A Reinforcement Learning Approach to Generating Test Cases for Web Applications

Xiaoning Chang[†‡§*], Zheheng Liang[§ξ*], Yifei Zhang[†‡], Lei Cui[§ξ], Zhenyue Long[§ξ], Guoquan Wu[†‡¶], Yu Gao[†‡], Wei Chen[†‡], Jun Wei[†‡], Tao Huang[†‡]

[†]*State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, Beijing, China*

[‡]*University of Chinese Academy of Sciences, Beijing, China*

[§]*Joint Laboratory on Cyberspace Security, China Southern Power Grid, Guangzhou, China*

[ξ]*Guangdong Power Grid, Guangzhou, China*

[†]{changxiaoning17, gqwu, gaoyu15, wchen, wj, tao}@otcaix.iscas.ac.cn

[†]zhangyifei.steven@gmail.com, [§]liangzheheng@qq.com, [§]cuilei@gdxx.csg.cn, [§]zhenyue@undecidable.org

***Abstract*—Web applications play an important role in modern society. Quality assurance of web applications requires lots of manual efforts. In this paper, we propose *WebQT*, an automatic test case generator for web applications based on reinforcement learning. Specifically, to increase testing efficiency, we design a new reward model, which encourages the agent to mimic human testers to interact with the web applications. To alleviate the problem of state redundancy, we further propose a novel state abstraction technique, which can identify different web pages with the same functionality as the same state, and yields a simplified state space. We evaluate WebQT on seven open-source web applications. The experimental results show that WebQT achieves 45.4% more code coverage along with higher efficiency than the state-of-the-art technique. In addition, WebQT also reveals 69 exceptions in 11 real-world web applications.**

***Index Terms*—State exploration, Reinforcement learning, Software testing**

I. Introduction

Web applications have become drastically increased over recent years. According to a recent survey [1], there are more than 1 billion web applications in July 2022. On average, users spend 7 hours per day online. It is important to develop web applications with high quality. Manual/automated testing technique can be used for quality assurance of web applications. However, manual testing is time-consuming. Moreover, there is a large number of feasible action sequences for real-world web applications. Manual test cases can only cover a small portion of them. Therefore, an automatic test case generation tool is becoming an urgent need to ensure the quality of web applications.

One challenge in automatic test case generation for web applications is that, some deep states can only be reached by specific action sequences. For example, in web application phoenix [2], only action sequence type username ⇀ type password ⇀ click login ⇀ click Add-new-board ⇀ type board-name ⇀ click Create-board ⇀ ... ⇀ click edition icon ⇀ click Tags can reach color edition state. However, a web page may contain a large number of interactive UI elements [3]. It is difficult to generate valid action sequences to reach diverse states. Any interruption when performing the action sequence would fail to reach the target state. Existing random-based approaches [4] randomly select actions and are prone to generate ineffective action sequences. Model-based approaches [5]–[7] first build a model for the target web application and then traverse the model to generate action sequences. Since it is hard to build a complete model, these approaches only generate limited action sequences. Recently, reinforcement learning approaches are widely adapted in automatic software testing [8], [9]. They design reward functions to train a policy to explore state space. However, their reward functions are too simple to efficiently generate valid action sequences. As a result, these approaches cannot discover new deep states in the limited time.

Another challenge is that, web applications widely consist of near-duplicate web pages [10]. Namely, web pages replicate functionality but their content and structure are different. These web pages should be identified as the same state. Otherwise, redundant states would degrade the state space exploration efficiency afterwards. However, a recent survey [10] reveals that, existing approaches [4], [11], [12] that directly compare the content or structure of web pages fail to identify near-duplicate web pages as the same state. In addition, computer vision-based approaches [9], [13] apply deep learning to identify states. These approaches need a large number of data to train the model, which require lots of effort.

In this paper, we propose WebQT, an automatic test case generation tool for web applications based on reinforcement learning. It relies on the reinforcement learning agent to select an optimal action, and then perform it on the target web application. Based on the result of execution, WebQT updates the reward to the executed action. To be able to generate valid action sequences effectively during the exploration, we design a novel reward model to guide the agent to mimic human testers to interact with web applications. After execution, WebQT extracts state and valid actions from the web page. To avoid state redundancy, our key observation to identify states is that, similar elements on the page will serve the same

*Xiaoning Chang and Zheheng Liang contribute equally.

¶Guoquan Wu is the corresponding author.

functionality. Based on this observation, we merge similar elements to represent one unique functionality. Web pages with similar functionalities are identified as the same state.

To demonstrate effectiveness of WebQT, we evaluate WebQT on real-world web applications from three aspects. First, we compare WebQT and WebExplor [8] on a research benchmark of 7 web applications. The experiment shows that WebQT achieves 41.23% more branch coverage and 45.4% more line coverage than WebExplor. Second, we implement $WebQT_{se}$ (i.e., WebQT with state extraction proposed by WebExplor) and $WebQT_r$ (i.e., WebQT with reward model proposed by WebExplor). The comparison between WebQT, $WebQT_{se}$ and $WebQT_r$ demonstrates WebQT outperforms $WebQT_{se}$ and $WebQT_r$ in coverage and efficiency. Third, we evaluate WebQT on 11 real-world web applications randomly chosen from top 50 web applications [14] in the world. WebQT discovers 69 exceptions in 11 web applications.

We summarize our main contributions as follows:

- We propose an automatic test case generation technique for web applications based on reinforcement learning. Specifically, a novel reward model is designed, which can guide the reinforcement learning agent to mimic human testers to efficiently explore the state space.
- To avoid state redundancy during the exploration, we design a new state abstraction technique, which can identify different web pages with the same functionality as the same state.
- We implement our approach as WebQT and evaluate it in real-world web applications. The experimental results shows that WebQT can effectively and efficiently generate test cases for web applications.

II. Motivation

We regard automatic test case generation for web applications as a problem of state space exploration. That is said, we aim to generate action sequences to reach diverse states of the web application under test. In order to achieve this goal, we need to address two technical challenges: (a) how to represent and abstract the state for web applications to avoid state redundancy problem and (b) how to design an effective exploration strategy to reach more different states giving limited time budget.

State Abstraction. Redundancy widely exists across different pages in web applications [10]. Namely, web pages replicate functionality but their content and structure are different. For example, Figure 1 (a-c) shows three of web pages, where web page (a-b) consists of two pieces of news respectively and web page (c) consists of ten pieces of news (we only show three of them). Although web pages are different from content (i.e., web page (a) and (b)) and structure (i.e., web page (a) and (c)), they are conceptually same and should be identified as the same state. Otherwise, there would be redundant states, degrading the efficiency of state space exploration.

