2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00048

Automatic Web Testing Using Curiosity-Driven Reinforcement Learning

Yan Zheng[†,‡,*], Yi Liu[‡,♌,*], Xiaofei Xie[‡,✉], Yepang Liu[♌], Lei Ma[✳], Jianye Hao[†], and Yang Liu[‡]
[†]Tianjin University, Tianjin, China; [‡]Nanyang Technological University, Singapore
[♌]Dept. of Comp. Sci. and Engr., Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, Southern University of Science and Technology, Shenzhen, China; [✳]Kyushu University, Fukuoka, Japan.

***Abstract*—Web testing has long been recognized as a notoriously difficult task. Even nowadays, web testing still heavily relies on manual efforts while automated web testing is far from achieving human-level performance. Key challenges in web testing include dynamic content update and deep bugs hiding under complicated user interactions and specific input values, which can only be triggered by certain action sequences in the huge search space. In this paper, we propose *WebExplor*, an automatic end-to-end web testing framework, to achieve an adaptive exploration of web applications. *WebExplor* adopts curiosity-driven reinforcement learning to generate high-quality action sequences (test cases) satisfying temporal logical relations. Besides, *WebExplor* incrementally builds an automaton during the online testing process, which provides high-level guidance to further improve the testing efficiency. We have conducted comprehensive evaluations of *WebExplor* on six real-world projects, a commercial SaaS web application, and performed an in-the-wild study of the top 50 web applications in the world. The results demonstrate that in most cases *WebExplor* can achieve significantly higher failure detection rate, code coverage and efficiency than existing state-of-the-art web testing techniques. *WebExplor* also detected 12 previously unknown failures in the commercial web application, which have been confirmed and fixed by the developers. Furthermore, our in-the-wild study further uncovered 3,466 exceptions and errors.**

I. Introduction

The past decades have witnessed the unprecedentedly rapid development and innovation of web technologies. Nowadays, web applications have become as powerful as native desktop applications. They are competitively convenient and do not require complicated installation either. However, web applications can be difficult to test due to their complicated business logic implemented in different languages across the client and server side (e.g., HTML, JavaScript, C# and Java). In general, the more web pages are explored with more states covered, the higher the possibility of discovering defects becomes. Hence, various kinds of approaches have been proposed to achieve a sufficient exploration by generating test cases.

Manual designing with the aid of automation frameworks such as Selenium [1] is a useful way to create test cases. The tester is required to create test scripts, simulating user operations (e.g., clicking buttons and filling in forms) on the web application's graphical user interface (GUI) [2, 3, 4, 5, 6, 7, 8, 9, 10]. However, such manual work is labor-intensive and costly, where the testing effectiveness heavily depends on the human testers' domain knowledge. Besides, web applications frequently evolve and the manually written test cases normally require substantial modifications before testing the new versions [11, 12].

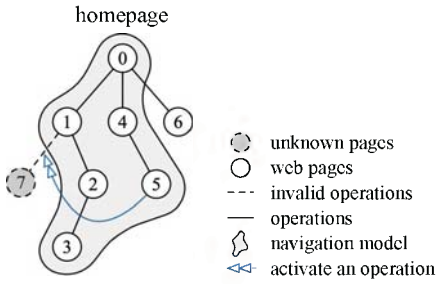

Fig. 1. An intuitive visualization of exploring web pages.

Random-based approaches [13, 14] generate pseudo-random operations to fuzz the web applications. Despite the wide adoption in practical development, the shortcomings of such approaches are obvious. That is, they often create invalid test cases like performing input operations on buttons. Also, the testing is unbalanced and some hard-to-reach web pages may never be explored.

Model-based approaches [15, 16, 17, 18, 19] build a navigation model of the web application under testing, and then generate test cases accordingly by random or sophisticated search strategies. In spite of the guidance of the navigation model, existing approaches still suffer from several limitations. Firstly, the constructed navigation model may cover only a part of the web application, restricting the exploration power of the generated test cases. As depicted in Fig. 1, pages 6 and 7 cannot be tested as they are not included in the navigation model. Also, domain knowledge is usually required in building high-quality models [15]. Besides, the content of web applications is usually dynamically updated (e.g., via JavaScript code execution), which cannot be easily captured by the static navigation models. Secondly, in web applications,

✉ Corresponding author: Xiaofei Xie.
* Yan Zheng and Yi Liu contributed equally to this work, and Yan Zheng is with the School of New Media and Communication in Tianjin University.

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00048

long sequences of actions (e.g., path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$) are often needed to complete certain tasks such as filling in and submitting forms. The business logic of real-world web applications can be arbitrarily complex. For example, page 7 can be visited only when page 5 is properly navigated. It could be challenging for random or search-based strategies [5, 15, 20] to generate effective action sequences.

To address the aforementioned challenges, an effective and end-to-end automatic testing is needed. Recently, reinforcement learning (RL) has demonstrated its potential for learning a policy to test and interact with complicated games [21, 22, 23, 24] or Android applications [25, 26, 27], which provides the possibility of applying RL on automatic web testing. However, existing techniques cannot be easily adapted to test web applications for the following reasons. Firstly, the testing domains are different, which make the RL modeling totally different. For example, the reward function may be different. Game playing usually has concrete goals to achieve (e.g., winning the game or maximizing a score), which is not the case in web testing. The state definition and abstraction are different either. Game playing [23] defines the state based on the outputs of APIs and Android testing [25] uses an existing tool UIAutomator [28] to extract structures as the states, which are both not applicable in web testing. Secondly, one fundamental challenge of RL is how to perform effective exploration especially when the space of the environment is huge [29, 30]. The existing techniques [21, 23, 31] mainly guide the exploration with simple reward functions, which could be ineffective for web applications that have complex business logic and frequent dynamic update. Thus, more effective exploration is needed for RL-based web testing.

Considering the dynamic and highly interactive nature of web applications, an effective model-free web testing technology can be highly desirable. In this paper, we propose a novel web testing framework, named *WebExplor*, which performs an end-to-end automated web testing. *WebExplor* leverages RL to perform an adaptive exploration of web applications and generate high-quality action sequences, which may be prerequisite operations (e.g., filling forms before submission) for discovering new pages. In particular, *WebExplor* performs an on-the-fly testing while constantly training the agent policies (rather than the usual AI solutions that can only be used after training). To achieve both high coverage and efficiency during testing, we first propose the state abstraction based on the HTML pages. Then, we propose a curiosity-driven reward function, which provides low-level guidance for the exploration of RL such that the learned policy could explore more behaviors of the web applications. To avoid falling into local optima, especially when the learning space is huge, we further propose a deterministic finite automaton (DFA) guided exploration strategy that provides high-level guidance for RL to efficiently explore the web applications. In particular, the DFA records the transitions and states visited during the RL exploration and is continuously updated. When RL gets stuck (i.e., cannot find a new state within a given time budget), *WebExplor* selects one path from DFA based on the curiosity and guides RL to explore along this path further. Both the low-level guidance (from the reward function) and the high-level guidance (from the DFA) play important roles in achieving effective web testing. To demonstrate the effectiveness of our technique, we implemented *WebExplor* and conducted a large-scale evaluation on a research benchmark of six real-world projects [15] and a commercial SaaS web application. We also conducted an in-the-wild study of the top 50 web applications in the world [32]. The contributions of this paper are summarized as follows.

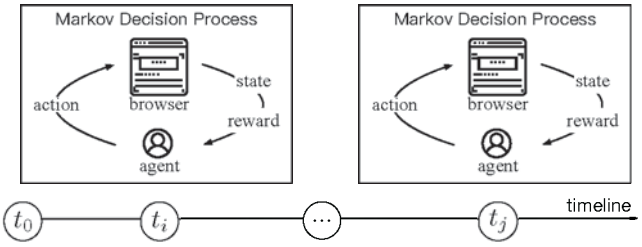

Fig. 2. Web interactions as a Markov decision process.

