Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

QExplore: An exploration strategy for dynamic web applications using guided search[☆]

Salman Sherin [*], Asmar Muqeet, Muhammad Uzair Khan, Muhammad Zohaib Iqbal

Quest Lab, Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan
UAV Dependability Lab, National Center of Robotics and Automation (NCRA), Pakistan

ARTICLE INFO

Article history:
Received 16 June 2021
Received in revised form 5 July 2022
Accepted 12 September 2022
Available online 24 September 2022

Keywords:
Dynamic exploration
Web application testing
Model generation
Guided search
Coverage
Automated testing
Q-learning
Reinforcement learning

ABSTRACT

Dynamic exploration approaches play an important role in automating web testing and analysis. They are extensively used to explore the state-space of a web application for achieving complete coverage of the application's functionality. Dynamic exploration approaches support end-to-end automation of testing to verify the correct behavior of a web application. However, existing approaches failed to explore the states behind the web forms and can get stuck in dynamic regions of web applications resulting in poor functionality coverage and diversity. Consequently, existing approaches are regressive in nature and sensitive to small DOM mutations which may not be interesting from a testing perspective. In this paper, we propose a dynamic exploration approach using guided search inspired by Q-learning that systematically explores dynamic web applications requiring less or no prior knowledge about the application. Our approach is implemented in a tool called QExplore and is empirically evaluated with six popular open-source and one real industrial application. The results show that QExplore achieved higher coverage with more diverse DOM than the existing state-of-the-art tools Crawljax and WebExplor. QExplore also results in a greater number of navigational paths, error states and distinct DOM states when compared with the existing tools.

1. Introduction

Modern web applications are highly dynamic in nature offering better responsiveness and interactivity to users. The highly dynamic nature of modern web applications is due to the extensive use of JavaScript allowing dynamic updates to the Document Object Model (DOM). This dynamic nature increases the complexity of testing these applications and make the dynamic exploration indispensable (Van Deursen et al., 2015). Dynamic exploration approaches explore the state-space of a web application by exercising the possible interface elements. During the exploration, a State-Flow Graph (SFG) is automatically constructed that is used to automate different web application analysis and testing tasks. For example, tasks such as invariant-based testing (Mesbah et al., 2011), cross-browser compatibility testing (Mesbah and Prasad, 2011), regression testing (Mirshokraie and Mesbah, 2012), broken link detection (Beroual et al., 2017), state-based testing (Marchetto et al., 2008) and system testing (Tanida et al., 2011) can be automated using the SFG. The effectiveness of testing activities such as test sequence generation, execution and evaluation are highly dependent on the effectiveness of SFG constructed during the exploration of web application. Similarly, dynamic exploration approaches assist exploratory testers in analyzing accessibility (Ferrucci et al., 2011), security (Muñoz et al., 2018; Park et al., 2017) and test dependency analysis (Biagiola et al., 2020) of web applications. Moreover, dynamic exploration approaches are used by search engines to aid the indexing of the ajax based web applications (Mesbah and Van Deursen, 2008; Duda et al., 2009), archiving web pages (Brunelle et al., 2016; Faheem, 2012) and data mining to perform data analytics (Ye and Li, 2017).

There exist several dynamic exploration approaches in the literature that can be classified broadly into two types i.e., generic exploration approaches (e.g., Crawljax Mesbah et al., 2008) and feedback directed or guided exploration approaches (e.g., FeedEx Fard and Mesbah, 2013). Generic exploration approaches are mostly used to explore unknown applications where the tester has less or no prior knowledge of the web application (Liu et al., 2020). Generic exploration approaches such as state-of-the-art Crawljax explores the dynamic web application by exercising the DOM elements at the browser side constructing a state-flow graph as an output. But such approaches use exhaustive

[☆] Editor: Burak Turhan.
[*] Corresponding author at: Quest Lab, Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan.
E-mail addresses: salman.sherin@questlab.pk (S. Sherin), asmar.muqeet@questlab.pk (A. Muqeet), uzair.khan@nu.edu.pk (M.U. Khan), zohaib.iqbal@nu.edu.pk (M.Z. Iqbal).

https://doi.org/10.1016/j.jss.2022.111512
0164-1212/© 2022 Elsevier Inc. All rights reserved.

search algorithms such as DFS or BFS which leads to state explosion problem due to inadequate DOM equality function and having no feedback to guide the crawl process during the exploration (Moosavi Byooki, 2014). Moreover, these approaches get trapped in certain specific regions of the dynamic web applications resulting in inadequate functionality coverage (discussed in Section 2 in detail). In contrast, the feedback directed or guided exploration approaches derive a partial test model (state-flow graph) by guiding the exploration process manually to reduce the state explosion problem. For example, FeedEx (Fard and Mesbah, 2013) uses a combination of code coverage, navigational diversity and structural diversity to guide the exploration during the execution. However, the user has to manually set the values of these parameters to guide the exploration, which requires several runs of the application to reach the optimal values. Similarly, Keyjaxtest (Qi et al., 2019) explores a web application to derive a partial test model by using specific keywords that describe specific functionality in the application. Without knowing appropriate keywords, Keyjaxtest may not accurately explore the target applications' functionality. This implies that these approaches require manual efforts and prior knowledge of the web application under test to guide the exploration process. This limitation makes the guided exploration approaches ineffective when given an unknown application. Although, one of the key advantages of dynamic exploration approaches is to automatically explore new or unknown applications (Liu et al., 2020).

There are several other limitations common to both generic and guided exploration approaches. Most of the existing approaches use manual or random data input. Manual data inputs limit the scalability of dynamic exploration approaches and is time consuming. Similarly, random data inputs fail to pass the validation constraints in a web form leaving several DOM states unexplored. Consequently, existing dynamic approaches are regressive in nature which means that every time a tester stops the exploration process for the required user-specific input (e.g., email or password), the approach starts from the initial state without saving the previous crawl session. To define user-specific input, the existing approaches allow writing crawl rules by locating the target DOM elements either by using ID or absolute XPath. This requires the user to read the source code of the application, which not only is tiresome and difficult but also requires the user to have knowledge of the application code. There is a need for a dynamic exploration approach that can reduce the limitations of the existing approaches. The goal is to automatically explore the state-space of an unknown environment to achieve better functionality coverage and DOM diversity alleviating the limitations of the existing dynamic exploration approaches.

Therefore, we propose an exploration strategy for dynamic web applications using guided search inspired from Q-Learning (a reinforcement learning approach). Our approach incrementally explores new or unknown dynamic web applications by guiding the exploration process using a reward function and constructing a state-flow graph as an outcome. The approach uses the contextual data method to feed the input fields during the exploration and uncovers the states behind web forms having validation constraints. The contextual data method uses NLP techniques to map the data on the target input fields by capturing the context of the fields at run time. For the input data, we use an existing open-source data generator called Mocker Data Generator,[1] which provides input data for the commonly used input fields (e.g. name or address) in web applications. Our approach allows defining user-specific inputs by using keywords, thus, reducing the difficulty of reading the application code for testers. It also allows testers to define multiple values for the same input field that enables the exploration of a dynamic web application with multiple views and different roles (e.g., administrator or regular user). Furthermore, our approach performs incremental exploration by allowing testers to interrupt and resume the exploration process in order to define user-specific inputs to uncover certain DOM states.

The main contributions of this paper include:

- Dynamic exploration strategy for dynamic web applications, which has the following features.
    - A guided exploration approach inspired by Q-learning for dynamic web applications
    - Contextual input data method for the web forms to maximize coverage and enable the completion of scenarios.
    - An integration of mocker data generator for data input in the web forms.
    - Building a state flow graph that can be used for automating many types of web analysis and testing techniques.
- Implementation of the approach in a tool called QExplore which supports the automatic exploration of modern web applications.
- Empirical Evaluation to analyze the efficacy of the proposed approach.

The organizational structure of this paper is as follows: Section 2 provides the background and motivation. Section 3 provides the proposed approach. Section 4 describes the implementation details of the proposed approach. Section 5 presents an empirical evaluation. Section 6 provides a discussion on the results and findings of the study. Section 7 discusses the different threats to the validity of the results. Section 8 presents the related work and finally, Section 9 concludes the study and discusses the future works.

2. Background & motivation

Dynamic web applications are increasingly becoming complex due to modern web technologies and programming languages offering more flexibility and interactive user interfaces. Dynamic web applications make extensive use of JavaScript that allows changes to the Document Object Model (DOM) on the fly without refreshing the page. Thus, it provides responsiveness and stateful behavior to the end-user, but (or while) poses substantial challenges to the testing of these applications (Van Deursen et al., 2015). To cope up with these challenges, a number of dynamic exploration approaches are proposed in the literature (Mesbah et al., 2008; Fard and Mesbah, 2013; Liu et al., 2020; Qi et al., 2019; Dincturk et al., 2012; Benedikt et al., 2002). These approaches dynamically explore the state space of a web application by exercising interactive DOM elements to reach the number of possible DOM states. During the dynamic exploration of a web application, a state flow graph is generated which is used for automating various testing approaches and web analysis (Mesbah et al., 2011; Marchetto et al., 2008; Tanida et al., 2011; Yandrapally et al., 2020). However, the existing approaches have several challenges and limitations discussed in detail as follows:

2.1. Challenges and limitations in the existing dynamic exploration approaches

- A simple dynamic web application (e.g., ToDo list) can have a huge state space, which makes the exploration of complete state space infeasible, i.e., leads towards a state explosion

[1] https://github.com/danibram/mocker-data-generator

```
<body>
<div style="width:600px; margin:0 auto;">
       <img class="image_con" id="testImg" src = "1.jpg" width="600" height="500">
       <div style=" text-align: center; margin: 5px;">
       <div style="font-size: 20px;" id = "container"></div>
</div><div style = "width: 300px; margin: auto; text-align: center;" >
       <a href="#" class="next round" onclick = "testImageUpdate(-1)">Prev</a>
       <a class="previous" style = "width:82px" onclick =
"dataUpdate();">Update</a>
       <a class="next round" onclick = "testImageUpdate(1)">Next</a></div>
<body>
<script>
       let myImg = ["1.jpg","2.jpg","3.jpg","4.jpg"];
       let indexNow = 0;
       function testImageUpdate(offset){
              indexNow += offset;
              if(indexNow > 3){
                     indexNow =1}
              else if(indexNow<0){
                     indexNow=4}
              document.getElementById("testImg").src = myImg[indexNow]; }
       function dataUpdate(){
              $.ajax({
                     url: "/test",
                     type: "GET",
                     success: function(data){
                     document.getElementById("container").innerHTML = data.move; },
                     error: function (xhr, status, error) { alert("Error!"); }
              }); }
</script>
```

Fig. 1. Motivational example.

problem. This is due to the use of inadequate DOM equality function and having no feedback to guide the crawl process during the exploration (Moosavi Byooki, 2014). Similarly, in any realistic software development environment, the amount of time given to testing is always time-bound. Therefore, a challenge for the existing approaches is to derive a partial test model out of the whole state space that should provide reasonable coverage of the applications' functionality within the given time (Van Deursen et al., 2015; Li et al., 2014). Furthermore, existing dynamic approaches that exhaustively explore the dynamic state-space can become mired in specific irrelevant regions of the web applications, resulting in inadequate functionality coverage and duplicate states in SFG (Fard and Mesbah, 2013). For example, consider the code of a simple dynamic gallery in a web application, as shown in Fig. 1. When a user clicks on the Next button, the image changes and a different description with a random number is shown on the image. Every time a user clicks on the next or previous button, a different DOM state appears which is added to the SFG. In the case of Crawljax, every DOM state is a different state due to the dynamic description with the random number displayed on the image. Though, such small DOM mutations become irrelevant from a testing perspective. Crawljax uses string matching, Levenshtein distance, to compare states which is sensitive to small changes in the DOM. This is one such typical example in which the existing crawlers get mired resulting in a large number of similar or duplicate states in the SFG.