However, existing works cannot address such a problem. For example, WebExplor [8] directly utilizes URLs and HTML documents of web pages to represent states. For web pages (a-c) in Figure 1, whose URLs are different and HTML documents vary a lot, WebExplor would identify them as different states.

To overcome this challenge, our intuition is that, similar elements on the page will serve for the same functionality, and one of these similar elements can be utilized to represent the functionality. After simplification, web pages that provide similar functionality can be identified as the same state. In this way, our approach is able to identify web pages with various amount of similar functionalities as the same state. For example, in Figure 1 (a), our approach identifies elements that representing title of news as similar elements and utilizes one of them (i.e., e_3 in Figure 1 (d)) to represent their functionality in the state. Similarly, we utilize element e_4 to represent the functionality of publisher of news, and element e_5 to represent the functionality of link of full coverage in the state. As a result, no matter how many news there are, we extract the same state, which is shown in Figure 1 (d). Note that, since we do not take text into consideration, our approach is able to identify web page with different content as the same state.

State space exploration. The aim of state space exploration is two folds. On the one hand, the exploration strategy should be able to randomly explore the web application to cover diverse states. On the other hand, since some deep states can only be reached by specific action sequences, the exploration strategy should be able to generate valid action sequences effectively. For example, for JPetStore web application [15], in order to reach checkout state, the following action sequence should be performed: type username $\rightharpoonup$ type password $\rightharpoonup$ click login $\rightharpoonup$ type keyword $\rightharpoonup$ click search $\rightharpoonup$ click a dog $\rightharpoonup$ click add to cart $\rightharpoonup$ click checkout. However, according to existing work [3], a web page has 76 actions on average, making it hard to efficiently generate valid action sequences. Any interruption when performing the action sequence would fail to reach the target state.

Existing state space exploration techniques cannot achieve above two goals at the same time. For example, random-based approaches [4], [16] are able to randomly explore the state space but hard to generate valid action sequences. In model-based testing, model can provide knowledge to generate valid action sequences. However, existing approaches [5]–[7] are hard to construct a complete model about the application under test. Recently, researchers tend to adopt reinforcement learning to test case generation for web applications [8], [9], as such technique can keep a balance between exploration and exploitation. However, the designed reward function, which is the key to reinforcement learning, is too simple, and cannot guide the action selection effectively when multiple actions need to be performed sequentially to discover new states. For example, in WebExplor [8], the reward function only considers the number of each transition in a state, and is hard to guide the generation of valid action sequences to expose some deep states of the application under test. To increase the effectiveness and efficiency of state space exploration, we observe that, if the reinforcement learning agent can mimic human interaction behavior with the application, it is able to

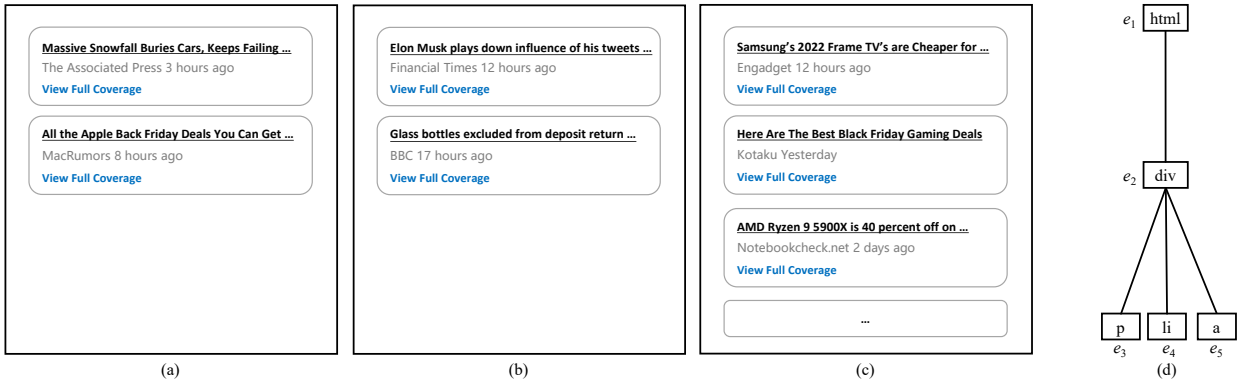


Fig. 1. An example of web pages that have different content and structure but have same functionality. Web page (a) and (b) have different content, while (a) and (c) have different structure. (d) is the state extracted from (a-c).

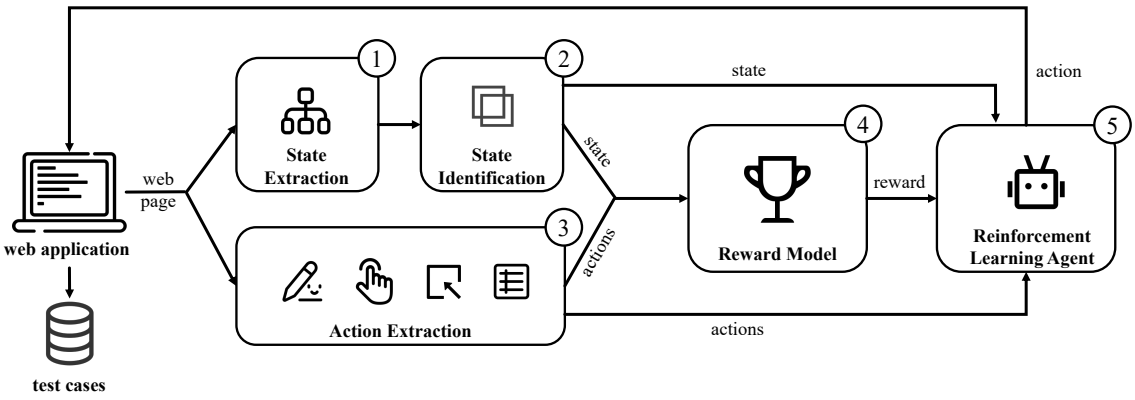


Fig. 2. The overview of WebQT

efficiently generate valid action sequence. For example, after filling the text box, if the agent chooses to click the search button *next* to the text box, it will reach the product detail state quickly, rather than clicking the homepage tablet far from the text box. Motivated by this observation, we design a novel reward model, which not only encourages the action for which the exposed state is worth exploring, but also the one for which the generated action sequence is consistent with human interaction behaviors. Under the guidance of the new reward model, the reinforcement learning agent is able to effectively generate action sequences to explore some deep states.