- We propose a novel web testing framework, *WebExplor*, to efficiently and effectively test real-world web applications. To the best of our knowledge, *WebExplor* is the first end-to-end web testing framework leveraging reinforcement learning.
- We propose a curiosity-driven reward function and a DFA to guide RL to efficiently explore diverse behaviors of web applications.
- We comprehensively evaluate *WebExplor* on six open-source web applications, a commercial web application, and top 50 real-world web applications. In the commercial application, 12 previously unknown failures, including logical and security defects, are discovered by *WebExplor*, and confirmed and fixed by the developers. Furthermore, 3,466 exceptions and errors are discovered in the top 50 web applications.

II. Preliminaries

A. Web Application and Reinforcement Learning

A typical web application requires the end user to input a sequence of actions (e.g., clicks) to interact, which, in turn, will change the web application's states (e.g., URL or GUI). This process can be modeled as a Markov Decision Process (MDP) [33]. MDP can be defined as a 4-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$, where $\mathcal{S}, \mathcal{A}$ represent the sets of states and actions, respectively. As shown in Fig. 2, the agent (tester) interacts with the environment (browser) over the time horizon. At timestamp t, the agent observes the state $s_t \in \mathcal{S}$ of the web applications (e.g., GUI or HTML), and selects an action $a_t \in \mathcal{A}$ to execute, after which, the agent receives an immediate reward $r_t = \mathcal{R}(s_t, a_t)$, and the environment can change to a new state $s_{t+1} \sim \mathcal{P}(s_t, a_t)$. The $\mathcal{R}(\cdot)$ and $\mathcal{P}(\cdot)$ are reward function and probability transition function, both of which depend only on preceding s_t and a_t.

Fig. 3. The action sequence of adding a new owner.

Intelligently, the agent selects an action $a \sim \pi(s)$ to execute according to a probabilistic policy function $\pi(\cdot)$. The agent interacting with the environment gives rise to a trajectory as follows:

$$traj = (s_0, a_0, r_0, \cdots, s_t, a_t, r_t, \cdots), \tag{1}$$

where the subscripts denote different timestamps over the finite time horizon. Each trajectory has a return, defined as $\sum_{t=0}^{T} \gamma^t r_t$, where rewards are discounted by a factor $\gamma \in [0, 1)$. In general, RL aims at finding one optimal trajectory with the maximum return rather than diverse trajectories. However, this is often not the goal of web testing, which seeks to explore diverse trajectories and states. Therefore, an adaptive reward function is required to guide RL to continuously generate diverse trajectories, which will be detailed in Section III-C.

B. Problem Formulation

In general, given a web application, the goal of testing is to generate action sequences along with suitable inputs in order to explore diverse states and behaviors of the application, potentially covering more logical application scenarios [15].

*Definition 1 (**Web State**):* A state s is a description of a web application's current status (e.g., the HTML page).

From a human perspective, the image (i.e., screenshot) that captures the changes of the HTML page is a natural representation of web states. However, due to the wide use of animations, images may not reliably represent web states (e.g., two completely different images may correspond to the same state of a web application). In comparison, the HTML page's code is a more precise representation as it encodes the URL and the structural characteristics of HTML pages. Thus, we propose a novel state representation by analyzing the HTML page's code (in Sec III-B). Unless stated otherwise, a concrete HTML page instance is referred to as the state.

*Definition 2 (**Action**):* An action a is a valid operation in a given web state (i.e., an HTML page).

Given a web state, we focus on the operable DOM elements (e.g., links, buttons or input boxes), on which the operations may result in changes of application status (e.g., submitting a form or URL jumping). It is worth mentioning that, different states may contain different action DOM sets. For example, in Fig. 3, the homepage has 5 valid actions (i.e., elements in the navigation bar) while the "add-owner" page has 6 more valid actions (input boxes and a button).

*Definition 3 (**Test Case**):* A test case is a sequence of actions $(a_0, \cdots, a_t, \cdots)$ with necessary input values.

For example, in Fig. 3, the "add-owner" function consists of three parts: navigating to the adding page, filling the form and clicking the submit button. One feasible action sequence (in red) for testing the "add-owner" function is visualized in Fig. 3. This sequence together with necessary inputs constitute a test case $(a_0, \cdots, a_7)$. When operating on an inputtable element, our technique will provide a suitable value to make a test case continue, which will be detailed in the Section III-B.

Similar to the existing work [15], we say a test case fails if exceptions or errors are thrown during the execution of the test case, including the JavaScript runtime exceptions, client errors or server errors, which can be analyzed via the returning status code [34]. For simplicity, we refer to the exceptions and errors as failures in the subsequent sections.

*Definition 4 (**Web Testing**):* For a web application, the tester aims at learning an adaptive policy π to continuously generate test cases for web exploration and failure detection.

Intuitively, the more states are uncovered, the higher the chance of finding failures is. Consequently, the goal of web testing is to generate test cases that can reach more diverse states, on which failures may be discovered. It is worth emphasizing that discovering test cases that can trigger business logic (e.g., creating data) is critical for testing, as it may be a prerequisite for discovering new scenarios (e.g., editing data).

III. Automatic Web Testing

A. An Overview of WebExplor

In a nutshell, *WebExplor* is an end-to-end framework aiming at achieving automatic web testing in an online fashion. Its goal is to automatically generate diverse sequences of actions to explore more behaviors of the web application under testing. To achieve this, *WebExplor* leverage curiosity-driven reinforcement learning (RL) to constantly optimize a policy, which can generate diverse test cases. In particular, the RL training and web testing are intertwined, which is different from usual AI solutions that performs training before deploying. Fig. 4 shows an overview of *WebExplor*, which comprises three major components. ❶ The *pre-processing* component maps an HTML page to an abstract state. Its main purpose is to avoid the state explosion caused by dynamic updates in a web page, such that a good policy can be learned effectively. ❷ The *curiosity-driven policy learning* component is designed for learning a policy that could explore diverse states of the web application. ❸ The *DFA-guided exploration* component further improves the efficiency of the exploration of RL by maintaining a continuously updated deterministic finite automaton (DFA) that records all visited states and their frequencies. When RL cannot discover new states within a certain time budget, *WebExplor* selects one novel state as the starting point of the next exploration based on the global information of DFA, so as to avoid being trapped around the local optima.

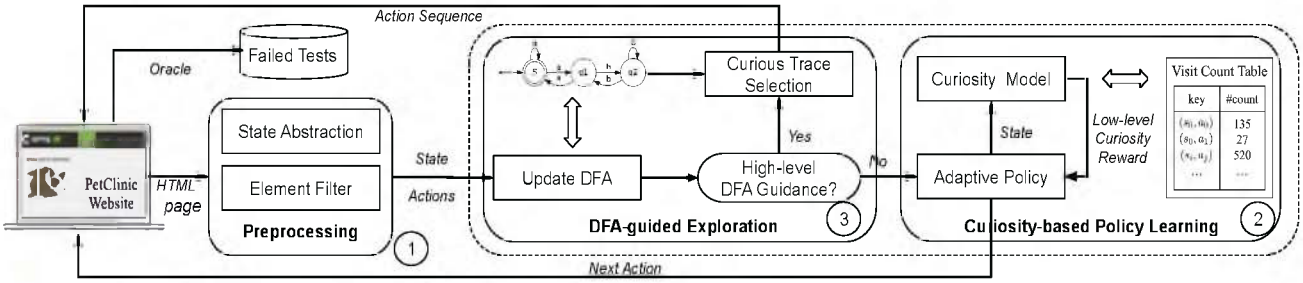

Fig. 4. The Workflow of *WebExplor*.