- Existing dynamic exploration approaches (e.g. Crawljax and FeedEx) are limited to providing either manual or random data where input from the user is required (e.g., Name or Email) during the exploration. On one hand, manual data input limits the use of dynamic exploration approaches. On the other hand, random data failed to pass various validation constraints (client-side or server-side) due to which several DOM states behind the web forms remain unexplored causing inadequate functionality coverage.
- Most of the existing exploration approaches allow users to define specific user input to gain access to a specific part of the web application. For example, it is really difficult for an exploration approach to randomly guess the right pair of username and password to log into the application. Therefore, it is important to allow users to define values for such specific inputs. However, existing approaches allow defining such inputs by identifying the ID or XPath of the target input field. This requires the user to read the source code of the application, which is not only tiresome and difficult but also requires the user to have knowledge of the application code. Similarly, a web application can have multiple users having specific roles and functionalities. For example, consider a web application with two kinds of users, an administrator and a student, both having different functionalities to perform. In this case, if an approach accepts only one value for a specific user input field then it will limit the crawler to explore only a part of the web application. However, existing approaches such as Crawljax and FeedEx allow defining only a single value for user-specific input resulting in limited functionality coverage. In other words, existing approaches do not support testers to define multiple values for the same input field.
- Existing dynamic exploration approaches such as Crawljax and FeedEx allows user to write specific crawl rules for specific input (as discussed earlier). This means that the user must be familiar with the application under exploration. If the user is not familiar with the application, the states explored by any dynamic exploration approach will be limited. The workaround to get familiar with the application is to run Crawljax on the target application for several rounds (Liu

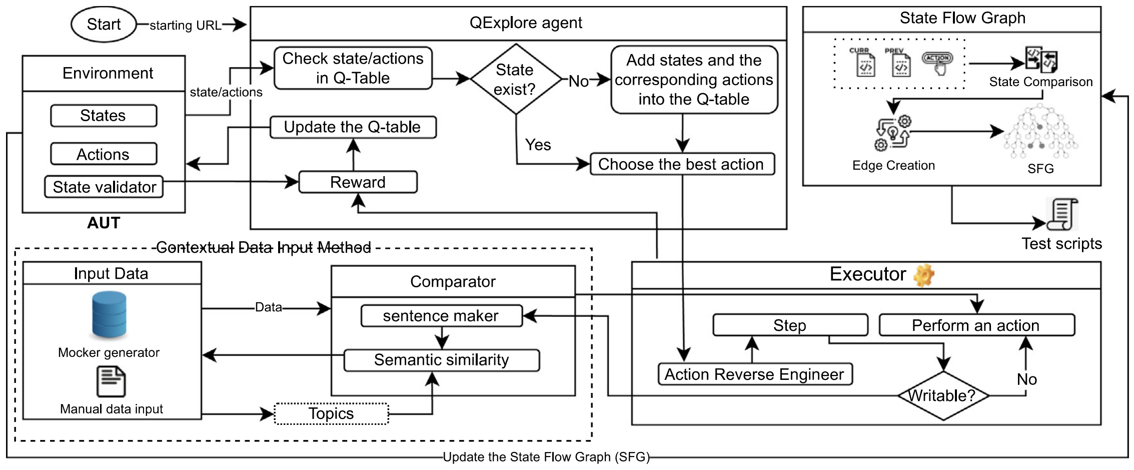


Fig. 2. An overview of the proposed guided exploration approach.

et al., 2020). The user identifies the required crawl rules by looking at the state flow graph obtained from the previous round and then define the necessary crawl rules for the next round. However, starting every crawling round to explore the application from the beginning is tedious and time-consuming. The challenge for the existing crawlers is to explore the target web application incrementally preserving the state of crawling in the previous round and resuming the next round from there onward. This concept is known as incremental crawling and a recent paper by Liu et al. (2020) proposed such an approach (called GUIDE) that focuses on scenario-based exploration when given concrete goals. In contrast, approaches like Crawljax (Mesbah et al., 2008) and FeedEx (Fard and Mesbah, 2013) (including the one in this paper) are different in nature, which works around the idea of exploring the application without having any user-defined concrete goals/scenarios.

- Most of the guided approaches are not effective in exploring the state-space without prior knowledge of the application under test. For example, the effectiveness of FeedEx depends on the different weights set before crawling the application. However, those weights are difficult to guess before crawling and would require to be executed several times on the application to reach the best combination for those weights. Similarly, the keyajxTest (Qi et al., 2019) not only requires different probabilistic weights to be known before crawling but also knowledge about the different functionalities available and their relevant keywords. Such guided approaches cannot be used where the user has no prior knowledge of the application under exploration. The challenge for the existing approaches is to crawl the application requiring minimal or no prior knowledge of the application under exploration.

2.2. *Q-learning in brief*

Q-learning is a well-known reinforcement learning approach in which the agent learns to perform optimally in an unknown environment with the help of trial-and-error interactions (Watkins, 1989). For Q-learning to converge towards an optimal policy, the environment should be modeled as a Markov Decision Process (MDP). In MDP, each state is a Markov state satisfying the memoryless property also known as markovian property. In every interaction, the agent takes an action on the basis of the current state in the environment and computes the significance of that action based on its immediate reward. The overall goal of an agent is to act in a way that maximizes the cumulative reward. This implies that the agent learns the best strategy for maximizing the sum of the rewards in a long run. Q-learning algorithm uses the following equation as a Q-value update function for convergence towards an optimal policy, as shown in Eq. (1).

Q-value update equation:

$$Q\ (s, a) \longleftarrow Q\ (s, a) + \alpha\ \left(\mathcal{R} + \gamma.max_{a'} Q\ \left(s', a'\right) - Q\ (s, a)\right) \quad (1)$$

In which $\mathcal{R}$ is the current reward and $\gamma.max_{a'}Q(s', a')$ is the product of the discount factor and expected future reward of action in the environment. The discount factor is represented as γ and its value varies between 0 and 1. When the value of γ is closer to 0 than the agent emphasis on the immediate reward and when it is closer to 1, then the agent emphasis on maximizing the future reward. The learning rate (α) defines the magnitude of the step taken towards the optimal policy. There are several conditions required for the convergence of the Q-learning i.e., Robbins–Monro sequence or GLIE condition for optimal convergence. Our proposed approach for exploration of dynamic web application is inspired by Q-learning. The goal of the approach is to explore the state-space of dynamic web application effectively, therefore, prioritizing exploration over exploitation. This means the Q-learning is adopted to be used as a guided search rather than a learning agent not necessarily requiring to meet the conditions of the convergence. The approach automatically explores the state-space of a dynamic web application to derive the test model that can be used to automate several other web testing and analysis techniques. Similar approaches inspired from Q-learning exist in other domains such as mobile (Adamo et al., 2018) and GUI testing of desktop applications (Mariani et al., 2011).

3. Proposed solution

3.1. *An overview*

We introduce a guided exploration strategy for dynamic web applications that effectively explore different parts of the web and construct the state flow graph as an output, as shown in Fig. 2. The Application Under Test (AUT) represents the environment for the agent where it exercises different states by performing actions available at a given state. To make it easy to understand, we are using the key concepts from the Q-learning to explain the working mechanism of QExplore.

To begin exploration, QExplore takes the URL of the web application to construct the exploration environment. It interacts with the environment (i.e., web application) to gather information of states and the available actions at each particular state. In each interaction, the agent chooses the best action to execute and receives the feedback (reward) from the environment to calculate the Q-value for a particular action at that state. The best action for next state is influenced by two main factors. The first is the number of times this specific action is executed and second the number of interactable elements in the resultant state, the higher the number of interactable elements the better the previous action was. The executor than takes the best-chosen action (interactable web element) to execute it in the given state. If the action is writable (e.g., input field) then the executor gets the input data from contextual data input method and performs the action in the environment. If the action is clickable than the executer executes it directly. During the execution, a state-flow graph is constructed as an outcome which is used for further test automation.

3.2. Representation of states and actions

QExplore evaluates the resulting reward for each action at a given state and attempts to maximize the cumulative reward. We dynamically detect different interactive DOM elements and their type (with which a user can interact such as buttons, checkboxes, input fields and dropdown lists etc.) by parsing the DOM tree during the execution. The interactive DOM elements are uniquely identified by Name, href, value and other attributes based on what is associated with the DOM element. This information forms the basis for the definition of action and state.

Definition. An action can be represented as a 5-tuple: $a\left[\tau, \mathcal{n}, \mathcal{v}, \hbar_{ref}\right]$ where τ is the tag name (e.g., input), $\mathcal{n}$ is the name attribute, $\mathcal{v}$ is the value attribute and $\hbar_{ref}$ is the *href* attribute of the DOM element. The $\hbar_{ref}$ holds the target link if the DOM element is <a>. The $\mathcal{v}$ holds an arbitrary string/text if the DOM element is of type ‘text’. The *href* value for all the non-anchor tags is empty. Similarly, the value of $\mathcal{v}$ for all the non-text interactive DOM elements is empty. Based on these attributes, we transform a DOM element into an intermediate string representation which is later on used for generating dynamic XPaths. In the consequent sections of this paper, we refer to these actions as DOM actions or interactable elements.

According to our definition of action, an interactive DOM must have a tag name and ‘name’ or ‘value’ attribute to uniquely identify that DOM element. It is important to note that ID is not selected to uniquely identify DOM actions because in a dynamic web application the same DOM element can have different IDs changing dynamically every time a web page is loaded or refreshed. Therefore, a unique ID may not exist for the DOM elements in dynamic web applications.

Definition. A DOM state can be represented as an n-tuple: $d_s = \left(d_{a_1}, d_{a_2}, d_{a_3}, \ldots, d_{a_n}\right)$ where d_{a_i} is an action and n is the number of unique actions at a given DOM state. In other words, the state is represented as the collection of attribute values of the interactable DOM elements of the web application. The DOM state definition in our case is different than the previous dynamic exploration approaches. The intuition is that a DOM state with the same interactable elements consolidates the same business logic and is considered as one state. This means that for QExplore the state remains similar as far as it is comprised of the same interactable DOM elements. A state is said to be invalid if it is not in the scope of the current crawling session. For example, an action that redirects the agent from the existing web application to another external source (having a different base URL), which is not in the scope of the current crawling process is said to be in an invalid state.

3.3. Reward function

We dynamically model the web application under test with the finite set of DOM states $\mathrm{D_s}$ and a finite set of actionable DOM elements $\mathrm{D_a}$. For a given state QExplore chooses an action and passes it to the executor to execute it from the available set of actions at that state. The goal of the agent is to maximize the DOM diversity by exploring new states with the passage of time. Therefore, the reward function should be defined in such a way that the agent gives priority to execute new actions instead of actions executed in the past. The rationale behind executing new actions is to explore new DOM states and increase the DOM coverage. To calculate the reward for a given actionable DOM element at a given DOM state, we define R for taking DOM action $\mathrm{d_a}$ in DOM state $\mathrm{d_s}$ which leads to $\mathrm{d_{s'}}$, as follows:

Reward function:

$$R\left(d_a, d_s, d_{s'}\right) := \begin{cases} R_{max} & if\ x_{d_a} = 0 \\ \frac{1}{x_{d_a}} & if\ x_{d_a} > 0 \\ R_{negative} & if\ d_s \neq valid \end{cases} \tag{2}$$

In the above equation, x_{d_a} is the number of times action d_a is executed by the executor and R_{max} is the maximum reward given to the action d_a. The action is given a negative reward of $R_{negative}$, if the resultant state is invalid. The negative reward is intended to penalize the agent for reaching an invalid state in the environment. The reward for performing an action is inversely proportional to the number of times that action is performed before. The agent is rewarded with the maximum reward of R_{max} for an action executed for the first time. The value of R_{max} can be set to any arbitrary positive value at the beginning. In our case, the value of R_{max} is set to 500. After each time step, the reward value for the same action on the same state is inversely proportional to the number of times that action is executed.

