69.8**	65.7	49.1	46.8	44.1	0.0	**3.7**	3.1	3.0	0.0	**66.9**	**67.2**	62.2	17.8*k*	12.6*k*	356	253	248	1,544*k*	1,607*k*
Average	73.0	73.7	69.3	53.8	53.7	52.0	24.7	3.2	3.0	2.8	0.3	69.8	70.6	64.9	18.7*k*	15.5*k*	375	310	283	2,397*k*	2,805*k*

Manual POs (RQ$_4$). Table 4 presents the results obtained when manually defined POs are utilized for test generation.

For all subjects, both tools did not cover the entire navigational model within the given time budget (30 minutes). This is motivated by the higher number of methods contained in the manual POs, which model the web applications more accurately, at the cost of making test generation more challenging.

Table 5 shows reports information about the size, in lines of code (LOC), of the POs of our study, as well as the number of methods they contain. Methods determine the transitions, hence the complexity, of the navigational model. The manually generated POs contain, on average, 72% more LOC (Column 4) and 127% more transitions (Column 7) than Apogen's POs.

For 3/6 subjects (*dimeshift*, *phoenix*, and *PetClinic*), we observed no significant difference between using automated or manual POs in terms of branch coverage and fault detection (see results of DIG in Table 3 and Table 4).

Overall, when manually defined POs are available, DIG outperforms SubWeb with statistical significance in terms of state coverage in all cases, and in 4/6 cases concerning branch coverage.

RQ$_4$: *By using manually-defined POs, the diversity-based approach outperforms the search-based approach in terms of structural coverage reached by the generated test suites.*

4.5 Threats To Validity

Using a limited number of subject systems in our evaluation poses an *external validity* threat, in terms of generalizability of our results. We tried to mitigate this threat by choosing six subject systems developed with real-world JavaScript frameworks and pertaining to different domains, although more subject systems are needed to fully address the generalization threat.

Threats to *internal validity* might come from confounding factors of our experiments. We compared all competing algorithms under identical parameter settings (e.g., time budget intervals), on real-world web applications. The manual PO development task poses a threat to validity that we tried to mitigate by following a rigorous, systematic procedure.

Table 5: Manually vs Automatically generated POs

	PO Size (LOC)			# PO Methods		
	Apogen	Manual	Incr (%)	Apogen	Manual	Incr (%)
dimeshift	511	760	49	35	72	105
pagekit	567	2,030	258	42	214	409
splittypie	492	560	14	31	44	42
phoenix	324	482	49	24	38	58
retroboard	292	350	20	26	29	11
PetClinic	312	450	44	20	47	135
Average	416	772	-	30	74	-

Conclusion validity is related to random variations and inappropriate use of statistical tests. To mitigate these threats, we ran each experiment 15 times and used the non-parametric Mann-Whitney U test for statistical testing.

With respect to *reproducibility* of our results, the source code of DIG and all subject systems are available online [52], making the evaluation repeatable and our results reproducible.

5 DISCUSSION

5.1 Effectiveness and Automation

By combining POs and test generation, diversity- and search-based approaches achieved substantially higher state coverage, code coverage and fault detection than a state-of-the-art crawling-based, PO agnostic approach.

DIG is fully automated while SubWeb is only semi-automated as manual preconditions need to be specified. Our results show that DIG can potentially save testers a considerable amount of time by generating both POs and test cases automatically. If necessary, testers can refine the generated POs with missing actions or transitions, and repeat the test generation.

Additionally, the test suites generated by our approach are based on the page object design pattern, which brings known advantages in terms of maintainability [23].

5.2 Efficiency

Our efficiency results demonstrate that our approach is preferable, especially in settings where the time devoted to testing is strict or when test cases are executed very often during the development process. Indeed, the diversity-based approach achieved high coverage and fault detection scores substantially earlier than the search-based approach, regardless the POs being used. This gives us confidence in the applicability of our technique in modern software development processes such as XP and DevOps.

5.3 Benefits to Feasibility

Under the conjecture of contiguous infeasibility regions, promoting diversity is beneficial not only to a thorough exploration of the application behaviours, but also to the feasibility of automatically generated test cases. The search-based approach, on the contrary, uses the guards in the navigational model explicitly to guide the search towards inputs that satisfy them.

5.4 Comparison with Manual POs

The POs automatically generated by Apogen are usually simpler than those developed manually, in terms of number and complexity of actions being exposed for testing. Apogen creates methods based on the actions statically extracted from each test state. As such, in most cases, the resulting methods may miss complex interactions that are possible on the web GUI. Additionally, due to a transition minimization strategy, Apogen does not create reusable components for repeated headers (such as menu bars). Thus, the number of possible test paths/cases that can be generated by DIG is lower.