Algorithm 1 presents the details of our approach. *WebExplor* takes the target web application env and the pre-processing function ϕ as the inputs and outputs a set of failed test cases F. *WebExplor* first initializes the policy π, the DFA M, and the failed test set F (Line 1). Then, it continuously tests the web application until the pre-defined time budget exhausts (Line 2). During testing, to avoid reaching "stuck webpages" that cannot jump to other pages and continue further, we limit the maximum number of steps for one test case (Line 7). After reaching the maximum number, we reset the web application: jump to the default homepage p' (Line 3), set the initial action sequence, which includes only an empty action ϵ (Line 4), and gets the initial state s' (Line 5). Each test case starts from the initial state, i.e., the homepage (Line 6). *WebExplor* performs on-the-fly testing by executing the current action sequence act on the current page p' (Line 8) (e.g., submitting a form). During the execution, *WebExplor* monitors the status of the browser's console such that failures can be captured automatically. The environment returns a new HTML page p and the error status. If an error is found, the test case is added into the failed test set (Line 10). *WebExplor* encodes the current page and returns the corresponding state s and valid actions va in the state s' (Line 11). If *WebExplor* cannot identify a new state within a certain amount of time, the RL may enter into the local optima. *WebExplor* checks the DFA d that records the global visit information, and selects one trace that is less visited (Line 14). It returns the trace t and the action sequence act, from which the trace can be restored. The web application is then reset to the homepage (Line 15) and the number of the current steps is set with the length of trace t (Line 16).

After a state s is explored, *WebExplor* calculates the curiosity reward r (Line 18). The policy π is trained with the current transition (s', a, s) and its reward r (Line 20). In addition, the DFA d and the current test case are updated with the transition (Lines 21–22). The next action is selected by feeding the current state s to the policy π (Line 23). Then, the previous state and HTML page are updated (Lines 24–25).

Algorithm 1: *WebExplor*

Input : The target web application env, the pre-processing function $\phi(\cdot)$
Output: The set of failed test cases F

1 Initialize policy π, DFA M and test case set $F = \emptyset$
2 **repeat**
3 $p' := reset(env)$
4 $act := [\epsilon]$
5 $s' := \phi(p')$
6 $t := [s']$
7 **repeat**
8 p,*failed* $:= env(p', act)$ ▷ on-the-fly testing
9 **if** *failed* **then**
10 $F := F \bigcup \{t\}$
11 ❶ $s, va = \phi(p)$ ▷ see Algorithm 2
12 update valid actions of $\pi(s)$ using va
13 **if** *no curious state within some time* **then**
14 ❸ $act := selectTrace(M)$ ▷ see Algorithm 3
15 $p' = reset(env)$
16 update the number of the current steps with the action sequence act
17 **continue**
18 ❷ calculate $r = curiosity(s', a, s)$
19 $a := act[-1]$
20 train policy π using (s', a, r, s) ▷ Q-learning
21 update DFA d using transition (s', a, s)
22 $t.append(a, s)$ ▷ store state-action sequence
23 $act := [\pi(s)]$
24 $s' = s$
25 $p' = p$
26 **until** *reach maximum steps*;
27 **until** *until time budget exhausts*;
28 **return** F

B. Pre-Processing

For policy learning via reinforcement learning, we need to define the state representation. A straightforward way is to leverage web page representations (e.g., GUI or HTML document). However, if one adopts such a method, the number of states can be quite large and even infinite due to the dynamic nature of web applications. For example, HTML documents can be different if the user operates differently (e.g., different form values or infinite scrolling pages). Thus, adopting HTML document as states often suffers from the state explosion problem, resulting in low effectiveness of RL [35]. To overcome such a limitation, we propose a novel state representation. The intuition is that HTML pages that focus on the same business logic should be consolidated into one state. For example, in a webpage, the content of a table may

Algorithm 2: Pre-processing ϕ

Input : HTML page $p = \langle url, html\_doc \rangle$
Output: State s, Valid action set va

1 Let S be the current state set of the web application
2 $va := retrieveValidElement(html\_doc)$
3 **for** $s \in S$ **do**
4 Fetch $\langle url', html\_doc' \rangle$ from the existing state s
5 **if** $url \neq url'$ **then**
6 **continue**
7 $sim := computeSimlarity(html\_doc, html\_doc')$
8 **if** $sim > threshold$ **then**
9 **return** s, va
10 Create new state s using $\langle url, html\_doc \rangle$
11 $S := S \bigcup \{s\}$
12 **return** s, va

be updated constantly (e.g., by adding or removing items). Although the HTML document may vary a lot, the pages still look similar and handle the same user interactions. We do not treat such pages as different states.

Given an HTML page, we use its *URL* and *HTML document* as the approximation of the business logic. It is intuitive that if two pages have the same URL and their HTML documents are very similar, they are more likely to focus on the same business logic. We argue that such similar pages represent the same state, and Algorithm 2 describes how to distinguish different pages. The basic idea is to calculate the HTML structure similarity of two pages. If the structure similarity (via tag-wise comparison [36]) is above a threshold, it is more likely that they focus on the same business logic and should be considered as the same state. Algorithm 2 takes the code of the HTML page as the input and outputs the state s as well as a set of valid actions va in the current page. We use S to represent the existing states during testing.

We adopt the browser built-in protocols [37] to filter some elements (e.g., no rendering or invisible) and only keep the ones that can be operated on, i.e., valid actions on this page (Line 2). These elements include the clickable buttons, links, input boxes, selectors. Next, we check the similarity between the current page p and pages in the previous states until one similar state is found. Specifically, for each previous state s, we obtain its page information (Line 4) and calculate the similarity between p and the page in s as follows. Firstly, we compare their URLs. Intuitively, if the URLs are not the same, the pages often tend to execute different business logic. Hence, we do not count them to be the same state (Line 5). Otherwise, we calculate the similarity between the two HTML documents (Line 7). More specifically, we convert an HTML document to a sequence of tags and adopt the gestalt pattern matching [36] algorithm to calculate the similarity between two sequences. If the similarity is above a pre-defined *threshold*, the previously existing state s and va are returned (Line 9). Note that, we extract all tags in the HTML document without any filtering so that no feature information in HTML will get lost. If the current page does not match any existing states, we create a new state s using the current page's code (Line 10), add it into S and return the results. It is worth noting that different URLs may represent the same business-logic, creating multiple states corresponding to the same business-logic (Line 5). However, such a correspondence causes little performance loss and doesn't affect *WebExplor*'s soundness.

Meanwhile, *WebExplor* focuses on generating action sequences rather than input values, even though input values can also affect the testing procedure. To be aligned with prior works [15, 38] and carry out testing procedure, when operating on inputtable elements, random values will be generated according to the W3C standards [39]. Note that, *WebExplor* can leverage dictionaries or user-specified values to enhance the testing capability, which will be studied as the future work.

C. Testing via Curiosity-Driven RL

WebExplor leverages RL to achieve an end-to-end testing by directly interacting with the web applications. Specifically, the purpose is to learn a policy (i.e., π) that provides an exploration strategy to generate diverse test cases. To achieve such a policy, we need to define an effective reward function that determines the optimal policy.

Reward function. Common RL tasks (e.g., game playing) usually have a concrete goal such as winning a game or achieving a high score, which eases the design of the reward functions [40, 41]. However, in web testing, reward function design becomes challenging as the goal is vague, i.e., to explore as many different behaviors of the web applications as possible. Moreover, a web application can be dynamically updated, indicating that the goal should also be dynamically adjusted. To address the challenge, we leverage the notion of curiosity, which has been proposed to counter the problem of coarse reward in RL [42, 43]. Specifically, we have devised a curiosity-driven reward function that adopts a general and adaptive mechanism to guide the exploration such that diverse states could be reached. For curiosity measurement (Line 18 in Algorithm 1), during testing, *WebExplor* maintains a visit count table to record the number of each transition (denoted by $N(s', a, s)$). The curiosity is measured by MBIE-EB [44]:

$$curiosity(s', a, s) = \frac{1}{\sqrt{N(s', a, s)}}. \quad (2)$$

$N(s', a, s)$ is initialized to 1. Each time when the state s' transits to s by performing the action a, the corresponding $N(s', a, s)$ is increased by 1.