3.4. Q-value update in QExplore

The exploration agent interacts with the environment (i.e., web application) to gather information of states and the available actions at each particular state. In each interaction, the agent chooses the best action to execute and receives the feedback (reward) from the environment to calculate the Q-value for a particular action at that state. We have taken the concept of immediate and future reward from conventional Q-learning, the selection of the best action in a particular state is not only influenced by the immediate rewards but also by the effect of that action on the future states. In conventional Q-learning, Q-value function allows the agent to maximize the future rewards while making the choice of an action at a particular state. Capturing the knowledge of how to maximize the cumulative reward in the Q-value function enables the agent to exploit and behave optimally in the environment. For the exploration agent in this approach, the Q-value function (inspired from the original Eq. (1)) is defined as follows:

Q-Value function:

$$\begin{aligned} Q\left(d_s, d_a\right) \longleftarrow\ & Q\left(d_s, d_a\right) + \alpha\left(R\left(d_a^*, d_s, d_{s'}\right)\right. \\ & \left. +\ \gamma\left(d_{s'}, d_A\right).max_{d_a^*} Q\left(d_s, d_a^*\right) - Q\left(d_s, d_a\right)\right) \end{aligned} \tag{3}$$

The Q-value function consists of two factors, i.e., the current reward and the expected future reward. The current reward is calculated by using the first part of the Q-value function ($R\left(d_a^*, d_s, d_{s'}\right)$) and the expected future reward is calculated recursively by using the second part of the equation ($\gamma\left(d_{s'}, d_A\right)$ $.max_{d_a^*} Q\left(d_s, d_a^*\right)$). In conventional Q-learning, the expected future reward with the discount factor balances out the impact of current action on future states. However, in QExplore, we use the notion of expected future reward as a term to define the impact of current action on the leading DOM states with varying discount factors. The intuition is to give higher weight to DOM states that can lead to several new DOM states. A DOM state with a greater number of interactable elements is more likely to lead to new DOM states, as compared to DOM state with fewer interactable elements. QExplore considers this potential of a DOM state in the second part of the Q-value function as an expected future reward. This helps the exploration process to reach new and diverse DOM states. The expected future reward of an action that leads to a DOM state with a larger number of interactable elements will be greater than the action that leads to a smaller number of interactable elements in future DOM states. The goal of the exploration agent is to maximize the cumulative reward while reaching the goal. Therefore, the Q-value is calculated by the sum of immediate action reward and the maximum expected reward that the agent anticipated considering the potential of reaching new DOM states, while executing the current action. In the second part of the equation, the discount rate parameter γ varies in the range of real values from 0 to 1, which balances between an immediate reward and expected future reward. The value approaching 0 gives high weight to the immediate rewards whereas the value approaching 1 gives a high weight to the future rewards.

In conventional Q-learning, the discount factor (γ) is kept constant (for example 0.9). However, the use of a constant value for γ makes it difficult to foresee if the potential of a future DOM state on the current action is beneficial or not. The discount factor is used to decrease or increase the influence of the expected future reward on the current action. However, our goal is to maximize the exploration of the dynamic state-space of the web application via exploitation of Q-value. Therefore, to anticipate the effect of future DOM states on the current action, we use the exponential decay function as a heuristic to calculate the value of the discount parameter. Instead of using a constant value for the calculation of parameter γ, an adaptive value of the discount parameter is calculated based on the changes in the future DOM states. Dynamically calculating the value for γ prioritizes the expected future rewards over the immediate reward when the agent interacts with a state having a small number of DOM actions. The equation for the dynamic discount factor is defined as follows:

Discount factor:

$$\gamma\ (d_{s'}, d_A) = 0.9 \times e^{-0.1\times(|d_A|-1)} \tag{4}$$

Where $|d_A|$ is the number of interactive DOM elements (actions) available in a DOM state $d_{s'}$. This dynamic discount parameter allows the agent to overlook states with a small number of DOM actions and prioritize the DOM states having a large number of DOM actions. In conventional Q-learning the learning rate defines the magnitude of change of current Q-value from the previous value. The value of α in our case is kept constant with the value 1. The intuition behind the maximum learning rate is to maximize the feedback (reward) for effective exploration to reduce redundancy in paths and explore new diverse paths.

Algorithm 1 shows the pseudocode of QExplore. Each state in the Q-table is represented by the hash value calculated from the string representation of the state (Lines 7–8). In QExplore, an episode is the execution of actions in a sequence before starting a new one. The *MaxDepth* parameter defines the maximum number of actions to be considered in one episode. The default value for the *MaxDepth* is 100 in case of QExplore.

At every given state, the agent checks whether the state exists in the Q-table or not. If the given state does not exist in the Q-table, the agent first adds the new state and its corresponding actions and then chooses the best action from the available actions at that state. The dynamic changes in state–action space are incorporated by maintaining the action space across all states to a global set of actions. To retain the state–action space, the Q-table is constructed in such a way that the action space across all states remains global. However, each state will only have a positive Q-value for those actions that are available at that state. For example, we have a table of dimensions $n \times m$ where n is the number of unique actions and m is the number of unique states explored by the agent so far. Thus, whenever a new state $m+1$ is explored, the number of unique actions is re-calculated and the new actions are added to the Q-table. Let suppose the $m+1$ state contains three unique actions ($d_{a_1}, d_{a_2}, d_{a_3}$) that are not already added to the Q-table, these actions will be added for each state in the Q-table. Therefore, the new dimensions of the Q-table will be $(n + 3) \times (m + 1)$. The Q-value of the newly added actions for the new state $(m + 1)$ is set to the initial Q-value whereas the Q-value for other actions not available at this state is set to a negative. Similarly, all the other states will have a negative Q-value for these newly added actions in the Q-table. This means every state contains every possible action, but the negative Q-value restricts the agent from choosing actions not available at that state.

Initially, all the actions are given the same Q-value and the agent chooses the action randomly. The chosen action is passed to the executor to execute the action on the given state. Before the action is performed, the executor reverse engineer's the action to make it accessible for the selenium driver and locate the target DOM element in the DOM tree (Lines 11–12). At this point, the executor identifies the type of action (by using the 'step' component) whether the action requires any input data or not (Line 15). If the action requires any input data, then the executor requests the comparator to return the contextual data from the database and populate the input data into the input field. The comparator is responsible for the identification of the semantic context of the DOM element. If the chosen action does not require any input data (i.e., radio button, checkbox, or button) then it is directly performed by the selenium driver. After the action is performed, the executor calls the reward function to calculate the reward for the target action (Lines 16–19) and the Q-table is updated by updating the Q-value (Lines 30–33). The goal of an agent is to explore the AUT by maximizing the cumulative reward. While exploring the state-space of the application, the state-flow graph is updated simultaneously. The DOM states are represented as nodes and actions are represented as edges in the SFG (discussed in Section 3.7). With every successful action, an edge is created from the source to the target node (Lines 34–36). Similarly, a state is added as a node if it is not already present in the SFG (see algorithm 2). The agent repeats the same steps for every action at a given state until the termination criterion is met (Lines 3–38). In our case, the crawling activity can be either time-bound or the termination criteria can be the number of episodes.

Example of Q-Value update

Consider a web application that has states S, A, B and C as show in Fig. 3. State S has two actions (interactable elements), a1 and a2, state A has three actions i.e., a4, a5 and a6, state B has one action (a3) and C is a leaf node having no actions that leads to another state. We demonstrate three episodes for this example where the termination criteria for each episode is either having a maximum depth of four actions or reaching a leaf node.

The Q-value update in all three episodes is shown in Fig. 4. Initially, the Q-value for all the actions is set to 500. QExplore starts exploration from the S state having two interactable elements, a1 and a2. Since both the actions have the same Q-value, a1 is selected randomly. The QExplore will take action a1 from S

```
Algorithm 1 : QExplore
Input: $The\ target\ web\ application\ (env), ExplorationTime = 0, Q-Table = None, SFG = default\_factory$
Output: $State\ Flow\ Graph\ (SFG)$
1  Initialize
2    $MaxReward = 500$
     $Qlearn(initialReward = MaxReward, \alpha = 1, \gamma = 0.9, Q-Table = \emptyset), start\_state = hash(env.getState())$
3  repeat
4    $saveMatrix()$
5    $saveStatemap()$
6    $reset(env)$
7    $state \leftarrow hash(env.getState())$
8    $Actions \leftarrow hash(env.getActions())$
9    $stateURL \leftarrow env.current\_url$
10   repeat
11     $action \leftarrow Qlearn.get\_best\_action(state, Actions)$
12     $elem \leftarrow env.reverseEngineerAction(action)$
13     $initialize\_NegativeReward \leftarrow -99999$
14     if $elem \neq None$ then
15       $status \leftarrow env.step(elem, depth = MaxDepth)$
16       if $status$ then
17         if $env.BaseURL\ in\ env.current\_url$ then
18           $Qlearn.\_qmatrix.Increment\_count(action)$
19           $reward \leftarrow getReward()$
20         end if
21         if $reward == -inf$ then
22           $next\_state \leftarrow state$
23           $next\_state\_actions \leftarrow Actions$
24         else
25           $next\_state \leftarrow hash(env.getState())$
26           $next\_state\_actions \leftarrow hash(env.getActions())$
27         end if
28       end if
29     end if
30     $\gamma \leftarrow getDiscountFactor(next\_state\_actions)$
31     $maxValue \leftarrow getMaxValue()$
32     $qvalue \leftarrow \alpha(reward + \gamma \times maxValue)$
33     $Qlearn.update\_model(qvalue, state, action, next\_state, next\_state\_actions)$
34     $SFG \leftarrow addState(SFG, state, action, next\_state, start\_state, stateURL)$    ∴ see Algorithm 2
35     $state \leftarrow next\_state$
36     $Actions \leftarrow next\_state\_actions$
37   Until reaches $MaxDepth$
38 Until reachers the time budget
39 Return SFG
```

```
Algorithm 2: Add State
Input: $SFG, state, action, next\_state, start\_state, stateURL$
Output: $SFG$
1  $SFG[state][src] \leftarrow state$
2    $SFG[state][edges].append(action, next\_state)$
3    if $state == start\_state$ then
4      $SFG[state][start] \leftarrow 1$
5    end if
6    $SFG[state][url] \leftarrow stateURL$
7  Return SFG
```

to A and the Q-table is updated according to the Q-value function. The Q-table incorporates the new actions (a4, a5, a6) discovered on state A and an initial Q-value of 500 is assigned to each action in its respective state as shown in ep1-I. An action that belongs to other states will have a Q-value of −9999. For instance, on state S, actions a4, a5 and a6 are assigned a value of −9999 because these actions belong to state A. Similarly, a1 and a2 have value −9999 in state A. After a transition from S to A, the new Q-value for a1 in state S is updated to 369.43. In state A, action a5 is selected at random causing a state transition from A to S as shown in ep1-II. Now in state S, a2 has the maximum Q-value. Therefore, a2 is selected causing a state transition from S to B. Since B has only one interactable element, the Q-value update for a2 will be different than a1 as shown in ep1-III. From state B the only action a3 is selected causing a state transition of B to S as shown in ep1-IV. The same process is repeated for each episode.

Q-value is updated based on immediate reward and future reward. The immediate reward gives priority to newly discovered actions and paths that lead to states having less actionable elements at the beginning as shown in ep1 till ep2-II. From episode 1 to episode 2, QExplore discovers two different paths from S to A and S to B. States in both paths have new actionable elements but the path S to B has less number of actionable elements than path S to A. Therefore, the Q-value function priorities exploring S to B path before S to A in episode 2. The intuition is to explore smaller paths in the first place before going deep into larger paths. This increases the DOM and path diversity and more time budget is spent on larger paths. This behavior can be seen from ep1 to ep2-II. After completing ep2-II, majority of actions are covered, and the second part (future reward) of the Q-value update comes into action. The future reward prioritizes the states and paths that contain more actionable elements so that larger paths can