III. Approach

Figure 2 presents the overview of WebQT. Given a web application, WebQT automatically generates sequences of actions (i.e., test cases) to test the web application. When the target web application arrives at a web page, WebQT extracts a state ❶ (Section III-A), and determines whether it is a previously visited state ❷ (Section III-B) to avoid redundant states. Next, WebQT extracts valid actions from the web page ❸ (Section III-C). Then, the action previously performed on the web application is evaluated by proposed reward model ❹ (Section III-D), estimating how much the action contributes to state space exploration. Based on extracted states and actions along with reward, the reinforcement learning agent is trained to learn a policy π to select actions for state space exploration ❺ (Section III-E).

A. State Extraction

To explore the web application via reinforcement learning, we need to define the state representation. As each web page is represented by a HTML document, a straightforward way is to leverage HTML document representation of the page. However, adopting original HTML document as state directly will suffer from the state redundancy problem as a large number of pages will be generated during the exploration and many of them only differ slightly in the document. To address this limitation, we design a new state representation, defined as follows.

Definition 1. *A state s is a simplified HTML document tree (V, E), where node $e \in V$ represents a DOM element, and $(e_1, e_2) \in E$ represents the parent-child relationship, in which e_1 is the parent of e_2.*

In order to construct an abstract state from the original HTML document, we first reduce the number of elements in the document tree. We traverse the HTML document tree to remove element e if it only has one child element. Namely, consider the parent element and child element of e is e_p and e_c, respectively. We treat e_c as the child element of e_p. The

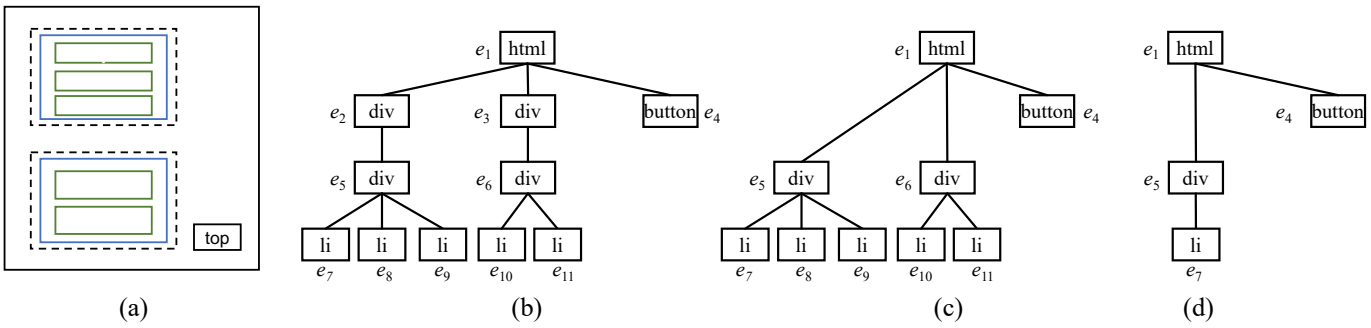


Fig. 3. An state extraction example, where (a) shows a simplified web page along with its HTML document tree, HTML document tree and extracted state shown in (b-d), respectively.

intuition is that, if element e only has one child element e_c, XPath of element e_c contains the information about element e. In this way, the number of elements in the HTML document is reduced.

Then we further merge the similar elements, assuming that similar elements will serve for the same functionality. In a web page, element e_1 and e_2 are deemed to be similar, if they satisfy following conditions.

(1) *They have similar structure.* If XPath of e_1 and e_2 is $tag_1[p_1]/.../tag_i[p_i]/.../tag_n[p_n]$ and $tag_1[p_1]/.../tag_i[p'_i]/.../tag_n[p_n]$ respectively, then we consider element e_1 and e_2 has similar structure with each other, where $p_i \neq p'_i$. Technically, given an element e, we take advantage of its XPath (by ignoring the index of its ancestors in the path) to find all elements in the same layer that have similar structures with e.

(2) *They have similar style.* The style similarity between element e_1 and e_2 is defined as the distance between their properties:

$$sim(e_1, e_2) = \sum_{p \in props} \frac{dist(e_1[p], e_2[p]) \cdot weight[p]}{|props|} \quad (1)$$

where $dist(e_1[p], e_2[p])$ is the edit distance between property p of element e_1 and e_2. We take property src, $href$, $type$, $className$, $height$, $width$, $positionX$ and $positionY$ into consideration, as shown in Table I. The first row presents the way we calculate property distance: (i) For property src, $href$ and $className$, whose values are strings, we calculate property distance by edit distance between them [17]. (ii) For property $height$, $width$, $positionX$ and $positionY$, whose values are number, we calculate ratio of them as the property distance. (iii) For property $type$, if property $type$ of two elements are same, the property distance is 1. Otherwise, the property distance is zero. The second row of Table I shows weights of property distances.

If the style similarity is greater than our predefined threshold, we consider they have similar display styles.

Next, we perform the breath-first traversal on the simplified tree to extract the state. For element e, if it has similar element e', we assume they serve for the same functionality, and add one of them into state s to represent the functionality. Next, we continue to search similar elements among child elements of e and e'. If child element e_c of element e is similar with child element e'_c of element e', we add element e_c into state s, and treat element e_c as the child element of e. Otherwise, if there is no similar element among child elements of e and e', we add child element e_c of element e and child element e'_c of element e' as the child of element e into state s.

Example. In Figure 3, (a) and (b) shows a simplified web page and its HTML document tree, respectively. To extract the state for web page (a), we first simplify its HTML document tree. Element e_2 and e_3 corresponds to dotted box in (a), respectively. Since element e_2 and e_3 only has one child element respectively, we remove them from the HTML document tree and treat element e_5 and e_6 as the child element of e_1. After removal of element e_2 and e_3, the simplified HTML document tree is shown in (c). Then, we traverse the simplified HTML document tree to extract state s. Since element e_5 are e_6 similar in both structure and style, which corresponds to blue box in (a) respectively, we regard they serve for the same functionality and add element e_5 into state s to represent their functionality in the state shown in (d). Next, we continue to search similar elements among child elements of e_5 and e_6. We find that, element e_7 is similar with element $e_8, e_9, \ldots, e_{11}$ in both structure and style, which corresponds to green boxes in (a), respectively. Similarly, we add element e_7 into the state s to represent the functionality that elements $e_7, e_8, \ldots, e_{11}$ serve for. In state s, element e_7 is processed as the child element of e_5. For element e_4, since it has no similar element, we reserve it in the state. The extracted state from web page (a) is shown in (d).

B. State Identification

To avoid duplicate states in the state space S, we determine whether the extracted state s is a previously visited state. If there is no state in the state space is same with state s, we add state s into the state space. Otherwise, if state $s' \in S$ is same with state s, we consider state s a previously visited state, and do not add state s_i into the state space.