Q-Learning. *WebExplor* leverages a model-free RL algorithm Q-learning [45] to optimize the policy with the curiosity-driven reward. Q-learning has a function $Q : S \times A \rightarrow \mathbb{R}$, which returns the Q-value for a state-action pair. Each time a new state s is reached from the previous state s' (i.e., (s', a, s)), we update the Q function (Line 20 in Algorithm 1):

$$Q(s', a) = curiosity(s', a) + \lambda max_{a'} Q(s, a'), \quad (3)$$

where $\lambda \in [0, 1]$ is a discount factor. The Q function keeps the temporal relations between actions since the Q-values will propagate to the ones in antecedent states recursively.

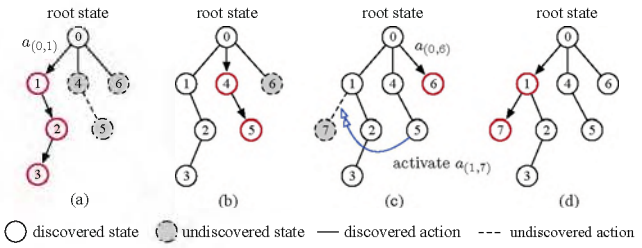

Fig. 5. An intuitive illustration of curiosity model.

Based on the Q function, the policy π measures the weights of the valid actions in a state s using the Gumbel-Softmax method [46]:

$$p(a) = \frac{\exp(\frac{Q(s,a)+g_a}{\tau})}{\sum_{a_i \in \mathbf{A}} \exp(\frac{Q(s,a_i)+g_i}{\tau})} \tag{4}$$

where $\mathbf{A}$ is the valid action set at the state s, $\tau = 1$ is a temperature coefficient, and $g_{(\cdot)}$ are i.i.d noise sampled from Gumbel$(0, 1)$ distribution. An action with a higher Q-value is more likely to be selected for interaction (Line 23 in Algorithm 1).

This enables *WebExplor* to balance between exploration and exploitation. Actions that discover a new state are given a high curiosity reward and are more likely to be selected for the execution. This trait ensures efficient exploitation as such actions have high possibility in discovering diverse behaviors. In the meantime, the curiosity decreases along with the action execution, making other less executed actions to be selected for execution. This facilitates sufficient exploration and helps to find complex business logic in the web application, as we will demonstrate in our evaluation.

Example. Fig. 5 illustrates how the curiosity-based RL works for web testing and how it can explore complex business logic. Assume that *WebExplor* starts from the root state s_0 and different action $a \sim \pi(s)$ can be selected for execution. Executing actions cause state transitions, e.g., $a_{(0,1)}$ results in $s_0 \rightarrow s_1$. Initially, the probability of choosing $a_{(0,1)}$, $a_{(0,4)}$, and $a_{(0,6)}$ are the same. Assume that s_3 is firstly covered through the red path (Fig. 5(a)), then the value of $Q(s_2, a_{(2,3)})$, $Q(s_1, a_{(1,2)})$, and $Q(s_0, a_{(0,1)})$ will be updated via back-propagation through the path. In this way, the temporal relations (along this path) is built and encoded in the policy π. The curiosity reward $curiosity(s_0, a_{(0,1)}, s_1)$ is decreased. In Fig. 5(b) and Fig. 5(c), actions $a_{(0,4)}$ and $a_{(0,6)}$ (with higher curiosity) are selected. Note that, after s_6 is reached, some previously invalid actions (e.g., $a_{(1,7)}$) may become valid due to the specific business logic. For example, an item in the table can only be deleted after being added. In this case, as $curiosity(s_0, a_{(0,6)}, s_6)$ decreases, $a_{(0,1)}$ and $a_{(0,4)}$ regains the same chance to be selected, which is critical for exploring the newly activated states (i.e., s_7), opening a new untouched area to be explored.

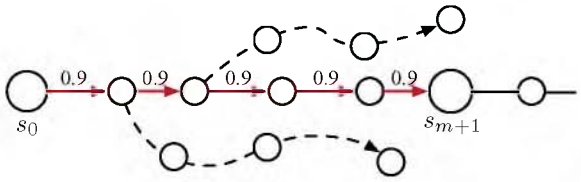

Fig. 6. An intuitive illustration of transition in long path.

Algorithm 3: selectTrace

Input : DFA M
Output: Action set act
1 $(s_m, a_m, s_{m+1}) := \arg\max_{(s',a,s)\in\delta}(curiosity((s', a, s)))$;
2 Find the shortest trace
$tr := (s_0, a_0, s_1, a_1, \ldots, s_m, a_m, s_{m+1})$;
3 **return** $(a_0, a_1, \ldots, a_m)$;

D. DFA-Guided Exploration

Exploration is widely regarded as one of the most challenging problems of reinforcement learning [33, 42, 43]. In the web application, a function is usually triggered by executing the actions in the specific order (e.g., the approval process in the Office Automation (OA) system). The exploration becomes more challenging as the sequence of actions gets longer. Although the curiosity-driven reward function provides guidance for the action selection, due to its stochastic nature, RL may still have low probability to select other actions especially in a long sequence of actions, interrupting the testing of the target function. Consider the example in Fig. 6, a policy π with 0.9 probability of selecting right actions (red arrow) to state s_{m+1}. However, the possibility of reaching s_{m+1} is only $(0.9)^5 = 0.53$ since the path can be interrupted whenever the policy takes an action. The longer a path becomes, the more frequent an interruption may occur, making reaching a desirable transition harder, especially when facing a long path.

To address this challenge, we propose to build an on-the-fly deterministic finite automaton (DFA) during the testing, which provides a high-level guidance for boosting the RL exploration. Specifically, if new states cannot be found after some time, *WebExplor* starts to find a transition, which has the highest curiosity, from the DFA. Then, the shortest path that can reach the transition is detected such that the RL could directly reach this transition.

*Definition 5 (**DFA**):* A deterministic finite automaton (DFA) M is a 5-tuple (S, A, δ, s_0, F), where S is a finite set of states, A is a set of actions, $\delta : S \times A \rightarrow S$ is a set of transitions, s_0 is the initial state and F is a finite set of states that cannot transit to other states.

During testing, once a new transition (s', a, s) is explored, the DFA will be updated, i.e., $\delta := \delta \bigcup \{(s', a, s)\}$ (Line 21 of Algorithm 1). Algorithm 3 presents the basic idea of curiosity-driven trace selection. From the DFA, we first select the transition with the highest curiosity (s_m, a_m, s_{m+1}). Then, we adopt the Dijkstra's algorithm [47] to identify the shortest trace tr that can reach the target (s_m, a_m, s_{m+1}). With this trace, RL

could directly restore to the target transition. Intuitively, some transitions could be very deep and thus difficult to reach by RL. For these transitions, *WebExplor* leverages DFA to further enhance the exploration efficiency and testing effectiveness.

Theoretically, the non-deterministic finite automaton can represent the dynamicity of the stochastic environment more accurately. However, we use DFA due to the following reasons: 1) the automaton is used to guide the selection of one viable path for exploration. Due to dynamic factors (e.g., network), one path in DFA may be infeasible, but it doesn't affect the soundness of *WebExplor* because the dynamic execution will ignore such infeasible paths; 2) The construction of non-deterministic finite automaton could be more expensive, especially on estimating the transition probabilities. Considering the trade-off between efficiency and granularity of automaton construction, DFA is a good-enough solution for *WebExplor*.