Despite such limitations, automated POs are competitive with manual POs in a subset of the considered applications. An interesting option available to testers could be the refinement of automatically generated POs, to achieve the same performance of manual POs at a lower development cost. Overall, our empirical results show that when high quality POs are available, our diversity-based approach outperforms all other approaches both in terms of effectiveness (state and branch coverage) and efficiency (rate at which coverage is reached).

6 RELATED WORK

Automated test case generation for web applications is a challenging and extensively researched activity, as highlighted by the high number of papers proposed in literature only in the last few years [3, 5, 33, 35, 37, 56]. The work presented in this paper instantiates concepts from adaptive random testing [8, 10] and diversity-based test generation [1, 13] in the context of web testing. Specifically, it provides a web-specific diversity-based algorithm, and defines ad-hoc distance measures and test generation heuristics.

The main existing model-based works have been already presented and discussed in Section 2.5. On the same category, FeedEx [38] uses DOM and path diversity to guide a crawler towards the generation of smaller yet more accurate testing models. However, FeedEx is not a test case generator, hence a direct comparison with DIG is left for future work.

Yu et al. [56] propose an incremental two-steps algorithm implemented in the prototype tool *InwertGen* in which creation of novel POs and generation of test cases are intermixed. Test cases are generated using the tool *Randoop* [40]. SubWeb [5] proposes a genetic algorithm that performs path selection and input generation at the same time, starting from a manually created PO-based model.

Existing techniques and tools for automated web test generation either ignore path feasibility or require a high number of executions to ensure path feasibility. Indeed, approaches based on crawling or dynamic analysis [33, 35] generate a navigational model and then extract paths from it, without considering that the associated inputs must ensure feasibility.

On the other hand, all approaches [5, 56] that explicitly address path feasibility require the execution of a *huge* number of test case candidates. In fact, they employ random [56] or search based [5] test generation algorithms, both of which assess the feasibility of the candidate test cases by executing them. On the contrary, we assess the quality of the candidates by their diversity, before executing them to determine their feasibility. Our empirical studies show that as a side effect, higher diversity results also in higher feasibility.

Recent papers have considered increasing the robustness and maintainability of web test suites. In order to make test scripts robust, several tools producing smart web element locators have been proposed [4, 24, 25, 55], or to repair them [19, 51] Additionally, Stocco et al. [49, 50] investigate the automated generation of page objects that confine causes of test breakages to a single class, a form of breakage prevention.

Finally, in the mobile domain, Sapienz [31] uses multi-objective search-based testing to automatically explore and optimize test sequences, minimizing length, while simultaneously maximizing coverage and fault revelation. Evodroid [30] is a model-based test generator for Android, based on an evolutionary algorithm that performs a step-wise search for test cases.

7 CONCLUSIONS AND FUTURE WORK

We have proposed a diversity-based approach for web test generation implemented in our tool DIG. DIG can assess a high number of test case candidates without executing them in the browser, making test generation significantly more efficient than state-of-the-art techniques. Differently from search-based approaches, our tool is fully automated, and it does not require any specific guidance to generate feasible test cases.

In our future work we plan to experiment alternative distance metrics, among which those based on information theoretic concepts, such as the normalized compression distance [13]. Moreover, we will also decouple DIG from EvoSuite, so as to further optimize its performance. A further interesting line would be to study the effectiveness of DIG, SubWeb and ATUSA starting from a navigational model produced by a diversity-oriented tool such as FeedEx. Lastly, we plan to experiment our technique with more subjects, as well as, to run a controlled experiment with human subjects to measure the accuracy of the generated tests

REFERENCES

[1] Nadia Alshahwan and Mark Harman. 2012. Augmenting test suites effectiveness by increasing output diversity. In *34th International Conference on Software Engineering, ICSE*. 1345–1348.

[2] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147.

[3] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of Javascript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 571–580. https://doi.org/10.1145/1985793.1985871

[4] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2015. Synthesizing Web Element Locators. In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, 331–341.

[5] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search Based Path and Input Data Generation for Web Application Testing. In *International Symposium on Search Based Software Engineering*. Springer, 18–32.

[6] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web Test Dependency Detection. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, 12 pages.

[7] Robert V. Binder. 1996. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability* 6, 3-4 (1996), 125–252.

[8] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[9] Tsong Yueh Chen, Hing Leung, and IK Mak. 2004. Adaptive random testing. In *Annual Asian Computing Science Conference*. Springer, 320–329.

[10] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*. ACM, 71–80.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill.

[12] dimeshift 2018. DimeShift: easiest way to track your expenses. https://github.com/jeka-kiselyov/dimeshift. (2018).

[13] Robert Feldt, Simon M. Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 223–233.

[14] Mark Fewster and Dorothy Graham. 1999. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley Longman Publishing Co., Inc.