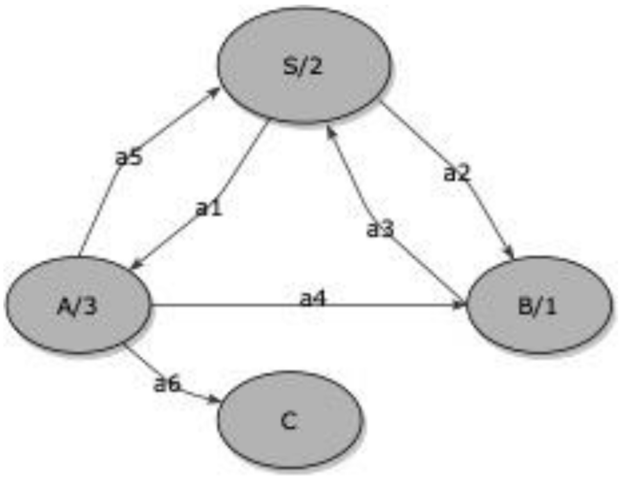
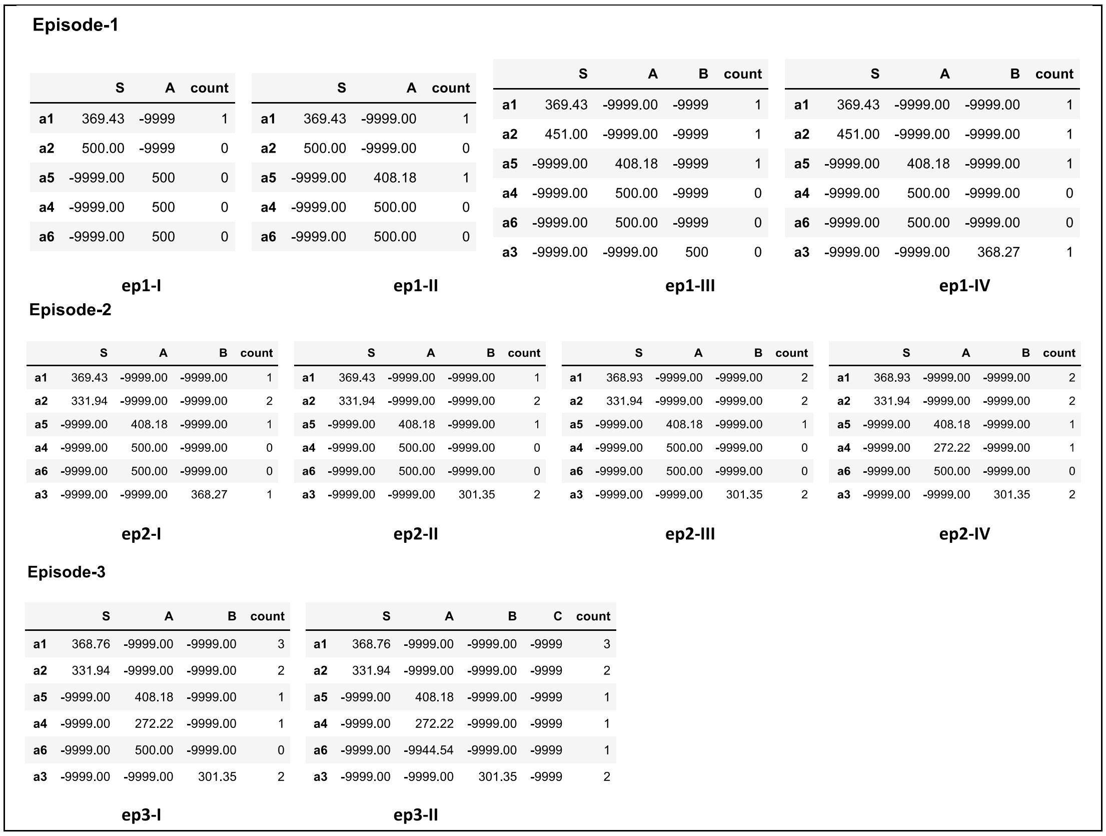


Fig. 3. Example EFG with actions and states.


Episode-1

	S	A	count
a1	369.43	-9999	1
a2	500.00	-9999	0
a5	-9999.00	500	0
a4	-9999.00	500	0
a6	-9999.00	500	0

ep1-I

	S	A	count
a1	369.43	-9999.00	1
a2	500.00	-9999.00	0
a5	-9999.00	408.18	1
a4	-9999.00	500.00	0
a6	-9999.00	500.00	0

ep1-II

	S	A	B	count
a1	369.43	-9999.00	-9999	1
a2	451.00	-9999.00	-9999	1
a5	-9999.00	408.18	-9999	1
a4	-9999.00	500.00	-9999	0
a6	-9999.00	500.00	-9999	0
a3	-9999.00	-9999.00	500	0

ep1-III

	S	A	B	count
a1	369.43	-9999.00	-9999.00	1
a2	451.00	-9999.00	-9999.00	1
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	500.00	-9999.00	0
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	368.27	1

ep1-IV

Episode-2

	S	A	B	count
a1	369.43	-9999.00	-9999.00	1
a2	331.94	-9999.00	-9999.00	2
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	500.00	-9999.00	0
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	368.27	1

ep2-I

	S	A	B	count
a1	369.43	-9999.00	-9999.00	1
a2	331.94	-9999.00	-9999.00	2
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	500.00	-9999.00	0
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	301.35	2

ep2-II

	S	A	B	count
a1	368.93	-9999.00	-9999.00	2
a2	331.94	-9999.00	-9999.00	2
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	500.00	-9999.00	0
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	301.35	2

ep2-III

	S	A	B	count
a1	368.93	-9999.00	-9999.00	2
a2	331.94	-9999.00	-9999.00	2
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	272.22	-9999.00	1
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	301.35	2

ep2-IV

Episode-3

	S	A	B	count
a1	368.76	-9999.00	-9999.00	3
a2	331.94	-9999.00	-9999.00	2
a5	-9999.00	408.18	-9999.00	1
a4	-9999.00	272.22	-9999.00	1
a6	-9999.00	500.00	-9999.00	0
a3	-9999.00	-9999.00	301.35	2

ep3-I

	S	A	B	C	count
a1	368.76	-9999.00	-9999.00	-9999	3
a2	331.94	-9999.00	-9999.00	-9999	2
a5	-9999.00	408.18	-9999.00	-9999	1
a4	-9999.00	272.22	-9999.00	-9999	1
a6	-9999.00	-9944.54	-9999.00	-9999	1
a3	-9999.00	-9999.00	301.35	-9999	2

ep3-II

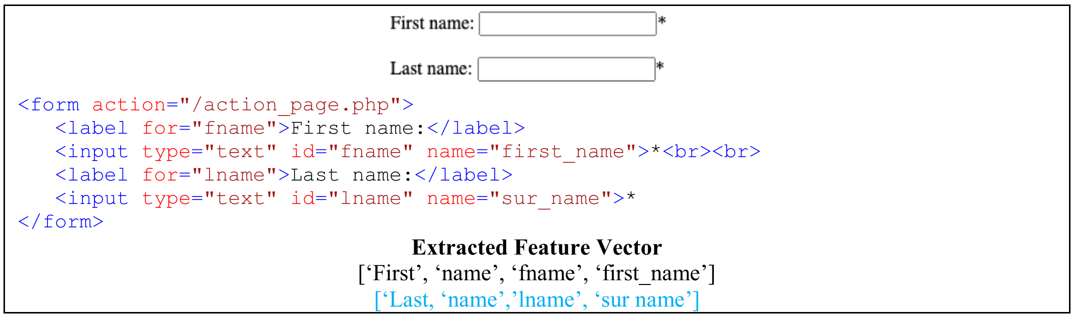
Fig. 4. An example of Q-value update and rewards for three consecutive episodes.

be explored with more diversity. This can be seen from ep2-III to ep3-II where a1 will have priority on a2 in state S and a5 will have priority on a3 and a4 in state A. If there was no future reward, we would have recorded a similar update in the Q-value of action a1 (in ep1-III) in comparison to action a2 (in ep1-III). However, after visiting the state B via action a2 once, the algorithm should prioritize a1 over a2 because state A have more actionable elements than state B. This prioritization between the actions can be achieved by future reward as shown in ep2-III.

In the Q-value function, the balance between an immediate reward and a future reward is maintained by the count value of each action. The lower the count, the more impact the immediate reward will have on the Q-value update and vice versa. One specific condition in the Q-value function is a state that has zero actionable elements, which no more leads to new states, causing a huge penalty to that specific action as shown in ep3-II. The rationale is to visit such states only once.

3.5. Executor

The executor interacts with the DOM elements concretely to perform an action. The executor is comprised of three main components: (1) Action Reverse Engineer, which converts the string representation of the DOM element into a selenium object to locate and makes it accessible for SeleniumDriver; (2) Step, which identifies the type of an action whether it requires the



```
<form action="/action_page.php">
   <label for="fname">First name:</label>
   <input type="text" id="fname" name="first_name">*<br><br>
   <label for="lname">Last name:</label>
   <input type="text" id="lname" name="sur_name">*
</form>
```

Extracted Feature Vector
[‘First’, ‘name’, ‘fname’, ‘first_name’]
[‘Last, ‘name’,’lname’, ‘sur name’]

Fig. 5. An example of DOM element and its feature vector.

Table 1
The type of interactive DOM elements supported by QExplore.

DOM element	Attribute	Description
<Input>	Type=text	Input field for the text
	Type=password	Input field for the password
	Type=checkbox	Input field for the checkboxes
	Type=radio	Input field for the radio buttons
	Type=search	Input field for the search
	Type=url	Input field for url
	Type=number	Input field for any number
	Type=tel	Input field for phone number
	Type=submit	Input field for submit button
	Type=button	Input field for the button
<button>		Tag for a clickable button
<select>		Input field for the drop-down list
<textarea>		Input field for multi-lines text
<a>		Anchor tag for hyperlink

input data or not (e.g. *text*, *password*, *textarea* and *email* etc.); (3) Perform Action, which uses the SeleniumDriver to locate and execute the action on the screen.

We generate dynamic XPaths to locate the interactable elements on a webpage from the information gathered during parsing the DOM tree for interactable elements, as shown in Table 1. Currently, QExplore supports the given interactable DOM elements in the table. But they can be extended to other elements in the future such as identifying drag and drop elements.

3.6. Contextual input data method

Input data plays a significant role in the exploration of the dynamic state-space in web applications (Van Deursen et al., 2015; Sherin et al., 2021). The input data needs to be as realistic as possible to mimic the real usage of the web application under exploration. The DOM attributes such as *name, id, class, value* and *placeholder* are not only used to uniquely identify the DOM element but also provide rich information regarding the context of the data required for the target input field. Similarly, the nearby labels of the DOM elements (i.e., input fields) in the web form can also help to identify the context of data required by the input field. For example, Fig. 5 shows two input fields with the nearby labels i.e., first name and last name which can help in identifying the context of those input fields to fill them with the appropriate input data. In order to specify semantically similar data for the input fields during the exploration, we extract the feature vectors from the text in the label tags and the DOM attributes. Each feature in the feature vector is transformed into normalized form by using Natural Language Processing (NLP) techniques. For normalization, we used uni-gram frequencies based probabilistic concatenated words splitter[2] to extract words that make sense from the user perspective (Qi et al., 2019). Moreover, each extracted word is spellchecked and corrected using an already available dictionary.[3] Fig. 5 shows an example of the feature vector corresponding to the DOM elements.

Typically, string-matching rules can be used to map the data and the DOM representation of the input fields. However, string-matching rules are often application-specific and cannot be generalized to other applications. Therefore, we use word embeddings instead of simple string-matching to semantically map the data topics on the corresponding input fields during the exploration. Topics are the labels provided by the user for the identification of the target input field. For example, the topic for the “Last name” in Fig. 5 can be “Last”, “name”, “lname” or “sur name”. The identification of the most similar topic for the DOM representation of the input field determines its mapping with the specific data, as shown in Fig. 6. To measure the similarity between the DOM elements and data topics, we use word2vec based fastText word embeddings (Bojanowski et al., 2017) to extract the semantic vector from the DOM representation and the data topics. FastText[4] is a free open-source library that provides a pre-trained word embedding model that transforms words into semantic vectors.

This method also enables QExplore to use online available data generators to automatically feed the input fields during the exploration. Instead of developing a new data generation tool for the input data (which is not the focus of this paper), we used an existing data generator called Mocker Data Generator,[5] which is an open-source test data generator based on a schema. Mocker Data Generator combines several existing test data generators such as FakerJS, RandExpJS, ChanceJS, CasualJS in one tool. It allows to specify several parameters, for example, the type of data to be generated, its minimum and maximum length and formatting etc to guide the generation of the data for the target input field. The tool supports the generation of data corresponding to several data topics such as username, first name, last name address. Currently, it supports data generation against a large number of data topics. In particular, FakerJS alone supports more than 200 data topics. The complete list of the topics supported by the Mocker Data Generator is available online[5] and the list can be extended to add application-specific or user-specific data. QExplore allows to specify data manually if a user wants to explore a web application against user-specific data (e.g. *username* and *password*). A user can define multiple values against a single data topic. In the case of multiple values, QExplore randomly picks up one value or value pair (e.g. *username* and *password*). This will reduce the limitations (discussed in Sections 2 and 8) of the previous dynamic exploration approaches. An overview of the contextual input data method is illustrated in Fig. 6.