IV. Empirical Evaluation

We have implemented *WebExplor* based on Python 3.7.6 and PyTorch 1.5.0 [48] with more than 5,000 lines of code[1]. To demonstrate the effectiveness and efficiency of *WebExplor*, we conduct an empirical evaluation investigating the following four research questions.

- **RQ1 (Code Coverage):** How is the exploration capability of *WebExplor* in terms of code coverage?
- **RQ2 (Failure Detection):** How effective is *WebExplor* for detecting failures of web applications?
- **RQ3 (DFA Guidance):** How effective is DFA in guiding the exploration during testing?
- **RQ4 (Scalability):** How effective is *WebExplor* in testing real-world web applications?

A. Experiment Setup

1) Benchmarks: Our large-scale evaluation uses three benchmarks, including a research benchmark from the prior work [15] to compare *WebExplor* with the state-of-the-art techniques, a benchmark of top 50 real-world web applications [32] to evaluate the scalability of *WebExplor*, and an industrial web application to conduct a detailed case study.

- **Research benchmark:** We adopt a benchmark containing six popular GitHub projects (each has more than 50 stars) from the prior work [15]. These projects use six most popular JavaScript frameworks: dimeshift (Backbone.js), pagekit (Vue.js), Splittypie (Ember.js), phoenix-trello (Phoenix/React), Retroboard (React), and PetClinic (AngularJS).
- **Real-world web applications:** According to the ranking [32], we select the top 50 web applications in the world for evaluation. To investigate the scalability, we directly leverage *WebExplor* for an end-to-end testing of these applications without fine-tuning.
- **Industrial web application:** A complex Software as a Service (SaaS) system is adopted for the further case studies. We omit the system name for anonymous review reasons.

[1]More details can be found in our website [49].

2) Web application failures: In subsequent experiments, we collect the system-level failures (defined in Section II-B) reported in the browser's console to study the failure detection capability of related approaches. Note that user-level failures may or may not cause system-level failures, which depends on the system's robustness. For example, if user-level failures are well handled by the web-system (e.g., strict input-field checking), no system-level failures will occur. Otherwise, failures will be triggered and captured by *WebExplor*. For identifying the root-cause of failures (e.g., by users or system), we adopt manual analysis. It is worth emphasizing that all discovered failures are manually vetted to ensure that the thrown exceptions and errors are actually failures (i.e., no false alarms).

3) Baselines Approaches: To evaluate the effectiveness of *WebExplor*, we select three state-of-the-art approaches as baselines for a comparative study. These baselines include both the model-based to model-free algorithms. Besides, one random strategy that adapts the idea of Monkey [13] is adopted as a baseline. Moreover, to evaluate the advantage of leveraging DFA, a variant of *WebExplor* is also implemented for the ablation evaluation.

- DIG [15] is a navigation model-based approach, leveraging a diversity-based test case generator for web testing.
- SUBWEB [5] is a navigation model-based approach, considering the uncovered branches and using a search-based strategy to achieve web testing.
- Crawljax [14] is a navigation model-free approach, discovering and clustering pages on the fly and adopting a crawling-based random test case generator for web testing.
- Random [13] is a model-free approach, randomly selecting one of the available actions to explore web states.
- *WebExplor* (no DFA) is a variant of *WebExplor* without the DFA guidance.

4) Configurations: For all experiments, we give each tool the same time budget (i.e., 30 minutes). To counteract the randomness from a statistical perspective, we repeat each experiment 15 times and calculate the average results. For the similarity in the pre-processing, we set 0.8 as the threshold. DFA provides high-level guidance for *WebExplor* if no new states are discovered in 2 minutes. For the discount factor in RL, we set $\lambda = 0.95$ in all experiments. We conduct an comprehensive evaluation in spending more than 300 CPU hours, i.e., 6 projects * 7 settings (for 5 tools) * 0.5 (hour time budget for single round) * 15 repetitions in total. Besides, during testing, actions that lead to external links (via domain checking) will be recorded during testing, marked as invalid actions, and not executed in the subsequent testing.

B. Code Coverage (RQ1)

To conduct a comprehensive comparison, for DIG and SUBWEB, we use the naviation models, which are based on automatically and manually generated page objects (denoted by APO and MPO), respectively. To counteract implementation bias, both the APO and MPO are directly adopted from the prior work [15]. Comparisons in terms of the branch coverage

of JavaScript code are conducted on six web applications. The averaged results of 15 runs are summarized in Table I, where bold numbers indicate the best result. Overall, we have the following findings.

The model-based methods DIG and SUBWEB can achieve an overall better branch coverage than model-free methods Crawljax and Random in most cases. This is because navigation models can provide more information (e.g., web structure), which is beneficial for effective testing. However, a counterfactual finding is that model-free *WebExplor* achieves competitive performance in terms of code coverage, significantly outperforming (i.e., calculated by Mann-Whitney U test [50] at 0.05 confidence level) model-based methods in 4/6 web applications (bold numbers in Table I). It not only demonstrates the exploration capability of *WebExplor* in terms of the code coverage, but also the robustness, which could be obtained by the model-free testing fashion.

Besides, we perform an in-depth analysis on the test cases generated by *WebExplor* to figure out why *WebExplor*, as a model-free algorithm, can achieve better performance than other model-free algorithms (i.e., Crawljax and Random). Take Petclinic for an example (shown in Fig. 3), we find that *WebExplor* can generate the correct operation sequence (i.e., filling form values before adding an owner) to achieve effective testing. Normally, generating such logically related sequence actions is hard for random-based model-free algorithms. However, by leveraging the curiosity-driven RL, *WebExplor* can capture such relations and encode this "knowledge" in the policy to create effective test cases without navigation models.

Furthermore, compared to model-based algorithms (i.e., DIG and SUBWEB), we found that *WebExplor* performs much better in four subjects and similarly for the rest. We investigate the reason and find that some pages in Splittypie and Retroboard need complex inputs that are difficult to generate randomly. For instance, the "transaction" page in Splittypie can only be discovered after typing in a time value with a *non-standard* W3C format (e.g., mmdd). However, *WebExplor* follows W3C standards [39] and cannot generate such inputs without human knowledge. Meantime, we analyze APO and MPO in DIG and SUBWEB, and find that both kinds of navigation models have been manually fine-tuned with human-knowledge, which enables non-standard input generation for higher coverage. Detailed results and analysis can be found in [49]. In rest four subjects, such non-standard inputs barely exist, where *WebExplor* achieves much better results.

Answer to RQ1: In contrast to model-based approaches, *WebExplor* achieves better code coverage and robustness in most cases with no navigation models or prior knowledge.

C. Failure Detection (RQ2)

We continue to analyze the ability of each baseline in discovering failures (defined in Section IV-A2). Table I shows the statistical results of the average number of failures discovered during the allocated testing time (see Section IV-A4). As one failure can be discovered multiple times during the entire testing process, here we only count the unique failures for all methods. Overall, we have the following findings.

First, compared with other baselines, random approach achieves relatively poor performance, which is consistent with the intuition that simple random exploration is ineffective for large web applications. Model-based approaches can discover more failures than the random approach but exhibit instability.

Among all baselines, *WebExplor* discovers the most number of failures (bold numbers in Table I), significantly exceeding other methods (i.e., calculated by Mann-Whitney U test at 0.05 confidence level). This reveals not only the competitive performance in failure detection, but also generality of *WebExplor*.