The basic strategy is to compare s with each state $s' \in S$ one by one. However, to compare state s and s', all elements of state s needs to be compared with elements of state s'. It is

TABLE I
STYLE SIMILARITY COMPUTATION

property	src	href	type	className	height	weight	positionX	positionY
type	str	str	specified	str	number	number	number	number
weight	0.62	0.62	0.5	0.45	0.1	0.1	0.1	0.1

time-consuming to compare state s with *all* states in the state space.

To address this problem, we observe that, given a web application $\mathcal{A}$, states of $\mathcal{A}$ share a number of elements. Based on this observation, given states $s_0, s_1, \ldots, s_{n-1} \in S$, we build a *state index tree*, which consists of all elements and edges of existing states $s_0, s_1, \ldots, s_{i-1}$. We compare s with $stateIndexTree$ to determine whether there is a state $s' \in S$ that is same with state s , i.e, whether s is newly visited.

We first label elements of state s. The label on an element e will be used to identify the state that element e belongs to. Then, we get state index tree of existing states. If no state index tree has been built, we regard state s as the state index tree, and there is no state that is the same with state s. If state index tree $stateIndexTree$ exists, we compare state s with $stateIndexTree$ to determine whether there is a same state with state s as follows.

We initialize that the number of similar elements between state s and each state $s' \in S$ to zero. Then, we perform pre-order traversal on state s and state index tree $stateIndexTree$. Suppose element u in state s is similar to element v in $stateIndexTree$. According to the label of element v, element v belongs to the set of states B, then the number of similar elements between state s and $s' \in B$ is increased by one. We also add label of state s on element v. Next, we continue to find similar elements for each child elements of element u, among the children of element v. Otherwise, if there is no similar element with u in $stateIndexTree$, we add element u and its descendants into $stateIndexTree$.

Based on the number of similar elements between state s and state s', we calculate state similarity $stateSim$ between state s and s', which is defined as

$$stateSim(s, s') = \frac{\#similarNum}{min(\#s, \#s')} \tag{2}$$

where $\#similarNum$ is the number of similar elements between state s and s'. $\#s$ and $\#s'$ denotes the number of elements of state s and s', respectively. Note that, since we compare states and state index tree in pre-order traversal, we also take the structure of similar elements into consideration.

If similarity between state s and s' is larger than our predefined threshold between states $threshold$, we regard s and s' are the same state.

Example. Figure 4 (a) presents a state index tree built by state s_a and s_b, where elements labeled by A (or B) belong to state s_a (or state s_b). Figure 4 (b) shows the state s_c. We compare state s_c with the state index tree to determine whether state s_c is a new state. We first label elements of state s_c by

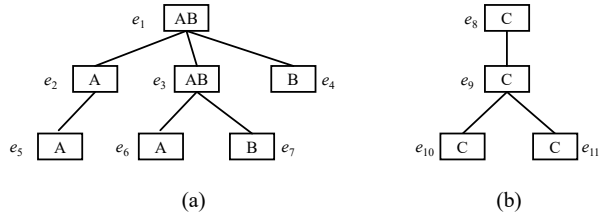


Fig. 4. An example of state identification. (a) presents a state index tree built by state s_a and s_b, where elements labeled by A (or B) belong to state s_a (or state s_b). (b) presents a state s_c.

label C and initialize the number of similar elements between state s_a (and state s_b) and s_c to zero. Then, we compare root element e_8 of state s_c with root element e_1 of the state index tree. Element e_8 is similar with e_1. Since element e_1 belongs to state s_a, we increase the number of similar elements between state s_a and s_c by 1. Since element e_1 also belongs to state s_b, the number of similar elements between s_b and s_c is also increased by 1. We continue to compare child elements of e_8 and e_1. We find that, element e_9 of state s_c is similar with element e_3. Again, we increase the number of similar elements between state s_a (and state s_b) and s_c by 1. Next, we compare child elements of e_3 and e_9. Element e_{10} is similar with element e_6, which belongs to state s_a, while element e_{11} has no similar element among child elements of element e_3. Therefore, state s_c has three similar elements with state s_a (i.e., e_8, e_9 and e_{10}), and $stateSim(s_a, s_c) = 3/min(5, 4) = 0.75$. State s_c has two similar elements with state s_b (i.e., e_8 and e_9), and $stateSim(s_b, s_c) = 2/min(4, 4) = 0.5$.

C. Action Extraction

Definition 2. *An action a in our approach is defined as $a = (ele, type[, param])$, where ele is the interactable element that a operates on, $type$ is the type of a. In particular, if action a is of type input, type form-fill or select, action a has input value $param$.*

Our current approach supports the action types: click, input, select and form-fill, which are common actions in modern web applications. In particular, type form-fill is specific to form elements. More types of actions can be integrated into our approach in the future.

We traverse the HTML document tree to check whether element e is interactable. If yes, we generate an action a that accesses to element e as follows:

- If tag of e matches our default configuration, we consider element e is interactable and generate an action for it. Our configuration is depicted in Table II. For example, if tag of element e is a, we generate an action of type click

TABLE II
ACTION EXTRACTION

	a	button	input	textarea	form	fieldset	select
click	√	√					
input			√	√			
form-fill					√	√	
select							√

on element e. Specially, if tag of element e is form or formset, we generate form-fill actions on it.

- If tag of e is $input$ or $textarea$ and $e.type \in \{radio, label, checkbox\}$, we consider element e is clickable and generate an action of type click on element e.
- If tag and $className$ of e matches user configuration, we consider element e is interactable and generate an action for element e. For example, if tag and $className$ of element e is div and $submit$ respectively, we generate a clickable action for it.

We provide values for actions of type input as follows [4]. First, we obtain values from user configurations so that our approach can reach certain states. For example, we need users to provide username and password to login. Second, if users do not provide custom values, we randomly generate values. Specially, for element e accessed by action, if $type$ of e is $email$, we randomly generate an email value. In a word, since input values can affect test procedure, we explore normal and exceptional states by valid input values (i.e., user-configured values) and invalid input values (i.e., randomly generated values). As for form-fill action that accesses to the form element f, we generate values for each action of type input that accesses to element inside form f. As for the action of type select on element e, whose tag is select, we process the option tag inside the element e and randomly choose one of available options as input value.