2 https://pypi.org/project/wordninja/

3 https://pypi.org/project/pyenchant/
4 https://fasttext.cc/
5 https://www.npmjs.com/package/mocker-data-generator

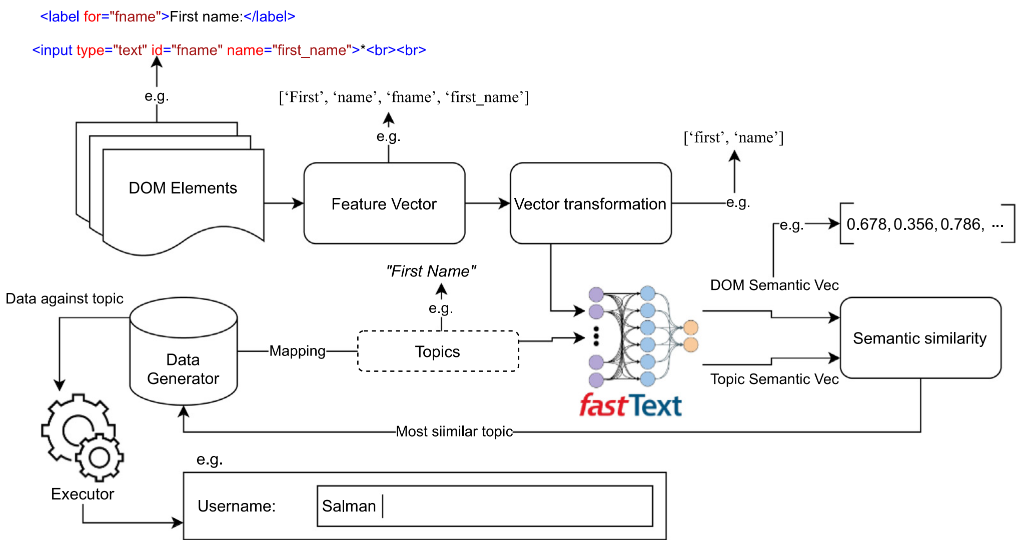


Fig. 6. An overview of the contextual data input method.

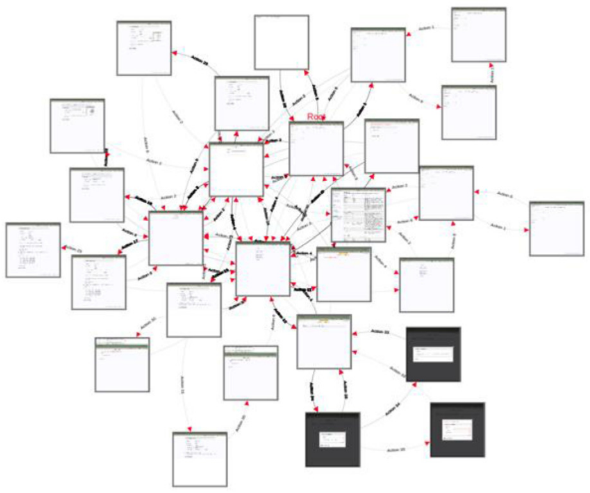


Fig. 7. A resultant State-flow graph by QExplore from Timeclock application.

3.7. State-flow graph generation

QExplore exercises the DOM of the dynamic web application at runtime and identifies interactive elements that can change the DOM state within the browser. We infer the state-flow graph by detecting the DOM changes at runtime capturing the states and transitions (DOM actions) between these states. The changes on the DOM at runtime are caused by either client-side code (AJAX) or server-side script propagated to the client-side. To detect the DOM changes for the identification of states, we used the same technique of *tree edit distance* used in the previous studies (Mesbah et al., 2011; Fard and Mesbah, 2013). We model the different DOM changes as states and DOM actions as transitions between those states for navigation. Therefore, we define a state-flow graph as follows:

State flow graph (SFG): A state flow graph can be represented as a 3-tuple: $\mathbb{G}[\mathbb{r}, \mathbb{s}, \mathbb{E}]$ where $\mathbb{r}$ is the index/root state/node, $\mathbb{s}$ is the set of states/nodes in the graph and $\mathbb{E}$ is the set of edges representing the possible transitions (by performing an action) between the DOM states. Fig. 7 shows an example of SFG derived from timeclock[6] web application after running QExplore for 300 s.

After deriving the SFG, different graph traversal algorithms can be used together with invariants to derive the test cases for the application under test automatically. For example, Mesbah et al. (2011) proposed an invariant-based technique for detecting faults in the DOM tree using invariants on the edges that serve as oracles.

4. Implementation

We have implemented our approach in a prototype tool called QExplore in Python, which is publicly available[7] to other researchers and practitioners. The QExplore comprises five major components, discussed in detail in Section 3, i.e., Environment, QExplore agent, Executor, Contextual Data Input Method, and state flow graph generator. The environment extracts the states and actions from the web application under test during the execution. QExplore agent implements the guided search strategy that explores the dynamic state and action space of the web application. The Executor interacts with the environment concretely to locate the target DOM element by using relative Xpath

6 http://timeclock.sourceforge.net/

7 http://www.github.com/salmansherin/QExplore

Table 2
An overview of the subject applications.

ID	Name	Version	kLOC	Description	Available at
1	Schoolmate	1.5.4	8.181	A management system for elementary to high school's administration	https://sourceforge.net/projects/schoolmate/
2	Addressbook	5.0	19.6	A web-based address and phone book management system	https://sourceforge.net/projects/php-addressbook/
3	Timeclock	1.0.2	20.789	A tool for effective management of time	https://sourceforge.net/projects/timeclock/
4	Collabtive	3.1	73	A web-based project management system	https://sourceforge.net/projects/collabtive/
5	Claroline	1.11.10	306.9	A collaborative e-learning platform	https://sourceforge.net/projects/claroline/
6	Brotherhood	0.4.5	218.8	An online social networking platform	https://github.com/HSJared/Social-Network
7	Flex	1.1	523	Web-based academic management system	https://flexstudent.nu.edu.pk/Login

and SeleniumDriver[8] as an executor. The Contextual Data Input Method implements the mapping between the data and form fields during the execution by using a word2vec based word embeddings called fastText[9] (Bojanowski et al., 2017). The state flow graph generator implements the generation of the state flow graph.

To instantiate, the QExplore requires the URL (e.g. http://www.example.com) of the application under test, crawl depth and the crawl time (in minutes) or the number of episodes. QExplore supports several other parameters to set different crawl configurations. These parameters include the initial Q-value, initial reward value and maximum discount factor.

Similar to existing crawlers such as Crawljax, QExplore also allows a tester to write specific crawl rules for specific user input, but in a more simple way providing ease to the tester. For example, if an application requires a specific user name and password to logged into the system, the tester defines it with a similar keyword as `"username"="Joe"`, `"password"="1234"`. Similarly, a tester can define multiple inputs (e.g. `"user name"= ["Joe", "Smith"]`) for each specific input type such as logging in with a different user name and password multiple times during the crawl session. Moreover, QExplore also allows a tester to save the previous crawl session, which can later be resumed and starts with, thus, enabling incremental crawling.

To detect the equality between two DOM states in the Q-Table, QExplore calculates the hash value from the string representation of the DOM states and then performs a simple string matching.

5. Empirical evaluation

To evaluate the effectiveness of QExplore in supporting the exploration of dynamic web applications, we conduct an experiment with six open-source dynamic web applications and one industrial case study of a web-based academic management system. We compare QExplore with the widely known exploration tool called Crawljax and a recently proposed tool called WebExplor based on well-defined evaluation metrics. The experiment is performed with three different test budgets of 600 s, 1200 s and 1800 s. The performance of QExplore is compared with Crawljax and WebExplor based on diversity, exploration and coverage. The purpose of running all the three tools with different time budget is to analyze their ability in exploring dynamic regions of the web. This is because one of the short comings of the existing exploration techniques (such as Crawljax) is that they get mired in the dynamic regions of web applications. Similarly, analyzing coverage, diversity, navigational paths, distinct states and error states with different time budget adds to measuring the effectiveness of all three tools. Based on this motivation, we have formulated the following two questions to conduct our experiment.

Q1. *How effective QExplore with contextual input data is in the exploration of dynamic web applications as compared to Crawljax and WebExplor with different time constraints?*

In this research question, we evaluate the effectiveness of QExplore with contextual input data based on the evaluation metrics (discussed in Section 5.3) and with different time constraints, i.e., 600, 1200 and 1800 s. Different time constraints will help develop confidence in the results and also assess if QExplore gets mired in dynamic regions of web applications (recall the motivational example).

Q2. *How effective QExplore with random input data is in the exploration of dynamic web applications as compared to Crawljax and WebExplor with different time constraints?*

This RQ measures the effectiveness of QExplore with random data inputs as compared to Crawljax and WebExplor. This will help to identify the effectiveness of our exploration strategy without contextual input data method. This makes the comparison of QExplore with Crawljax and WebExplor in a balanced manner because these tools work with random data rather than contextual data.

5.1. Subject application

Our approach targets dynamic web applications, thus, we selected seven (six open-source and one industrial) subject applications. The selection criteria were (i) subject applications must have client-side (JavaScript) and server-side code (PHP), (ii) must have dynamic DOM manipulations, (iii) must have user inputs and (iv) Does not require domain specific data. All the selected web applications are open-source projects and have been used in several existing studies on web application testing (Sherin et al., 2021; Imtiaz and Iqbal, 2021; Zou et al., 2014; Mirzaaghaei and Mesbah, 2014). Table 2 provides an overview of the subject applications and their properties including version, lines of code, description and online link.

5.2. Experimental setup

We performed our experiment on Ubuntu 18.04.5 LTS and Firefox browser running on 3.2 GHz Intel Core i7-8700 CPU and 32 GB memory. To study the effectiveness, we compare QExplore with the state-of-the-art dynamic exploration approaches such as Crawljax and WebExplor. Though, QExplore is different than other existing exploration approaches (as discussed in the Related

8 https://www.selenium.dev/projects/
9 https://fasttext.cc/

work) the way it explores the state-space of a web application. However, to build the confidence of the practitioners on its practical usage and effectiveness, we compare it with Crawljax and WebExplor. This is because Crawljax is widely known generic tool for exploration of dynamic web applications and several guided exploration tools proposed over the years use Crawljax as a baseline. QExplore is compared with WebExplor because it is closely related to QExplore as it uses reinforcement learning and outperforms all the previously proposed exploration approaches. As the implementation of WebExplor is not publicly available, the algorithm was implemented by our development team. This is a common practice in previous studies when the application code is not available (Imtiaz and Iqbal, 2021; Choudhary et al., 2011; Stocco et al., 2018). To evaluate the implementation of the approach, we replicated the experiment of the original paper using the implemented tool on Dimeshift, Splittypie and Retroboard, which are publicly available open-source case studies. The results of the execution were equivalent to the original experiment. The complete implementation of WebExplor is also made available in an online repository.[10]

For execution of the experiment, the minimum time constraint for exploration (crawl time) is set to 600 s (10 min) for all the approaches because it is an acceptable time for test generation in most testing environments (Humble and Farley, 2010). However, existing studies use different time constraints to empirically evaluate their approaches. For instance, the effectiveness of GUIDE (Liu et al., 2020) is empirically evaluated and compared to Crawljax by setting the exploration time for 150 min (9000 s). WebExplor (Zheng et al., 2021) is compared with Crawljax for an exploration time of 30 min. Similarly, Keyjaxtest (Qi et al., 2019) is empirically evaluated and compared to Crawljax for setting the exploration time up to 15 min (900 s). Furthermore, the effectiveness of FeedEx (Fard and Mesbah, 2013) is evaluated by setting the exploration time for 300 s. But the largest subject application selected for evaluating FeedEx is about 26k lines of code. In our case, the target applications are of different sizes ranging from 8K–523K lines of code having a large state-space which is not feasible to be explored in a minimum time of 300 s. Therefore, we set the minimum time limit of 600 s. Consequently, two different time constraints i.e., 1200 and 1800 s are set to study the effectiveness of our approach as compared Crawljax and WebExplor (Zheng et al., 2021). The experimental results are publicly[11] available for other researchers to replicate our results.