Another counterintuitive finding is that, in Splittypie and Retroboard subjects, although *WebExplor* achieves lower code coverage than DIG (in Table I), it still detects more failures (in Table I). We analyze the test cases generated by *WebExplor* and DIG, compare the logs (see detailed logs in [49]) and find that *WebExplor* discovers more server request errors (e.g., error code 400 and 500) via generating test cases with illegal operation sequences. Intuitively, the code related to each operation can be easily covered by independent execution. However, some failures are only triggered by illegal operation orders, indicating that it seems ineffective to detect failures by simply improving code coverage. Moreover, DIG combines a sequence of actions with a specific order as a macro operation in both APO and MPO, ignoring the fact that different orders or executing only part of the sequence may result in potential failures. This explains why *WebExplor* performs better in generating effective test cases.

Answer to RQ2: Compared to other baselines, *WebExplor* achieves the best performance in failure detection, while requiring no human knowledge or fine-tuned navigation models, revealing its potentials across different subjects.

D. DFA Guidance (RQ3)

This section investigates how DFA contributes to boosting the testing efficiency through high-level guidance. Comparisons between *WebExplor* and *WebExplor* (no DFA) are conducted to evaluate the failure detection rate and efficiency. Fig 7 illustrates the results, where the x-axis is the testing time and y-axis (#Failures and #Coverage) are the average number of discovered failures and code coverage rate, respectively. To avoid statistical bias, the results are averaged using 15 runs, and the bold lines and shadow areas represent the mean and standard deviation.

First, in Fig. 7 (top), we observe that *WebExplor* can not only discover more failures than the one without DFA, but also a higher failure detection efficiency (blue line rises faster). Meanwhile, DFA achieves an early performance jump regarding the number of discovered failures, especially in the dimeshift, pagekit and petclinic subjects. All the advantages benefit from a better exploration guided by DFA. Take the petclinic subject for example, many failures discovered by *WebExplor* have long execution traces, containing many actions that need to be executed in a specific order. Consider

TABLE I

COMPARISONS OF RELATED BASELINES REGARDING THE AVERAGED BRANCH COVERAGE AND FAILURE DETECTION WITH THE CORRESPONDING STANDARD DEVIATION. (VALUES IN BOLD INDICATE THE BEST AVERAGE RESULTS USING 15 RUNS.)

Subjects	Average Branch Coverage (%)							Average Unique Failures (#)						
	WebExplor	Crawljax	Random	DIG		SUBWEB		*WebExplor*	Crawljax	Random	DIG		SUBWEB	
	Navigation model-free			APO	MPO	APO	MPO	Navigation model-free			APO	MPO	APO	MPO
Dimeshift	**51.0** (2.7)	11.8 (0.0)	24.1 (12.8)	38.5 (1.6)	40.1 (1.6)	36.7 (2.3)	38.9 (1.8)	**9.0** (1.7)	1.0(0.0)	1.4 (0.5)	2.6 (0.8)	2.9 (0.5)	2.4 (0.9)	2.9 (0.5)
Pagekit	**36.0** (1.3)	0.5 (0.0)	0.5 (0.0)	24.3 (2.6)	31.7 (4.9)	24.7 (2.8)	27.6 (3.8)	**4.2** (0.9)	0.0 (0.0)	0.0 (0.0)	1.0 (0.0)	2.9 (0.4)	1.0 (0.0)	0.7 (1.0)
Splittypie	43.4 (0.3)	37.7 (3.9)	18.9 (1.1)	**46.9** (1.8)	**45.3** (2.5)	**44.1** (3.5)	44.7 (2.0)	**8.0** (1.4)	5.6 (0.5)	3.0 (0.0)	4.0 (0.0)	4.0 (0.0)	4.0 (0.0)	6.0 (0.0)
Phoenix	**81.7** (1.3)	30.2 (9.2)	42.1 (0.0)	60.1 (5.7)	65.1 (9.6)	61.3 (2.3)	63.1 (8.7)	**5.3** (0.5)	0.0 (0.0)	1.2 (0.0)	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.1 (0.5)
Retroboard	61.4 (0.3)	22.2 (0.0)	56.1 (0.0)	68.7 (2.1)	70.9 (4.3)	68.1 (2.7)	**71.9** (3.8)	**1.0** (0.0)	0.0 (0.0)	**1.0** (0.0)	**1.0** (0.0)	**1.0** (0.0)	**1.0** (0.0)	**1.0** (0.0)
Petclinic	**85.0** (0.0)	18.0 (2.5)	35.0 (26.3)	80.5 (7.3)	49.5 (7.9)	83.0 (2.9)	49.5 (7.9)	**11.7** (1.0)	0.7 (0.0)	2.0 (0.0)	1.5 (0.0)	3.0 (0.7)	1.5 (2.1)	3.3 (0.7)
Average	59.8	20.1	29.45	53.2	50.4	53.0	49.3	6.5	1.2	1.4	2.0	2.6	2.0	2.7

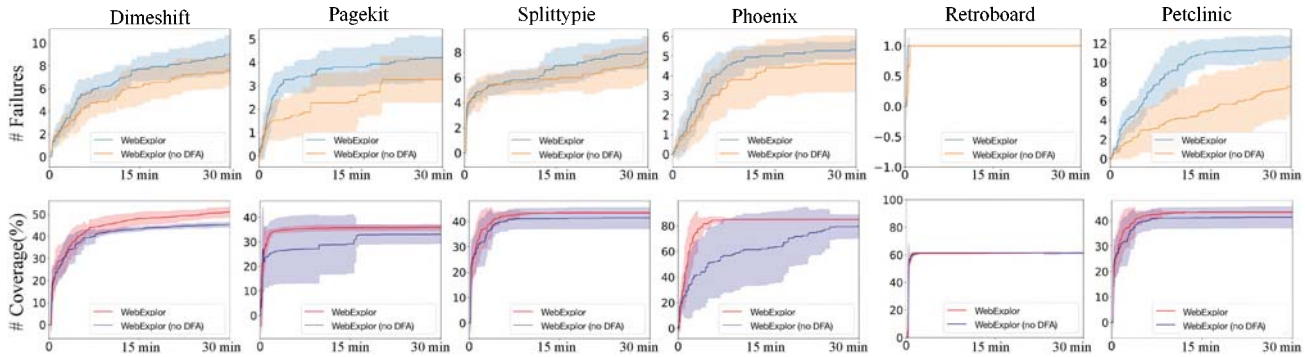

Fig. 7. Evaluation of DFA regarding averaged number of discovered failures (top), code coverage (bottom) and testing efficiency (15 runs). The results are averaged using 15 runs, while the solid line and shaded represent the mean value and standard deviation, respectively.

this page transition trace as an example: homepage → owner-list → owner-info → pet-list → pet-info → visit-info page. Many failures can only be discovered when entering the visit-info page and submit a form with invalid values. *WebExplor* can efficiently achieve this transition via DFA guidance and start subsequent testing. Otherwise, any interruption in the process will result in visiting another page, degrading the testing efficiency (e.g., Fig. 5). Therefore, such high-level guidance effectively leads to promising testing directions, by which *WebExplor* achieves efficient and effective web testing.

Another finding is that DFA can achieve a more stable performance in terms of both failure discovery and code coverage. As shown in Fig. 7, the standard deviation (shaded area) of using DFA is smaller than not using DFA. The variation is particularly obvious in pagekit, phoenix and petclinic subjects.

Answer to RQ3: DFA complements curiosity-driven RL by providing high-level guidance for effective exploration and high efficiency. *WebExplor* achieves a better performance in terms of failure detection rate, code coverage and stability.

E. Evaluation on Real-World Web Applications (RQ4)

RQ4 aims to evaluate the effectiveness of *WebExplor* on real-world web applications. According to the *Alexa* rank list [32], top 50 most popular web applications are chosen for evaluation. In total, *WebExplor* discovered 3,466 failures. After manual inspection, we find these failures consist of 1,889 JavaScript failures, 147 server failures (i.e., status code ≥ 500) and 1,430 other failures (more details in [49]). Moreover, we analyze the URLs of the detected failures and find that 83.71% failures come from the original web applications, while the rest are from third-party libraries. This indicates that most failures indeed exist in the original web applications, resulting in an urgent need of end-to-end testing such as *WebExplor*.