D. Reward Model

We adapt reinforcement learning algorithm Q-Learning [18] to generate test cases to explore the state space. A key to reinforcement learning is a reward model, which can optimize policy π so that WebQT can reach diverse states efficiently. When designing the reward model, we have the following two aims: (a) the reward model should guide WebQT to reach diverse states during the exploration. (b) the reward model should encourage WebQT to interact with the application like human, which can generate valid action sequences to cover more deep states. To achieve the above two goals, we manually explore web applications and obtain the following observations, which constitute the basis of our reward model:

- *Action locality.* Human usually selects actions located in the same area because these actions usually serve for the same functionality of web applications. For example, in order to search for a product, human fills the text box and then click the search button *next* to it, rather than clicking "About us" link at the bottom of the web page after entering keyword in the text box.
- *Attention of action.* Human is prone to perform actions on elements that newly appear at the current state. For example, a pop-up menu appears on the new state after performing an action. Compared to other elements which already exist in the previous state, human testers tend to be more attracted by newly appeared elements, and choose to click items on the pop-up menu.
- *Frequency of action.* In a state, if an action is less to be chosen in the past compared to other actions, more chance should be given to this action to reach diverse states during the exploration.
- *Proportion of unexecuted actions.* If a state has more actions that have not been executed after performing an action, it is more worthy to encourage this action as new states may be exposed when reaching such a state.

Reward indicator. Based on above observations, we define *reward indicators* calculated after performing each action. After execution of action a_i, the action sequence is $as = \langle a_0, ..., a_i \rangle$, which makes state transition sequence $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \dots s_{i-1} \xrightarrow{a_i} s_i$, we define:

(1) r_{loc}: this reward based on the locality of action a_i. Consider element e_{i-1} and e_i, which are accessed by action a_{i-1} and a_i, respectively. We encourage the agent to select action a_i so that element e_i is near to element e_{i-1} in the page. If size of element e_{i-1} or e_i is large, the spatial distance between element e_{i-1} and e_i is large, even if these two element are next to each other. In order to ease the impact caused by size of elements, we define r_{loc} as:

$$r_{loc} = \frac{\sqrt{(h(e_{i-1}) + w(e_{i-1})) * (h(e_i) + w(e_i))}}{dist(e_{i-1}, e_i)} \quad (3)$$

where $h(e)$ and $w(e)$ represents height and width of element e, respectively. $dist(e_{i-1}, e_i)$ is the Levenshtein distance [17] between element e_{i-1} and e_i.

(2) $r_{attention}$: this reward based on the attention of action a_i. Human is more likely to perform an action on the *mutant element* (e.g., newly added elements) that appears at state s_i but does not exist at state s_{i-1}. Therefore, we define $r_{attention}$ as:

$$r_{attention} = \begin{cases} 1/mutants(s_i) & isMutant(e_i) \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

Namely, if element e_i accessed by action a_i is a mutant element at state s_i, action a_i is rewarded by $1/mutants(s_i)$, where $mutants(s_i)$ is the number of newly appeared actionable elements in s_i. This reward value is inversely proportional to the number of mutants in s_i. If the number of mutants in s_i is large, a small value will be given to a_i.

To guide WebQT to generate test cases with diversity, we define the following reward indicators:

(3) r_{freq}: this reward based on the execution frequency of transition (s_{i-1}, a_i, s_i). If transition (s_{i-1}, a_i, s_i) has been executed many times, we assign a small reward r_{freq} to it, which is defined as:

$$r_{freq} = \frac{1}{\sqrt{N_i}} \tag{5}$$

where N_i is the execution number of transition (s_{i-1}, a_i, s_i).

(4) $r_{explore}$: the degree of exploration of state s_i. After execution of action a_i, the web application transits to state s_i. If state s_i has a large portion of valid actions that have not been executed, action a_i contributes to state exploration and we reward it by $r_{explore}$, which is defined as:

$$r_{explore} = \frac{n_i}{m_i} \tag{6}$$

where m_i and n_i denotes the number of executable actions at state s_{i+1} and the number of actions that have not been executed at state s_i, respectively.

Reward function. Based on reward indicators, we define a reward function $r_i(a_i)$ as:

$$r_i(a_i) = \begin{cases} penalty & \text{if } ext(s_i) \text{ or } s_i = s_{i-1}, \\ r'_i(a_i) & \text{otherwise.} \end{cases} \tag{7}$$

If the web application transits to external link (denoted as $ext(s_i)$) or state s_i is same with state s_{i-1} (denoted as $s_i = s_{i-1}$), then we assign a negative reward for such a transition, i.e., $penalty < 0$. Otherwise, we assign a positive reward $r'(as)$ calculated by:

$$\begin{aligned} r'_i(a_i) = w_{loc} * r_{loc} + w_{attention} * r_{attention} \\ w_{freq} * r_{freq} + w_{explore} * r_{explore} \end{aligned} \tag{8}$$

where $w_{loc} > 0$, $w_{attention} > 0$, $w_{freq} > 0$ and $w_{explore} > 0$ are weights, which are determined by several tests in the experiment.

E. Reinforcement Learning Agent

In this section, we illustrate how our approach explores state space, as shown in Algorithm 1. Given a target web application env, we first retrieve the URL of its homepage, i.e., url (Line 1) and initialize the set of test cases T, policy π and state space S (Line 2). WebQT runs N episodes to train policy π. At the beginning of each episode, the target web application is reset by visiting its homepage page p_0 via url (Line 4). WebQT extracts state and executable actions from web page p_0 (Line 5).

WebQT is allowed to try M steps to generate action sequence as during each episode (Line 7). In each step, WebQT selects action a_i at state s_{i-1} (Line 8). After performing action a_i, the web application is directed to the web page p_i (Line 9), from which WebQT extracts state s_i along with the set of valid actions $actions$ (Line 10). Then, WebQT updates state space S by s_i (Line 11). The reward r_i is calculated for action a_i (Line 12). Based on s_{i-1}, a_i, s_i and r_i, policy π is update (Line 13). Finally, act a_i is added into action sequence $acts$ (Line 14).

In order to train the policy π, our approach adapts reinforcement learning algorithm Q-Learning [18]. Q-Learning trains the policy via function $Q : S \times A \to \mathbb{R}$, where $Q(s_{i-1}, a_i)$ estimates how good it is to perform action a_i at state s_{i-1}. After state transition from s_{i-1} to s_i by performing action a_i, reward r_i is calculated by Equation 7, and function Q is updated by:

$$Q(s_{i-1}, a_i) \leftarrow Q(s_{i-1}, a_i) + \alpha(r_i + \gamma Q^*(s_i, a_{i+1}) - Q(s_{i-1}, a_i)) \tag{9}$$

where $Q^*(s_i, a_{i+1})$ is the maximum cumulative reward can be achieved from state s_i. As we can see in this equation, future cumulative reward is discounted by factor $\gamma \in [0, 1]$. $\alpha \in [0, 1]$ is the learning rate.

Different from traditional reinforcement learning applications, the action space of web applications is large, making it hard to select an action that contributes to state space exploration. In order to overcome this challenge, we design following strategies, in addition to reward model.