5.3. Evaluation metrics

QExplore supports the exploration of dynamic web applications and derives a test model as an output. To demonstrate the applicability and measure the effectiveness of our approach, we discuss the following evaluation metrics to analyze the efficacy of the test model achieved from our approach. We compare QExplore with the state-of-the-art dynamic exploration approach called Crawljax based on the following metrics proposed in the literature. These metrics are believed to be the properties that a test model should have to cover various aspects of a web application effectively (Fard and Mesbah, 2013; Qi et al., 2019; Benedikt et al., 2002).

5.3.1. Code coverage analysis

Code coverage is an important indicator when it comes to assessing the adequacy of testing. In fact, one of the main objectives for deriving a test model is to achieve sufficient Code coverage. Therefore, we measure the statement coverage of client-side (HTML, JavaScript) and server-side (PHP or ASP.Net) achieved after exploring the application for the given time constraints. To measure the client-side and server-side coverage, we have instrumented the source code by using the methods explained in Sherin et al. (2021) and Zou et al. (2014).

5.3.2. DOM diversity

In dynamic web applications, the DOM changes occur dynamically through JavaScript that reflects a change in the DOM state. These dynamic changes in the DOM are incremental in nature and are small modifications that may not be interesting from an exploration perspective, specifically when given time and space constraints. Therefore, DOM diversity is measured to assess the diversity of the resultant SFG from a dynamic exploration approach. The diversity in DOM states can result in better structural coverage (Fard and Mesbah, 2013). This implies that an SFG with more DOM diversity indicates better structural coverage. This metric was initially proposed by Fard and Mesbah in their approach called FeedEx to measure the efficacy of the test model (Fard and Mesbah, 2013). We adopt this metric in our experiment for evaluation of QExplore in comparison to Crawljax. We used the same method to measure the DOM diversity of the resultant SFG as discussed in the original paper.

5.3.3. Navigational coverage

A dynamic exploration approach aims to discover possible execution paths that a user can follow in a web application. The number of paths explored by the dynamic exploration approach determines the navigational coverage. For adequate navigational coverage, a dynamic exploration approach should capture multiple different execution paths among the DOM states in the SFG. Therefore, it is better for a dynamic exploration approach to explore as many paths as possible in an application. This metric was originally proposed by Fard and Mesbah in Fard and Mesbah (2013) to evaluate the effectiveness of a test model.

5.3.4. Navigational diversity

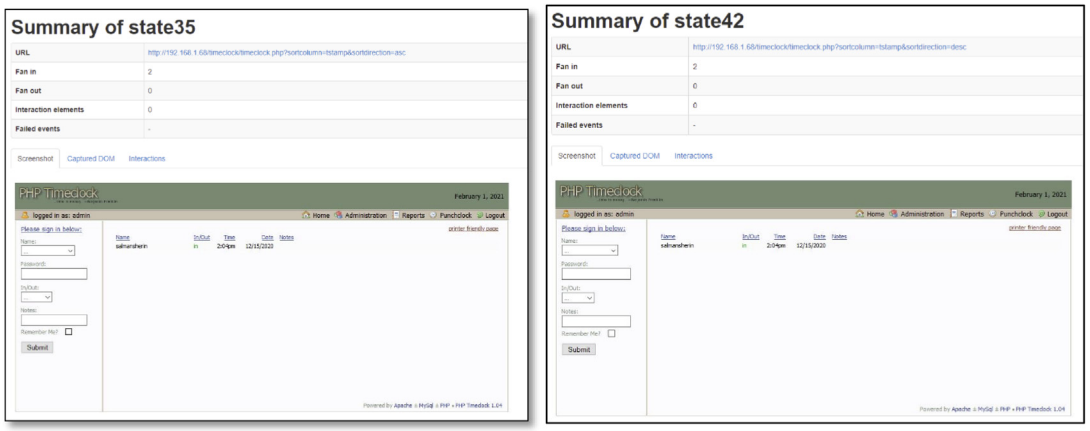
The number of possible execution paths in an SFG indicates the effectiveness of a dynamic exploration approach. But the number of navigational paths alone is not sufficient to assess the effectiveness of a test model. It is also important to analyze the diversity among those explored paths because diverse paths yield better navigational functionality coverage. This metric was proposed in Fard and Mesbah (2013) and is also used by other studies (Qi et al., 2019). Navigational diversity is determined by measuring the average pair-wise navigational diversity of leaf nodes. A detailed discussion on the calculation of average navigational diversity is available in Fard and Mesbah (2013).

5.3.5. Error states

One of the primary goals of exploration is to derive a test model for the application under test that can be used by different testing techniques (e.g. Mesbah et al., 2011) to identify discrepancies between the actual and expected behavior. However, several errors can be detected at the DOM level such as broken links, server-side notices and warnings and HTML conformance violations. These errors also indicate the effectiveness of the dynamic exploration approaches and are used in other studies as well (Benedikt et al., 2002; Groeneveld et al., 2010). Error states can easily be identified by using the returning state code (Belshe et al., 2015).

[10] https://github.com/salmansherin/QExplore/tree/main/Experimental%20results/webExplore

[11] http://www.github.com/salmansherin/QExplore


Summary of state35

URL	http://192.168.1.68/timeclock/timeclock.php?sortcolumn=tstamp&sortdirection=asc
Fan in	2
Fan out	0
Interaction elements	0
Failed events	-

Name	In/Out	Time	Date	Notes
salmansherin	in	2:04pm	12/15/2020	

Summary of state42

URL	http://192.168.1.68/timeclock/timeclock.php?sortcolumn=tstamp&sortdirection=desc
Fan in	2
Fan out	0
Interaction elements	0
Failed events	-

Name	In/Out	Time	Date	Notes
salmansherin	in	2:04pm	12/15/2020	

Fig. 8. An example of similar states.

5.3.6. The number of distinct DOM states

The total number of DOM states alone does not indicate the strength of the dynamic exploration approach due to the number of similar/duplicate DOM states in the resultant SFG which might not be interesting from the testing perspective, particularly when given time and space constraints (Fard and Mesbah, 2013). Since exploring diverse DOM states can result in better structural coverage, we analyzed the number of distinct DOM states captured in an SFG by QExplore and Crawljax. The number of distinct DOM states explored within a given time constraint adds to the effectiveness of the dynamic exploration approach. To measure the number of distinct DOM states, we use the *tree edit distance* (recommended by recent studies Van Deursen et al., 2015; Yandrapally et al., 2020; Sherin et al., 2021) to measure the similarity between the two DOM states. The *tree edit distance* is specified by the minimum cost of operations need to transform one DOM tree into another (Pawlik and Augsten, 2011; Tai, 1979). Therefore, the DOM states in an SFG are distinct when the minimum-cost operations required to transform one DOM tree into another is greater than zero. Fig. 8 shows an example of the two similar DOM states from Timeclock obtained through Crawljax having the required minimum-cost operations equal to zero. The main difference between the two DOM states is the time recorded in the DOM which is not visible to the user. The given DOM states in the figure are not distinct but similar because such a minor difference does not add value when it comes to the testing of the application at a system level.

To calculate the DOM diversity, navigational diversity and distinct DOM, we use *tree edit distance* having time complexity of O(n^2) in best case and O(n^3) in worst case when comparing two DOM states (Augsten, 2016). In our case, we compared every DOM state with all the other DOM states in the tree (obtained via Crawljax and QExplore) having worst time complexity of $O(m^2n^3)$ and best time complexity $O(m^2n^2)$. Where m is the number of DOM states and n^3 is the time complexity for two DOM states. The complexity of the algorithm increases with the increase in duplicate states. On average, we captured 80 DOM states across all the case studies explored by Crawljax, WebExplor and QExplore. The average number of DOM elements in a state is 93. The total number of iterations required for the defined metrics is $((80^2 93^2)/12) = 4{,}612{,}800$ based on the machine having 12 cores used for this experiment. Each iteration roughly takes a second for execution, therefore, this experiment required ∼54 days (4,612,800/(3600*24)) of computational resources. The time increases exponentially with an increase in the number of DOM states. The computational cost for this experiment limits the number of subject applications used in this experiment.

5.4. Experimental results

In this section, we present results and discuss the answers to our research questions.

5.4.1. How effective QExplore with contextual input data is in the exploration of dynamic web applications as compared to Crawljax and WebExplor with different time constraints?

The motivation to answer this question is to compare QExplore with contextual input data (Q^C) with the existing tools, i.e., Crawljax and WebExplor. Though, we understand that other tools do not support contextual data but in practice our tool is going to be used with contextual data. Also, recall the working example of our approach which works in deterministic manner when the environment remains the same. This is not true for Crawljax (C^R) and WebExplor (W^R), which works in non-deterministic manner and uses random data inputs. Therefore, it is not fair to compare the averaged results obtained from deterministic approach (Q^C) with the averaged results obtained from non-deterministic approaches (C^R and W^R). Therefore, to be fair enough, we compare the best run result from 15 runs of C^R and W^R for different time constraints with Q^C, as shown in Table 3. We also present the comparison of Q^C with the worst run results of C^R and W^R in Table 4. Moreover, we assess the performance of QExplore with random data as compare to Crawljax and WebExplor in RQ2.

All three approaches were provided with the login credentials of the applications and were allowed to click on the same DOM element multiple times. The table shows the results for code coverage, DOM diversity, navigational coverage, navigational diversity, error states and the number of distinct states obtained Crawljax, WebExplor and QExplore. The results for each metric are discussed as follows:

Code Coverage: The overall code coverage achieved by all the three approaches within different time budget is shown in the first column of Table 3. It can be seen that QExplore outperforms Crawljax showing an improvement of 6%–21%, 12%–26% and 11%–30% within 600, 1200 and 1800 s of time budget respectively. Similarly, QExplore achieved greater coverage than WebExplor showing an improvement of 6%–29%, 9%–34% and 10%–28% within 600, 1200 and 1800 s of time budget respectively. Interestingly, Crawljax within time budget of 600 s achieved slightly better coverage than WebExplor in five out of seven subject applications. Within the time budget of 1200 s, WebExplor slightly achieved better coverage than Crawljax in three out of seven subject applications. However, within the time budget of 1800 s, WebExplor achieved greater coverage than Crawljax in all the seven subject applications. This indicates that WebExplor needs relatively

Table 3
Best run results from Crawljax, WebExplor in comparison to QExplore with contextual data for 600, 1200 and 1800 s of time.