On the other hand, we find that the failures can be caused by either the client or server sides, including static resources loading errors, JavaScript errors due to uncommon operation sequences, and crossing site errors due to accessing different servers (refer to [49] for details), which demonstrates the effectiveness of *WebExplor* in detecting failures in real-world web applications. Moreover, *WebExplor* leverages no manually-tuned parameters, demonstrating high scalability among various modern web applications. More analysis and failure screenshots including web pages and console outputs can be found on our website [49].

Furthermore, a commercial SaaS platform (over a million lines of code) is employed as a detailed case study. *WebExplor* successfully finds 12 unspotted failures, which are confirmed and fixed by developers. We analyze some specific cases of discovered failures as follows.

Asynchronous rendering: The following code segment shows a discovered failure, where gray lines are newly added fixing solutions. The elements (i.e., "kg-container") is asynchronously rendered, which will be a null pointer before

finishing rendering. Therefore, a failure will be triggered once accessing such a null pointer (Line 3). This finding also suggests that developers should pay attention when using asynchronous techniques in web applications.

```
1   // Disable default right click behavior
2 + if (!document.getElementById("kg-container"))
      // throws null point exception without checking
3     document.getElementById("kg-container")
4         .oncontextmenu = function(e) {
5             e.preventDefault();
6         };
```

```
1   // Check invalid email
2 + if (match_email("kg-container"))
      register_email(email);
```

Security defect: The following example shows a module loading failure that results in a security defect. Specifically, when *WebExplor* operates on the "online-editor" in the browser, the server tries to load a non-existent module "brace/mode/c_cpp" by traversing all folders (Line 3).

Consequently, as shown in Fig. 8, the server exposes all folders and folder structure to the client, which could be exploited by malicious attackers, and results in security problems. The "online-editor" is difficult to reach but *WebExplor* discovered it by adopting an effective exploration.

```
227. ^v1/invitation/
228. ^v1/upload/$ [name='upload']
229. ^accounts/
230. ^webhook/
231. ^ci-scan/ [name='ci_scan']
232. ^ci-scan-results/ [name='ci_scan_results']
233. ^prometheus/metrics [name='prometheus_metrics']
234. ^scans/(?P<scan id>[0-9]+)/status/$ [name='scan status']
235. ^secret-admin/
236. ^secret-admin/defender/
237. ^400/$
238. ^403/$
239. ^404/$
240. ^500/$
241. ^admin/
The current path, v1/projects/mode-c_cpp.js/, didn't match any of these.
```

Fig. 8. The exposure of sensitive server information.

Beyond these, *WebExplor* discovered some other failures. For instance, a password-reset API throws an internal exception (400 bad request) once receiving an invalid email input. Besides, when *WebExplor* executes the tab-switching behaviors, an event handler error is thrown, which results in incorrect data rendering. Moreover, searching API throws an error (500 internal server error) if the request payload contains no product or organization ID, which cannot be handled. More analysis and details can be found in our website [49].

Answer to RQ4: *WebExplor* tests modern web applications in an end-to-end fashion with no additional manual efforts (e.g., building a navigation model). Besides, 3,000+ failures detected from the real-world applications further demonstrate the effectiveness of *WebExplor* in failure detection.

F. Threats To Validity

Randomness can be a major threat during testing and the web interaction. We reduce this threat by repeating 15 times for each testing configuration and average the results. Besides, the oracle we proposed may not be complete, and thus *WebExplor* may miss some other unknown failures like UI bugs or other types of bugs. As *WebExplor* performs black-box web testing, the server-side codes are assumed to be unavailable. Therefore, only the coverage of the client-side codes (e.g., JavaScript) are reported, and the evaluation of *WebExplor* in terms of the code coverage may be incomprehensive. The selection of the RL algorithm used for training policies could be biased. We mitigate this problem by selecting the standard RL algorithm [45] for web testing. Furthermore, hyperparameters may be a potential threat. The choice of such hyperparameters is dependent on the domain knowledge, and may be biased. Besides, the selection of the web applications could be biased. To address this, a research benchmark and various real-world web applications, including active commercial websites, are adopted for evaluation.

V. Related Work

Many techniques have been proposed to automate web testing. In the following, we briefly discuss the most relevant solutions and their limitations, which motivates the need for a novel and efficient web testing technique.

Model-Based Testing. Model-based technique is a major paradigm for achieving automatic web testing [15, 5, 38, 51]. This kind of techniques build models to describe the web applications' behaviors in advance, and then derive test cases from the models to find bugs. For instance, approaches like DIG [15], SubWeb [5], and ATUSA [38] extract paths from the navigation model, where genetic algorithm is adopted for performing path selection and input generation. Further, an incremental two-steps algorithm InwertGen [51] is proposed, where the generation of navigation model and test cases are intertwined. Overall, model-based techniques exhibit the advantage of fast test cases generation since no web interaction is required. However, the navigation model may cover only some behaviors of the web applications while others cannot be tested. Besides, expert domain knowledge are required in building high quality models. Such limitations motivate the need for a model-free testing technique. *WebExplor*, to the best of our knowledge, is the first RL-based technique that performs an end-to-end automated web testing for real-world applications and achieves competitive performance comparing to the state-of-the-art techniques.

Test Case Generation. Test case generation consists of building test path and corresponding input values [15]. Given a navigation model, test paths can be generated by search-based approaches [5, 52], and inputs can be generated using random or evolutionary algorithms [15, 53]. In many cases, search-based techniques need to explicitly address path feasibility, resulting in tremendous executions of test case candidates. Search-based algorithms also need to evaluate each test case's fitness value, which is costly since plenty of candidates need to be generated and executed in the browser before converging [54]. Existing techniques for automated test case generation either ignore path feasibility or require

a large number of executions. As for input values, random strategy [38, 53] can generate feasible inputs in most cases. Furthermore, symbolic execution techniques (e.g., Apollo [55], Jalangi [56] and SymJS [57]) use systematic strategies to generate specific inputs to cover hard-to-reach codes. However, generating such specific inputs is not the primary concern of *WebExplor*, and all advanced input generation techniques can be incorporated in the *WebExplor* to further enhance the performance. Recently, there are also some research on testing deep learning models [49, 58, 59, 60], which could be used to test reinforcement learning models.

Reinforcement Learning Based Testing. In recent years, RL has also been applied to help software testing. For instance, Wuji [23] leverages RL and multi-objective optimization for game testing, which is rather different from web applications. Böttinger et al. [61] introduce the first RL based fuzzer to learn high reward seed mutations for testing traditional software. Retecs [62] uses RL to facilitate test prioritization and selection in regression testing. Besides, RL is also adopted in mobile testing techniques [63, 25, 26, 27], ranging from QBE [63] for crash detection to GUI testing [26, 27]. The most recent Q-testing [25] achieves the best mobile testing performance by leveraging the curiosity reward, which is similar to *WebExplor*. Both *WebExplor* and Q-testing are built on the Q-learning, however, due to different characteristics between Android and Web applications, the challenges addressed by the two works are different. In *WebExplor*, we carefully design state representation and suitable rewards for web testing, which are different from Q-testing. Furthermore, we also found that it is not effective when only using Q-learning. Thus DFA-augmented exploration is proposed in *WebExplor*, which is also a key difference compared with Q-testing. The existing studies most related to both the RL and web domains are Artemis [64] and DOM-Q-NET [65] that utilize RL to perform explicitly defined web-based tasks. Different from completing explicit tasks, *WebExplor* targets at web testing via exploring diverse behaviors using curiosity-driven RL, which has not been tackled before.