ϵ-greedy algorithm. We adapt ϵ-greedy algorithm to keep balance between exploration and exploitation. Before selecting an action at state s_{i-1}, WebQT first computes a random value $k \in [0, 1]$. If k is no less than ϵ, WebQT makes exploitation: action with maximum Q value is selected. Otherwise, WebQT makes exploration: WebQT randomly selects one of valid actions at state s_{i-1}. With the increasing number of episodes, ϵ is decaying, WebQT switches from exploration to exploitation, and action with maximum Q value is more likely to be selected.

Form-fill actions. Form elements are widely used in web applications. However, it is difficult for the agent to fill a form.

Algorithm 1: State space exploration

Input: env (the target web application)
Hyperparameter: N (the number of episodes), M (the number of step in each episode)
Output: T (test cases)

1 get url of target application env;
2 $T \leftarrow \emptyset, \pi \leftarrow \emptyset, S \leftarrow \emptyset$;
3 **for** $e \leftarrow 1; e \leq N; e++$ **do**
4 $\quad reset(env)$;
5 $\quad s_0, actions \leftarrow extract(p_0)$;
6 $\quad as \leftarrow \emptyset$;
7 $\quad$ **for** $i \leftarrow 1; i \leq M; i++$ **do**
8 $\quad\quad$ select action a_i at state s_{i-1};
9 $\quad\quad p_i \leftarrow env(a_i)$;
10 $\quad\quad s_i, actions \leftarrow extract(p_i)$;
11 $\quad\quad$ update S by s_i;
12 $\quad\quad r_i \leftarrow reward(s_{i-1}, a_i, s_i)$;
13 $\quad\quad$ update π using s_{i-1}, a_i, s_i and r_i;
14 $\quad\quad as.push(a_i)$;
15 $\quad$ **end**
16 $\quad T.push(as)$;
17 **end**
18 **return** T;

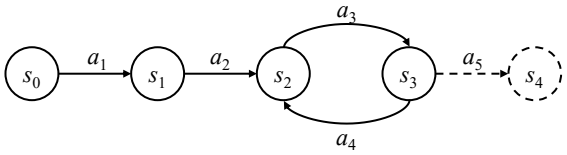


Fig. 5. An example of state transition, where known and unknown states are denoted by solid and dotted circles respectively.

According to the work [3], in modern web applications, a web page contains 76 actions on average. For a form consisted of 5 text boxes, the possibility of randomly finding a correct action sequence to fill the form is $1/(76)^5 = 10^{-10}$.

To address above challenge, we identify form-fill actions on form elements, as discussed in Section III-C. If the agent selects a form-fill action on form element f, our approach fills all input elements in the form element f. For example, in the login page, we fill the username text box and password text box. In this way, we increase the possibility of finding a correct action sequence on f.

Local optima. Although we apply ϵ-greedy algorithm to keep balance between exploration and exploitation, it is still challenging for reinforcement learning to achieve effective exploration. For example, in Figure 5, acton a_3 with maximum Q value is selected at state s_2. After execution of acton a_3, state s_2 transits to state s_3. Similarly, acton a_4 with maximum Q value is selected at state s_3, and state s_3 transits to state s_2. Next, acton a_3 is selected and state transits to s_3. We regard the phenomena, where no new state is explored, as a local optima.

In order to overcome the problem of local optima, we check whether the agent is trapped in the local optima before action selection. Consider the acton sequence $as = \langle a_1, ..., a_n \rangle$. If there is no new state in recent window of size Z (i.e., there is no new state among state $s_{n-Z+1}, s_{n-Z+2}, ..., s_n$), we consider that the agent is strapped in the local optima.

We propose two strategies to make our agent jump out of local optima. If the state where agent is trapped has more than one valid action, our approach selects actions from valid actions except the action with maximum Q value. For example, in Figure 5, when strapped at state s_3, the agent selects action a_5 and transits to new state s_4 to escape local optima. Different from existing work that ends current episode immediately when trapped in the local optima [8], we find that states (e.g., state s_4) behind the local optima need to be explored. Stopping current episode makes it hard to explore states behind local optima.

Second, if the state where agent is trapped only has one valid action (i.e., the action with maximum Q value), our approach ends current episode and search for the transition with lowest execution times, denoted as $s_{min-1} \xrightarrow{a_{min}} s_{min}$. State s_{min} is the beginning of the new episode. Our intuition is that, putting WebQT at a different state, which has been visited much less, can increase the possibility to exploring diverse states. To achieve this goal, we build a state graph, whose nodes are states and edges are transitions among states. Our approach computes a shortest path from state s_0 to state s_{min} on state graph via Dijkstra algorithm [19] and the web application executes corresponding actions to reach state s_{min}.

IV. Evaluation

We have implement WebQT based on Node.js and Python 3.7 with more than 20,000 lines of code. In order to demonstrate our approach, our evaluation answers following research questions:

- **RQ1:** How is the exploration ability of WebQT in terms of code coverage, compared with WebExplor [8], the state-of-the-art web testing tool based on reinforcement learning?
- **RQ2:** How effective is proposed state extraction and identification method? How effective is proposed reward function?
- **RQ3:** How effective is WebQT in testing real-world web applications?

A. Experiment Setup

Dataset. To answer research question RQ1 and RQ2, we utilize web applications in existing works [5], [8], [9]. One of these application, *pagekit*, cannot be instrumented. Applications *gadael*, *mean-blog* and *webogram* are not maintained and cannot be built. We also include two web application *management* and *management* from GitHub. In total, we perform our evaluation on 7 web application. We instrument each web application by *nyc* [20]. To answer RQ3, we randomly select 11 real-world web applications from top 50 web applications according to [14]. In order to demonstrate scalability of WebQT, we directly perform WebQT on these web applications without fine-tuning.

Configuration. To answer research question RQ1, we select state-of-the-art testing tool WebExplor [8] as baseline, which has been proved in existing works [8], [9]. We perform WebQT and WebExplor on each web application five times and measure average branch and line coverage. To answer research question RQ2, we implement following two baselines: (a) $WebQT_{se}$: extract and identify states by URLs and tag sequences and equipped with other components of WebQT; (b) $WebQT_r$: replace reward model component of WebQT with the one of WebExplor. We compare WebQT with above two baselines and measure line coverage. In all experiment, we set a time budget for each tool, i.e., 15 minutes. We set DOM element similarity threshold and state similarity threshold is 0.8 and 0.85, respectively. Reward weights are set as follows: $w_{loc} = 10$, $w_{attention} = 50$, $w_{freq} = 5$ and $w_{explore} = 5$. Hyperparameter M and N is set to 10000 and 100, respectively. Since similarity thresholds and reward indicator weights are important to our approach, we determine them by several tests. Hyperparameters are not optimized. In order to avoid threats introduced by above parameters, these parameters are set equally in each tool.