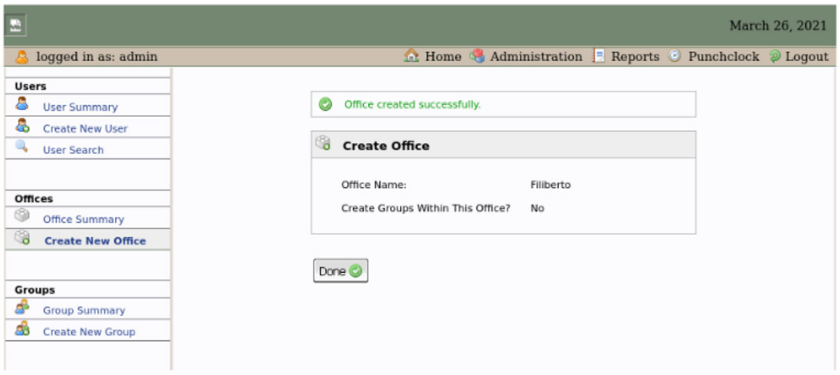
An overview of the results with 600 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	58%	51%	49%	48%	38%	44%	29%
	W^R	50%	50%	41%	43%	41%	39%	28%
	Q^C	79%	60%	67%	54%	47%	57%	39%
DOM Diversity	C^R	0.225	0.546	0.08	0.5	0.548	0.58	0.47
	W^R	0.25	0.60	0.45	0.55	0.541	0.59	0.43
	Q^C	0.26	0.647	0.53	0.62	0.5488	0.62	0.68
Navigational coverage	C^R	58	125	115	37	302	23	166
	W^R	356	98	44	127	198	18	110
	Q^C	580	270	197	272	356	85	207
Navigational diversity	C^R	0.1	0.59	0.25	0.21	0.14	0.25	0.31
	W^R	0.29	0.51	0.52	0.48	0.60	0.57	0.40
	Q^C	0.56	0.82	0.9	0.97	0.86	0.96	0.57
Error states	C^R	4	3	4	3	3	4	0
	W^R	4	2	4	2	5	4	1
	Q^C	7	5	4	3	7	9	2
Total number of distinct states	C^R	53	26	40	9	20	7	62
	W^R	51	29	55	10	22	9	70
	Q^C	62	32	68	10	25	10	88
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	62%	55%	54%	59%	48%	54%	32%
	W^R	54%	58%	45%	49%	50%	47%	34%
	Q^C	88%	67%	74%	73%	66%	70%	55%
DOM Diversity	C^R	0.20	0.33	0.07	0.41	0.456	0.45	0.33
	W^R	0.20	0.49	0.445	0.52	0.486	0.50	0.36
	Q^C	0.25	0.57	0.49	0.59	0.52	0.58	0.63
Navigational coverage	C^R	88	474	869	54	395	49	211
	W^R	423	389	1679	254	883	176	276
	Q^C	803	642	2295	382	1075	212	398
Navigational diversity	C^R	0.07	0.31	0.12	0.19	0.08	0.18	0.22
	W^R	0.27	0.48	0.44	0.39	0.53	0.48	0.32
	Q^C	0.53	0.80	0.84	0.92	0.76	0.88	0.51
Error states	C^R	4	3	4	3	4	6	0
	W^R	5	5	6	4	6	7	2
	Q^C	9	7	10	6	8	11	5
Total number of distinct states	C^R	59	29	46	27	34	9	57
	W^R	60	35	78	27	36	21	83
	Q^C	92	39	89	30	36	29	95
An overview of the results with 1800 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	63%	58%	60%	61%	55%	56%	33%
	W^R	65%	59%	62%	63%	60%	55%	35%
	Q^C	89%	69%	79%	85%	78%	77%	63%
DOM Diversity	C^R	0.170	0.31	0.06	0.34	0.256	0.40	0.31
	W^R	0.19	0.50	0.38	0.48	0.39	0.422	0.34
	Q^C	0.24	0.57	0.47	0.57	0.47	0.51	0.62
Navigational coverage	C^R	111	545	1069	167	465	69	217
	W^R	764	671	1856	356	656	186	277
	Q^C	1103	878	2655	582	1375	252	499
Navigational diversity	C^R	0.07	0.31	0.12	0.19	0.07	0.18	0.16
	W^R	0.17	0.34	0.36	0.331	0.47	0.38	0.276
	Q^C	0.53	0.80	0.84	0.92	0.74	0.88	0.42
Error states	C^R	4	4	5	3	4	6	0
	W^R	6	8	12	6	5	6	3
	Q^C	9	12	14	9	8	12	5
Total number of distinct states	C^R	65	29	50	27	36	9	69
	W^R	71	43	85	36	43	29	101
	Q^C	111	54	97	39	46	36	195

more time budget to explore web applications as compared to Crawljax and QExplore. In contrast, QExplore with contextual input data consistently improves the coverage in all the subject applications for three consecutive time budgets which indicates that QExplore efficiently uses the time budget to explore the web application by achieving higher coverage than Crawljax and WebExplor. It is noticeable that none of the approaches achieved a reasonable code coverage (80%, according to Sherin et al., 2021) in any of the subject applications within 600s of exploration time. Overall, QExplore achieved better code coverage in all the three

Table 4
Worst run results from Crawljax, WebExplor in comparison to QExplore with contextual data for 600, 1200 and 1800 s of time.

An overview of the results with 600 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	35%	25%	24%	21%	20%	18%	14%
	W^R	29%	25%	26%	21%	20%	21%	14%
	Q^C	79%	60%	67%	54%	47%	57%	39%
DOM Diversity	C^R	0.09	0.35	0.03	0.36	0.39	0.41	0.32
	W^R	0.1	0.41	0.27	0.36	0.39	0.41	0.3
	Q^C	0.26	0.647	0.53	0.62	0.5488	0.62	0.68
Navigational coverage	C^R	50	87	65	32	209	14	100
	W^R	230	66	40	41	54	14	66
	Q^C	580	270	197	272	356	85	207
Navigational diversity	C^R	0.01	0.44	0.15	0.1	0.12	0.18	0.12
	W^R	0.12	0.4	0.35	0.36	0.39	0.36	0.27
	Q^C	0.56	0.82	0.9	0.97	0.86	0.96	0.57
Error states	C^R	1	2	1	1	1	2	0
	W^R	2	2	2	2	1	2	1
	Q^C	7	5	4	3	7	9	2
Total number of distinct states	C^R	32	13	20	4	11	3	32
	W^R	35	15	36	5	11	5	35
	Q^C	62	32	68	10	25	10	88
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	30%	28%	27%	34%	25%	24%	16%
	W^R	27%	28%	22%	27%	25%	22%	17%
	Q^C	88%	67%	74%	73%	66%	70%	55%
DOM Diversity	C^R	0.06	0.2	0.03	0.3	0.36	0.35	0.27
	W^R	0.07	0.32	0.22	0.34	0.34	0.39	0.29
	Q^C	0.25	0.57	0.49	0.59	0.52	0.58	0.63
Navigational coverage	C^R	72	209	14	34	72	35	134
	W^R	332	102	703	124	401	67	143
	Q^C	803	642	2295	382	1075	212	398
Navigational diversity	C^R	0.03	0.23	0.06	0.09	0.04	0.07	0.15
	W^R	0.1	0.39	0.34	0.24	0.35	0.31	0.23
	Q^C	0.53	0.80	0.84	0.92	0.76	0.88	0.51
Error states	C^R	2	2	1	1	2	2	0
	W^R	3	2	2	2	2	4	2
	Q^C	9	7	10	6	8	11	5
Total number of distinct states	C^R	29	15	23	16	18	4	29
	W^R	40	17	39	15	18	10	42
	Q^C	92	39	89	30	36	29	95
An overview of the results with 1800 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	C^R	31%	28%	30%	29%	38%	24%	16%
	W^R	32%	28%	30%	31%	29%	26%	19%
	Q^C	89%	69%	79%	85%	78%	77%	63%
DOM Diversity	C^R	0.09	0.23	0.04	0.24	0.15	0.34	0.25
	W^R	0.1	0.37	0.2	0.28	0.32	0.32	0.25
	Q^C	0.24	0.57	0.47	0.57	0.47	0.51	0.62
Navigational coverage	C^R	104	321	865	167	87	322	54
	W^R	432	388	998	145	331	112	124
	Q^C	1103	878	2655	582	1375	252	499
Navigational diversity	C^R	0.02	0.21	0.05	0.1	0.01	0.05	0.04
	W^R	0.07	0.28	0.28	0.23	0.28	0.28	0.16
	Q^C	0.53	0.80	0.84	0.92	0.74	0.88	0.42
Error states	C^R	1	1	2	1	1	2	0
	W^R	3	3	6	3	3	4	1
	Q^C	9	12	14	9	8	12	5
Total number of distinct states	C^R	33	14	25	13	25	4	34
	W^R	49	21	42	18	21	14	56
	Q^C	111	54	97	39	46	36	195

different time budgets. The maximum code coverage is achieved on Schoolmate application whereas the minimum code coverage is achieved on our industrial application (Flex). This shows that 600 s of exploration time is not enough to explore the state-space of real dynamic web application when it comes to achieving a reasonable code coverage. Crawljax achieved 33% coverage and WebExplor achieved 35% coverage within (1800 s) on Flex. One of the main reasons behind achieving the lower coverage is the

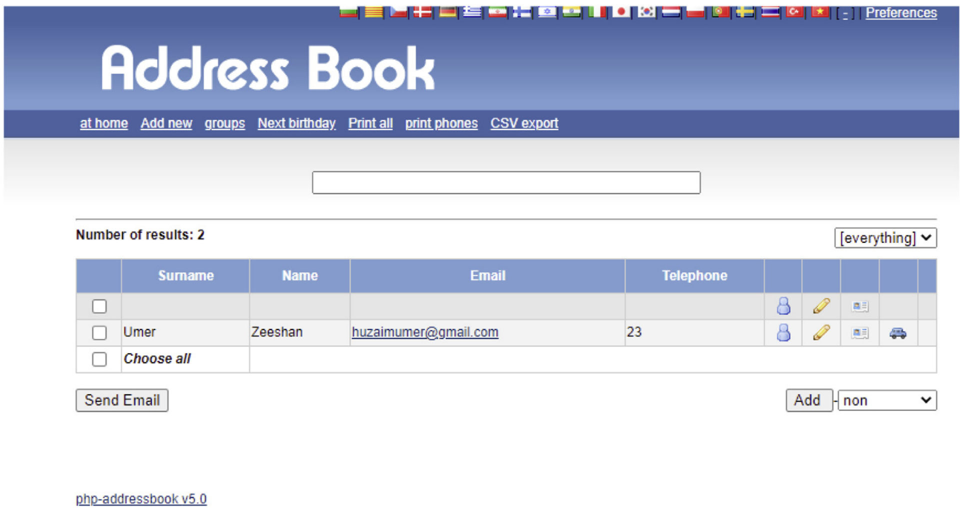


Office created successfully.

Create Office

Office Name:	Filiberto
Create Groups Within This Office?	No

Fig. 9. QExplore successfully adding a new office in Timeclock.



Number of results: 2

	Surname	Name	Email	Telephone
☐				
☐	Umer	Zeeshan	huzaimumer@gmail.com	23
☐	*Choose all*			

Fig. 10. An overview of Addressbook.

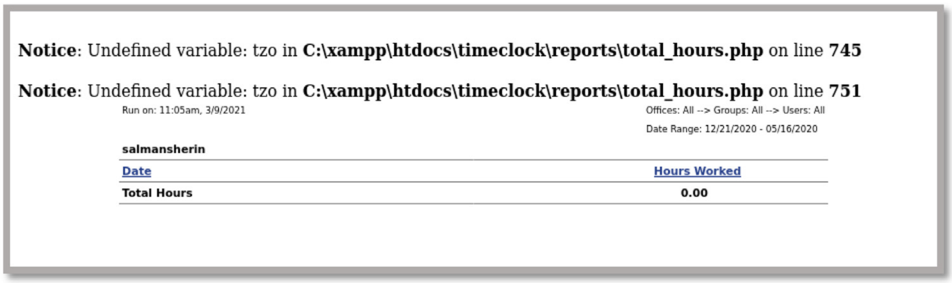
use of random input by both Crawljax and WebExplor. There are several web forms in the application (such as adding new semester, course and student) that has validation constraints and cannot be reached with the random data. However, QExplore reached those states behind forms due to the contextual data method achieving better code coverage (63% within 1800 s) than the Crawljax and WebExplor. This shows that our contextual data input method plays a key role in increasing the exploration of DOM states behind the web forms which helps in achieving higher coverage than other exploration approaches.

Though, QExplore is effective in achieving higher coverage. Still, only coverage does not provide sufficient evidence when it comes to the exploration of dynamic web applications because a single server-side script can generate several DOM states making it important for the exploration approaches to capture diverse DOM states in the state-flow graph. This brought us to measuring another important aspect of the test model generated by the exploration approaches called diversity.

Finding 1. Overall, QExplore achieved greater code coverage than existing state-of-the-art approaches such as Crawljax and WebExplor.

DOM Diversity: The DOM diversity of all the three approaches is analyzed by calculating the average diversity of all the states captured by each approach in its EFG or test model. The key idea to capture diverse DOM states of the application instead of similar ones because similar DOM states having same business logic may not be interesting from testing perspective. Table 3 shows the average diversity obtained by each exploration approach in its test model (EFG). It is clearly evident that the test model obtained through QExplore comprises more diverse DOM states than Crawljax and WebExplor. Crawljax performed worse to explore diverse DOM states because it captures similar or duplicate DOM states in the resultant test model which minimizes the average DOM. As compared to Crawljax, WebExplor performed better in capturing diversity in its test model. In fact, the DOM diversity difference between QExplore and WebExplor is marginal in five subject applications. However, WebExplor uses random or manual data input during the exploration and failed to discover states behind the web forms. Therefore, in applications such as Claroline and Flex where more input data is required from the user, QExplore performed exceptionally well. For example, QExplore was able to capture a DOM state, shown in Fig. 9, from Timeclock that Crawljax and WebExplor failed to capture. The figure shows a DOM state captured by QExplore after adding a new office in Timeclock which has several validation constraints that other existing exploration approaches (Crawljax and WebExplor) failed to satisfy and reach states behind those forms.