[15] Martin Fowler. 2013. PageObject. http://martinfowler.com/bliki/PageObject.html. (2013). Accessed: 2018-08-01.

[16] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In 11^{th} *International Conference on Quality Software (QSIC)*, Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo (Eds.). IEEE Computer Society, Madrid, Spain, 31–40.

[17] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (feb. 2013), 276 –291.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI 2005)*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223.

[19] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 751–762.

[20] Istanbul 2018. Istanbul: JavaScript test coverage made simple. https://istanbul.js.org. (2018). Accessed: 2018-08-01.

[21] JS-frameworks 2018. Front-end JavaScript frameworks. https://github.com/collections/front-end-javascript-frameworks. (2018).

[22] O. Koresteleva. 2004. *Nonparametric Methods in Statistics with SAS Applications*. CRC Press, Boca Raton, FL.

[23] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and Tools for Automated End-to-End Web Testing. *Advances in Computers* 101 (2016), 193–237. https://doi.org/10.1016/bs.adcom.2015.11.007

[24] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. IEEE, 1–10.

[25] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process* (2016), 28:177–204.

[26] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2018. PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification And Reliability* 28, 4 (2018).

[27] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[28] Yu Lin, Xucheng Tang, Yuting Chen, and Jianjun Zhao. 2009. A divergence-oriented approach to adaptive random testing of Java programs. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 221–232.

[29] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[30] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[31] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[32] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-Based Testing of Ajax Web Applications. In *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST)*. 121–130.

[33] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-Based Testing of Ajax Web Applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST '08)*. IEEE Computer Society, Washington, DC, USA, 121–130. https://doi.org/10.1109/ICST.2008.22

[34] Phil McMinn. 2004. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.* 14, 2 (2004), 105–156.

[35] Ali Mesbah and Arie van Deursen. 2009. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 210–220. https://doi.org/10.1109/ICSE.2009.5070522

[36] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web (TWEB)* 6, 1 (2012), 3:1–3:30.

[37] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Trans. Software Eng.* 38, 1 (2012), 35–53.

[38] Amin Milani Fard and Ali Mesbah. 2013. Feedback-directed Exploration of Web Applications to Derive Test Models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 278–287. http://www.ece.ubc.ca/~amesbah/docs/issre13.pdf

[39] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. 2017. Detecting unknown inconsistencies in web applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 566–577.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[41] pagekit 2018. Pagekit: modular and lightweight CMS. https://github.com/pagekit/pagekit. (2018).

[42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Trans. Software Eng.* 44, 2 (2018), 122–158.

[43] PetClinic 2018. Angular version of the Spring PetClinic web application. https://github.com/spring-petclinic/spring-petclinic-angular. (2018).

[44] phoenix 2018. Phoenix: Trello tribute done in Elixir, Phoenix Framework, React and Redux. https://github.com/bigardone/phoenix-trello. (2018).

[45] retroboard 2018. Retrospective Board. https://github.com/antoinejaussoin/retro-board. (2018).

[46] Selenium 2018. SeleniumHQ Web Browser Automation. http://www.seleniumhq.org. (2018). Accessed: 2018-08-01.

[47] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In 10^{th} *European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*, Michel Wermelinger and Harald Gall (Eds.). ACM, 263–272.

[48] splittypie 2018. Splittypie: easy expense splitting. https://github.com/cowbell/splittypie. (2018).

[49] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2016. Clustering-Aided Page Object Generation for Web Testing. In *Proceedings of 16th International Conference on Web Engineering (ICWE '16)*. Springer, 132–151.

[50] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal* 25, 3 (Sept. 2017), 1007–1039.

[51] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM.

[52] tool 2019. DIG: Diversity-based E2E web test generator. https://github.com/matteobiagiola/FSE19-submission-material-DIG. (2019).

[53] Arie van Deursen. 2015. Beyond Page Objects: Testing Web Applications with State Objects. *ACM Queue* 13, 6 (2015), 20.

[54] Arie van Deursen. 2015. Testing Web Applications with State Objects. *Commun. ACM* 58, 8 (July 2015), 36–43. https://doi.org/10.1145/2755501

[55] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust Test Automation Using Contextual Clues. In *Proceedings of 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 304–314.

[56] Bing Yu, Lei Ma, and Cheng Zhang. 2015. Incremental Web Application Testing Using Page Object. In *Proceedings of the 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb) (HOTWEB '15)*. IEEE Computer Society, Washington, DC, USA, 1–6. https://doi.org/10.1109/HotWeb.2015.14

[57] Xun Yuan and Atif M. Memon. 2007. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 396–405.

[58] Andreas Zeller. 2017. Search-Based Testing and System Testing: A Marriage in Heaven. In *Proceedings of 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST '17)*. 49–50.