TABLE III
CODE COVERAGE RESULT

App	Branch coverage		Line coverage	
	E	Q	E	Q
timeoff [21]	13.98%	39.87%	6.54%	48.27%
dimeshift [22]	14.72%	39.17%	18.94%	46.31%
splittypie [23]	7.69%	45.16%	32.95%	64.67%
phoenix [2]	22.37%	69.74%	24.25%	79.10%
hospital [24]	-	30.97%	-	82.35%
retroboard [25]	22.81%	60.23%	58.19%	83.82%
petclinic [26]	0%	85.00%	41.67%	95.83%
Average	**11.65%**	**52.88%**	**26.08%**	**71.48%**

TABLE IV
EVALUATION ON REAL-WORLD WEB APPLICATIONS

App	Client	Server	Total
www.qq.com	2	3	5
www.tmall.com	4	7	11
www.baidu.com	5	1	6
www.taobao.com	3	1	4
www.sohu.com	1	8	9
www.bing.com	4	1	5
www.amazon.com	8	4	12
www.ebay.com	0	1	1
www.aliexpress.com	1	6	7
www.360.com	6	0	6
www.reddit.com	3	0	3
Total	**37**	**32**	**69**

B. The Ability of Exploration

The result of coverage comparison between WebExplor and WebQT is shown in Table III, where column *E* and *Q* denotes WebExplor and WebQT, respectively. Since WebExplor cannot perform on application *hospital*, we denote coverage measured on it as -. As shown in Table III, WebQT performs better than WebExplor. After running 15 minutes, WebQT achieves 41.23% more branch coverage and 45.4% more line coverage than WebExplor on average. The results shows WebQT better exploration ability on web applications, compared with state-of-the-art testing tool WebExplor.

C. Effectiveness of State Extraction Identification

We demonstrate how effective state extraction and identification method is by comparing WebQT with $WebQT_{se}$. The result is shown in Figure 6, where the x-axis and y-axis represents the testing time and line coverage, respectively. Note that, some actions, e.g., clicking "Sales" that transits application *timeoff* from employee-calendar page to team-view page, can trigger much code and thus causing the sudden increasing of line coverage in the figure.

As shown in Figure 6, WebQT achieves higher exploration efficiency than $WebQT_{se}$ (i.e., red line arises faster than blue line). As discussed in Section II, existing approaches cannot tolerate redundancy in web applications and causes redundant states, which degrades exploration efficiency. In addition, in our evaluation, we find that, with the increasing number of states, it is possible for state extraction by tag sequences to incorrectly treat two different states s_a and s_b as the same state. This is because only taking tag sequences into consideration loses too much information of web pages to identify states. Consequently, at state s_b, $WebQT_{se}$ chooses to perform the action that is executable at state s_a but non-executable at state s_b. As a result, $WebQT_{se}$ (blue line) achieves lower line coverage than WebQT (red line).

D. Effectiveness of Reward Model

We demonstrate how effective reward model is by comparing WebQT with $WebQT_r$. The result is shown in Figure 6.

Compared with $WebQT_r$, our approaches benefits WebQT in two aspects. First, compared with $WebQT_r$ (black line), WebQT achieves higher exploration efficiency (i.e., red line arises faster). Second, WebQT achieves higher coverage. For example, in application *phoenix*, there is a page transition sequence: homepage → my-boards → board-list → new-board → board → edit-board-info page. Only visiting edit-board-info can increase coverage. Any interruption leading to another web pages during such a transition sequence degrades exploration efficiency. Since WebQT is prone to accesses to actions that newly appear in the current page, e.g., clicking edition icon in board page, it increases the possibility to explore diverse pages.

E. Scalability

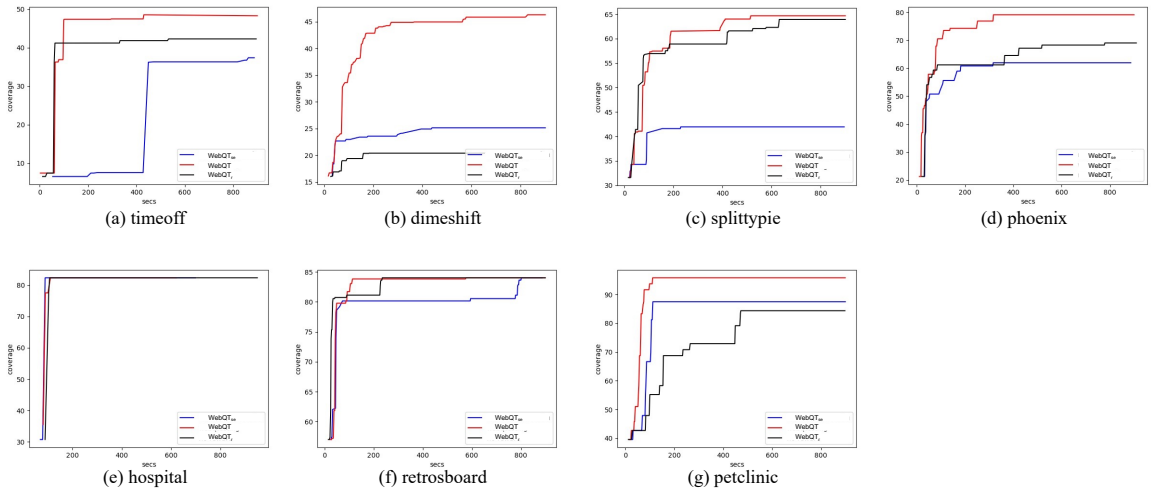
In this section, we demonstrate the scalability of WebQT on real-world web applications. We perform WebQT on 11 web applications randomly selected from top 50 web applications [14] and catch exception message in the console. As a result, WebQT discovers 1,091 exceptions. We manually inspect these exceptions and find most of them are duplicate. For example, when the connection is reset, multiple exceptions is reported. After filtering duplicate exceptions, we obtain 69 exceptions in total, as shown in Table IV, where column *Client* and *Server* denotes the number of exceptions found in client and server side, respectively.

As shown in Table IV, we find exceptions can happen in both client side and server side. In addition, we reveal a wide range of exceptions, including net related errors, resource loading errors, cross-domain errors and JavaScript errors, which demonstrates the scalability of WebQT.

F. Threats to Validity

Representativeness of our studied web applications. We select a number of web applications to demonstrate effectiveness of WebQT in our evaluation. First, these experimental projects come from real-word web applications. Second, these applications have been widely used in existing works [5], [8], [9]. Therefore, we believe our studied web applications are representative.