VI. Conclusion

This paper proposes *WebExplor*, a new RL based approach for automatic web testing. *WebExplor* leverages the curiosity-driven RL to achieve efficient and adaptive exploration. Meanwhile, DFA is proposed to provide high-level guidance, so as to further boost the testing efficacy. Experiments show that *WebExplor* outperforms existing web testing techniques in both code coverage and failure detection. In the future, we plan to extend *WebExplor* with hierarchical RL for a better exploration, and extract business scenarios from the DFA, whereby fault localization can be achievable.

VII. Acknowledgements

The work is supported in part by the National Natural Science Foundation of China (Grant No. U1836214), Special Program of Artificial Intelligence and Special Program of Artificial Intelligence of Tianjin Municipal Science and Technology Commission (No. 569 17ZXRGGX00150), Tianjin Natural Science Fund (No. 19JCYBJC16300), Research on Data Platform Technology Based on Automotive Electronic Identification System, Singapore National Research Foundation, under its National Cybersecurity R&D Program (No. NRF2018NCR-NCR005-0001), Singapore National Research Foundation under NCR (No. NRF2018NCR-NSOE003-0001), NRF Investigatorship (No. NRFI06-2020-0022), JSPS KAKENHI (Grant No. 20H04168, 19K24348, 19H04086), JST-Mirai Program (Grant No. JPMJMI18BB), Japan, and Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), China.

References

[1] J. Huggins and S. Contributors, "Selenium-web browser automation," http://www.seleniumhq.org, 2018.

[2] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 154–164.

[3] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 503–514.

[4] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Pesto: Automated migration of dom-based web tests towards the visual approach," *Software Testing, Verification And Reliability*, vol. 28, no. 4, p. e1665, 2018.

[5] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*. Springer, 2017, pp. 18–32.

[6] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: An algorithm for generating robust xpath locators for web testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 177–204, 2016.

[7] M. Hammoudi, G. Rothermel, and A. Stocco, "Waterfall: an incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 751–762.

[8] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[9] R. V. Binder, "Testing object-oriented software: a survey," *Software Testing, Verification and Reliability*, vol. 6, no. 3-4, pp. 125–252, 1996.

[10] M. Fewster and D. Graham, *Software test automation*. Addison-Wesley Reading, 1999.

[11] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar, "Efficient and change-resilient test automation: An

industrial case study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1002–1011.
[12] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "APOGEN: automatic page object generator for web testing," *Software Quality Journal*, vol. 25, no. 3, pp. 1007–1039, 2017.
[13] Google, "Monkey," https://developer.android.com/, 2018.
[14] "Crawljax," https://github.com/crawljax/crawljax, 2017.
[15] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 142–153.
[16] A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 1–30, 2012.
[17] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 121–130.
[18] S. Athaiya and R. Komondoor, "Testing and analysis of web applications using page models," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 181–191.
[19] M. Schur, A. Roth, and A. Zeller, "Procrawl: mining test models from multi-user web applications," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 413–416.
[20] Y. Zheng, Z. Wang, X. Fan, X. Chen, and Z. Yang, "Localizing multiple software faults based on evolution algorithm," *J. Syst. Softw.*, vol. 139, pp. 107–123, 2018.
[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[22] Y. Zheng, Z. Meng, J. Hao, Z. Zhang, T. Yang, and C. Fan, "A deep bayesian policy reuse approach against non-stationary agents," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 962–972.
[23] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.
[24] R. Shen, Y. Zheng, J. Hao, Z. Meng, Y. Chen, C. Fan, and Y. Liu, "Generating behavior-diverse game ais with evolutionary multi-objective deep reinforcement learning," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, 2020, pp. 3371–3377.
[25] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.
[26] D. Adamo, M. K. Khan, S. Koppula, and R. C. Bryce, "Reinforcement learning for android GUI testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, 2018, pp. 2–8.
[27] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, 2018, pp. 31–37.
[28] Google, "UI automator," https://developer.android.com/training/testing/uiautomator, 2019.
[29] DeepMind, "Dota2," https://openai.com/five/, 2019.
[30] Y. Zheng, Z. Meng, J. Hao, and Z. Zhang, "Weighted double deep multiagent reinforcement learning in stochastic cooperative environments," in *PRICAI 2018: Trends in Artificial Intelligence - 15th Pacific Rim International Conference on Artificial Intelligence, China, 2018*, ser. Lecture Notes in Computer Science, vol. 11013, 2018, pp. 421–429.
[31] Y. Zheng, J. Hao, Z. Zhang, Z. Meng, and X. Hao, "Efficient multiagent policy optimization based on weighted estimators in stochastic cooperative environments," *J. Comput. Sci. Technol.*, vol. 35, no. 2, pp. 268–280, 2020.
[32] Aleax Inc., "Top web sites rank list," https://www.alexa.com/topsites, 2020.
[33] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[34] W. W. W. Consortium, "Hypertext transfer protocol – http/1.1," https://tools.ietf.org/html/rfc2616s, 2012.
[35] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.
[36] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.
[37] Google, "Chrome devtools protocol viewer," https://chromedevtools.github.io/devtools-protocol/tot, 2020.
[38] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38,

no. 1, pp. 35–53, 2011.
[39] Mozzila, "HTML," https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input, 2020.
[40] T. Yang, J. Hao, Z. Meng, Y. Zheng, C. Zhang, and Z. Zheng, "Bayes-tomop: A fast detection and best response algorithm towards sophisticated opponents," in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS*, 2019, pp. 2282–2284.
[41] Y. Zheng, J. Hao, Z. Zhang, Z. Meng, T. Yang, Y. Li, and C. Fan, "Efficient policy detecting and reusing for non-stationarity in markov games," *Auton. Agents Multi Agent Syst.*, vol. 35, no. 1, p. 2, 2021.
[42] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 16–17.
[43] N. Savinov, A. Raichuk, D. Vincent, R. Marinier, M. Pollefeys, T. P. Lillicrap, and S. Gelly, "Episodic Curiosity through Reachability." *ICLR*, 2019.
[44] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in neural information processing systems*, 2016, pp. 1471–1479.
[45] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
[46] E. Jang, S. Gu, and B. Poole, "Categorical reparametrization with gumble-softmax," in *International Conference on Learning Representations*, 2017.
[47] J. A. Bondy, U. S. R. Murty *et al.*, *Graph theory with applications*. Macmillan London, 1976, vol. 290.
[48] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.
[49] WebExplor, "WebExplor," https://sites.google.com/view/webexplor, 2020.
[50] O. Korosteleva, *Nonparametric methods in statistics with SAS applications*. CRC Press, 2013.
[51] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE, 2015, pp. 1–6.
[52] M. E. Dincturk, G.-V. Jourdan, G. V. Bochmann, and I. V. Onut, "A model-based approach for crawling rich internet applications," *ACM Transactions on the Web (TWEB)*, vol. 8, no. 3, pp. 1–39, 2014.
[53] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 571–580.
[54] S. Luke, *Essentials of metaheuristics*. Lulu Raleigh, 2013, vol. 2.
[55] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 261–272.
[56] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.
[57] G. Li, E. Andreasen, and I. Ghosh, "Symjs: automatic symbolic testing of javascript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 449–459.
[58] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 146–157. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330579
[59] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 477–487.
[60] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 5772–5778.
[61] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, 2018, pp. 116–122.
[62] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, 2017, pp. 12–22.
[63] Y. Köroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: qlearning-based exploration of android applications," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, 2018, pp. 105–115.
[64] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang, "Reinforcement learning on web interfaces using workflow-guided exploration," *arXiv preprint arXiv:1802.08802*, 2018.
[65] S. Jia, J. Kiros, and J. Ba, "Dom-q-net: Grounded rl on structured language," *arXiv preprint arXiv:1902.07257*, 2019.