Evaluation configuration. Randomness can threat validity of our evaluation. We alleviate this threat by repeating five runs for each tool. Another threat is the measurement of our evaluation. WebQT is a black-box testing approach and we cannot access to the server-side code. Therefore, we only



(a) timeoff (b) dimeshift (c) splittypie (d) phoenix

(e) hospital (f) retrosboard (g) petclinic

Fig. 6. Evaluation of WebQT, Web_{se} and $WebQT_r$ regarding code coverage.

collect client-side code coverage, which is a common practice in existing works [5], [8], [9]. Finally, we only monitor exception messages and do not take other types of failures into consideration. Other types of failures detection can be integrated into WebQT in the future.

State identification. Our approach depends on similarity thresholds to determine whether two states are same. It is possible to identify a state as a new state incorrectly, introducing state redundancy. To alleviate this threat, we optimize thresholds by several tests. As a result, WebQT still separates states in much scenarios and thus increasing the efficiency of state space exploration, as demonstrated in the evaluation. Moreover, since test cases generated by state-of-the-art tool WebExplor [8] are less diverse than ours, it is unfair to compare states identified by these two approaches. Therefore, we do not conduct such an evaluation.

V. Related Work

Random based test case generation. Random based approaches [4], [16], [27] analyze candidate actions and randomly execute one of them. Although they have been widely adopted, they are prone to generate invalid test cases, e.g., filling values in buttons. In addition, since they are often interrupted by the randomly selected action, they hard to explore states that can only be reached by valid action sequences.

Model based test case generation. Model based approaches [6], [28]–[30] first dynamically or statically build the model to depict behaviors of the target web application. Then, they design strategies to search paths on the model to generate test cases. For example, SubWeb [6] leverages Page Object [31] defined by developer to build the navigation model, and designs a set of genetic operators to generate test inputs and feasible navigation paths. DIG [28] pre-selects the most promising candidate test cases based on their diversity from previously generated test cases. FragGen [30] ranks actions, and chooses the next state based on the total score of actions in each state. Model based approaches are also widely applied on Android applications [7], [32]–[39]. However, model based approaches are neither complete nor sound.

Reinforcement learning based test case generation. Recently, reinforcement learning is applied to software testing. For example, at the same time winning the game, Wuji [40] explores the state space of the game. QTesting [13] leverages LSTM to divide scenarios with different functionalities and adopts Q learning algorithm to generate test cases for Android applications. Similarly, UniRLTest [9] utilizes CNN to identify states and interactable actions, and adopts DQN to generate test cases for both web applications and Android applications. WebExplor [8] generates test cases for web applications, which identifies states by URL and tag sequences without tolerance of redundancy. Consequently, it yields the large state space and degrades the exploration efficiency. In addition, it inefficiently explores state space only under the guidance of the number of executions. On the contrary, our approach design a novel reward model to guide the agent to generate human-like test cases with high efficiency, as demonstrated by our evaluation.

VI. Conclusion

Web applications are increasingly popular and greatly impact our daily life. However, it is challenging to maintain web applications with high quality. In this paper, we propose WebQT, an automatic test case generator for web applications based on reinforcement learning. Specifically, we present a new state abstraction technique to avoid state redundancy, and design a novel reward model to encourage reinforcement learning agent to mimic human behaviors to explore the state space. We evaluate WebQT on real-world web applications and experimental results show it outperforms state-of-the-art tool with higher effectiveness and efficiency.

VII. Acknowledge

This work was partially supported by National Natural Science Foundation of China U20A6003, China Southern Power Grid Company Limited under Project 037800KK52220005.

REFERENCES

[1] "Web server survey," 2022. [Online]. Available: https://news.netcraft.com/archives/2022/07/28/july-2022-web-server-survey.html

[2] "Phoenix," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/tree/master/fse2019/phoenix

[3] Y. Li and O. Riva, "Glider: A reinforcement learning approach to extract UI scripts from websites," in *Proceedings of International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, 2021, pp. 1420–1430.

[4] A. M. andArie van Deursen and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 35–53, 2011.

[5] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 142–153.

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

[7] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 245–256.

[8] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2021, pp. 423–435.

[9] Z. Zhang, Y. Liu, S. Yu, X. Li, Y. Yun, C. Fang, and Z. Chen, "Unirltest: Universal platform-independent testing with reinforcement learning via image understanding," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 805–808.

[10] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of International Conference on Software Engineering (ICSE))*, 2020, pp. 186–197.

[11] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *Proceddings of International Symposium on Softwar Reliability Engineering (ISSRE)*, 2013, pp. 278–287.

[12] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Clustering-aided page object generation for web testing," in *Proceedings of International Conference on Web Engineering (ICWE)*, 2016, pp. 132–151.

[13] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 153–164.

[14] "Top web sites rank list," 2022. [Online]. Available: https://www.alexa.com/topsites

[15] "Jpetstore demo," January 1, 2023. [Online]. Available: https://petstore.octoperf.com/actions/Catalog.action

[16] "Monkey," 2022. [Online]. Available: https://developer.android.com/

[17] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[18] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[19] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Macmillan Education UK, 1976.

[20] "Nyc," 2022. [Online]. Available: https://istanbul.js.org/

[21] "Timeoff-management-application," 2022. [Online]. Available: https://github.com/timeoff-management/timeoff-management-application

[22] "Dimeshift," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/blob/master/fse2019/dimeshift/README.md

[23] "Splittypie," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/tree/master/fse2019/splittypie

[24] "Hospital-management-nodejs," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/blob/master/fse2019/dimeshift/README.md

[25] "Retroboard," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/tree/master/fse2019/retroboard

[26] "Petclinic," 2022. [Online]. Available: https://github.com/matteobiagiola/FSE19-submission-material-DIG/tree/master/fse2019/petclinic

[27] 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.

[28] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 142–153.

[29] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 245–256.

[30] R. K. Yandrapally and A. Mesbah, "Fragment-based test generation for web apps," *IEEE Transactions on Software Engineering (TSE)*, 2022.

[31] "Page object," 2022. [Online]. Available: https://martinfowler.com/bliki/PageObject.html

[32] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2014.

[33] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *Proceedings of IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 1–6.

[34] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, M. Goedicke, T. Menzies, and M. Saeki, Eds., 2012, pp. 258–261.

[35] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 238–249.

[36] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 103–114.

[37] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 115–127.

[38] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 94–105.

[39] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: Generating high-quality test inputs for Android apps via use case combinations," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2020, pp. 469–480.

[40] 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 *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.