The lowest average diversity in 600 s, 1200 s and 1800 s is recorded in Addressbook from Crawljax i.e., 0.08, 0.07 and 0.06 respectively. This is because most of the DOM states in the resultant SFG of Crawljax are duplicates/similar. Addressbook is a multilingual web application that allows users to switch the application to 21 different languages, as shown in Fig. 10. For QExplore, the DOM actions in the DOM state are the same despite changing the language of that DOM state. Because the *tree*



```
Notice: Undefined variable: tzo in C:\xampp\htdocs\timeclock\reports\total_hours.php on line 745

Notice: Undefined variable: tzo in C:\xampp\htdocs\timeclock\reports\total_hours.php on line 751
```

Run on: 11:05am, 3/9/2021

Offices: All --> Groups: All --> Users: All

Date Range: 12/21/2020 - 05/16/2020

salmansherin

Date	Hours Worked
Total Hours	**0.00**

Fig. 11. An example of error DOM state from Timeclock.

edit distance, in QExplore, takes the structure of the DOM into consideration rather than its content. However, Crawljax uses Levenshtein distance which takes the content of the DOM state into consideration, which results in duplicate/similar DOM states and lower DOM diversity. Similarly, the state representation in WebExplor is based on the structure of HTML page whereas QExplore represents each state in terms of a set of interactable elements. The problem with structural representation of the state in dynamic web applications is that different business logic can have the same structural representation restricting the exploration strategy to consider DOM states with different business logic as one state.

Finding 2. QExplore is effective in capturing diverse DOM states followed by state-of-the-art WebExplor and Crawljax.

Navigational Coverage and Diversity: The results in Table 3 show the navigational paths discovered by the exploration approaches and its average diversity. The results indicate that QExplore explored more number of paths than WebExplor and Crawljax. Between the WebExplor and Crawljax, the former explored more number of executable paths than the later one. QExplore discovered more navigational paths with greater diversity than Crawljax and WebExplor to reach the different DOM states in an application. As compared to Crawljax, QExplore achieved an improvement of 41–1586 navigational paths in three consecutive runs in all the subject applications. Similarly, when compared to WebExplor, QExplore achieved an improvement of 36–799 in navigational coverage. The improvement in the navigational coverage along with the average diversity of the navigational path shows the effectiveness of QExplore in exploring the state-space of dynamic web applications.

One of the main reasons behind the greater navigational diversity achieved by QExplore is that it systematically explores the web application by clicking on different interactive elements and visiting different DOM states based on the feedback (reward) and considering the potential of DOM states that can lead to new DOM states and navigational paths. In contrast, Crawljax uses DFS or BFS having no feedback to guide the crawl process when it is allowed to perform the same action multiple times resulting in similar DOM states and navigational paths. Similarly, the reward function of WebExplor does not consider the potential of a DOM state that can open-up several new DOM states and executable paths. Hence, WebExplor assigns the same weight to two DOM states having different number of interactable elements restricting the exploration of DOM states having more potential of new DOM states and navigational paths. In contrast, QExplore consider the potential of DOM states that can leads to several new DOM states and diverse paths which increases the navigational coverage and diversity within the defined time budget. Another reason for achieving diversity and greater navigational paths by QExplore in all the subject applications is the contextual data input method providing contextually correct data that help in exploring states and paths beyond the web forms.

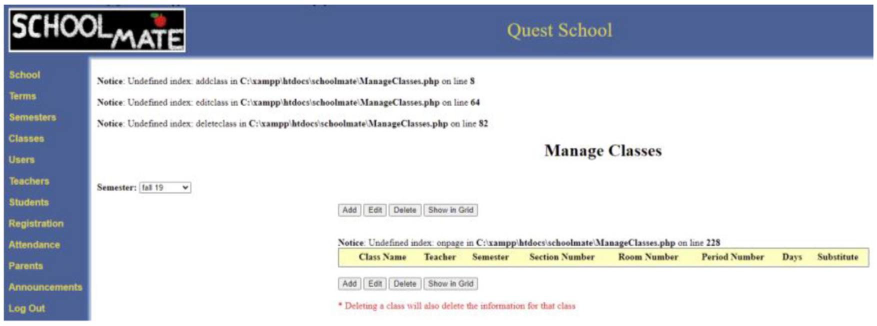
Finding 3. QExplore achieved more navigational paths with greater diversity followed by WebExplor and Crawljax.

Distinct states and Error States: As discussed in Section 5.3.6, the number of DOM states in the SFG alone does not indicate the strength of the dynamic exploration approach because it contains duplicate/similar states which may not be interesting from a tester's perspective. Similarly, the number of error states captured in the EFG also indicates the effectiveness of an exploration approach. It is interesting to note that QExplore captured more distinct DOM states in 600 s of exploration time than Crawljax in all our subject applications. This indicates that QExplore is not only effective in achieving greater coverage and DOM diversity but is also effective in exploring distinct DOM states and error states as compared to state-of-the-art Crawljax and WebExplor.

It can be seen from the results that overall QExplore discovered more error states than Crawljax and WebExplor in all the subject applications. In total QExplore discovered 38, 46 and 69 error states in 600 s, 1200 s and 1800 s respectively. Consequently, WebExplor discovered 22, 35 and 46 error states in 600 s, 1200 s and 1800 s respectively whereas Crawljax discovered 21, 24 and 26 error states in 600 s, 1200 s and 1800 s respectively. Figs. 11 and 12 show examples of error states captured by QExplore. On further investigation, we found that 30 out of 153 total error states detected by QExplore in all the three time budgets are due to the contextual data input method used by QExplore. Overall, QExplore reported a greater number of error states and distinct states followed by WebExplor and Crawljax.

Finding 4. QExplore captured more distinct DOM states and error states than Crawljax and WebExplor within crawl time of 600 s, 1200 s and 1800 s

In all three rounds (600 s, 1200 s,1800 s), QExplore shows improvement in the results revealing its applicability and effectiveness compared to the existing state-of-the-art approaches such as WebExplor and Crawljax. We performed an in-depth analysis to figure out why QExplore achieved better results than Crawljax and WebExplor in all the subject applications. We find that QExplore uses a contextual data input method which helps in exploring DOM states behind the web forms such as shown in Fig. 13. Crawljax and WebExplor use random input data which restricts the exploration of states behind the forms resulting in poor coverage and inadequate EFG. QExplore also supports multiple values for a single input field which further improves the coverage and exploration of different states behind the same web forms. For instance, Flex has multiple user roles showing different interfaces to different users such as students and teachers. A student can see his grade and performance in every course whereas a teacher can manage and organize the results of students. QExplore successfully explored the system by logging in with different user credentials provided to it. However, Crawljax and WebExplor are unable to explore the different aspects of the



Quest School

Notice: Undefined index: addclass in **C:\xampp\htdocs\schoolmate\ManageClasses.php** on line **8**

Notice: Undefined index: editclass in **C:\xampp\htdocs\schoolmate\ManageClasses.php** on line **64**

Notice: Undefined index: deleteclass in **C:\xampp\htdocs\schoolmate\ManageClasses.php** on line **82**

Manage Classes

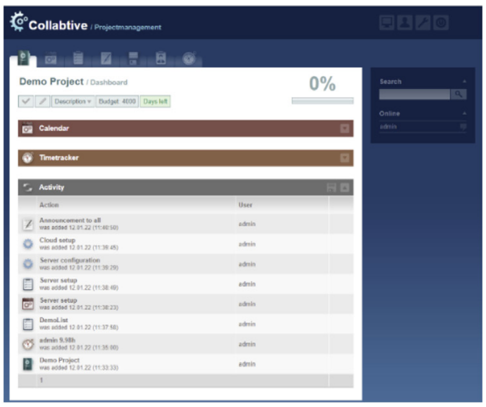
Semester: fall 19

Notice: Undefined index: onpage in **C:\xampp\htdocs\schoolmate\ManageClasses.php** on line **228**

Class Name	Teacher	Semester	Section Number	Room Number	Period Number	Days	Substitute

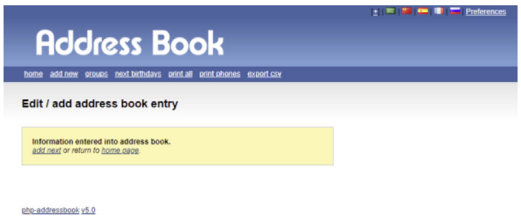
* Deleting a class will also delete the information for that class

Fig. 12. An example error DOM state from Schoolmate.



Demo Project / Dashboard

Action	User
Announcement to all was added 12.01.22 (11:40:50)	admin
Cloud setup was added 12.01.22 (11:39:45)	admin
Server configuration was added 12.01.22 (11:39:29)	admin
Server setup was added 12.01.22 (11:38:49)	admin
Server setup was added 12.01.22 (11:38:23)	admin
DemoList was added 12.01.22 (11:37:58)	admin
admin 9.98h was added 12.01.22 (11:35:00)	admin
Demo Project was added 12.01.22 (11:33:33)	admin
1	



Edit / add address book entry

Information entered into address book.
add next or return to home page.

php-addressbook v5.0

Fig. 13. Example of states behind web forms.

system in the same crawling session resulting in missing DOM states in the resultant SFG.

In the case of Crawljax, it is interesting to note that there is no substantial improvement in code coverage, DOM diversity, navigational coverage, error states and the number of distinct states when the crawl time is increased from 1200 s to 1800 s. This is because Crawljax gets stuck in the dynamic regions of subject applications failing to improve code coverage and add more states to the resultant SFG. Crawljax uses DFS or BFS having no feedback to guide the crawl process when it is allowed to perform the same action multiple times resulting in similar DOM states and navigational paths. It is also noticeable that WebExplor performs better than Crawljax keeping in mind some of the exceptions in coverage and diversity within 600 s but it failed to outperform QExplore in any of the defined metrics. Besides using random data, the state representation in WebExplor is based on the structure of the HTML page whereas QExplore represents each state in terms of a set of interactable elements. The problem with structural representation of the state in dynamic web applications is that different business logic can have the same structural representation restricting the exploration strategy to consider DOM states with different business logic as one state (as seen in Addressbook). Moreover, the reward function of WebExplor does not consider the potential of a DOM state that can open up several new DOM states. This implies that WebExplor assigns the same weight to two DOM states having different number of interactable elements. However, a DOM state with greater number of interactable elements has the potential to lead to new DOM states more than the one with fewer number of interactable elements. In contrast, QExplore considers the potential of a DOM state in terms of interactable elements in its reward function, thus, complementing the exploration process to capture new and diverse DOM states.

QExplore is incremental in nature and allows testers to resume the crawl process from the previous state. Since Crawljax and WebExplor are regressive in nature and do not support incremental crawling, therefore, we gathered the results for QExplore by running it independently for all the three rounds like Crawljax and WebExplor. However, to assess the incremental nature of the QExplore, we have also gathered results running incrementally for the subsequent runs (1200 s and 1800). As predicted, the results gathered incrementally are similar to the results gathered by running QExplore incrementally. There is a negligible difference between the results gathered by running QExplore both ways, i.e., independently for three runs (600 s, 1200 s and 1800 s) and incrementally for the two subsequent runs (1200 s and 1800 s). The results are available online[12] and interesting readers can see them for further details.

To further verify the impact of contextual data input method in the approach, we executed QExplore with random data input on the same subject applications for the same crawl times. We compared the results of QExplore with contextual data input method and QExplore with random data input to see the impact of the contextual data input method in exploring web applications. A detailed discussion on the results is provided to the answer in RQ2.

Since Table 3 shows the results from Q^C with best run results of Crawljax and WebExplor that we discussed above in detail. For interesting readers, we also present the results of Q^C with the worse run results Crawljax and WebExplor in Table 4. It is clearly evident that Q^C shows a sufficient improvements in all our defined metrics as compare to the existing tools.

5.4.2. How effective QExplore with random input data is in the exploration of dynamic web applications as compared to Crawljax and WebExplor with different time constraints?

In this RQ, we compare QExplore with random data (Q^R) against the Crawljax (C^R) and WebExplor (W^R). The motivation

12 https://github.com/salmansherin/QExplore/

Table 5
Averaged results for 15 runs with random data for 600, 1200 and 1800 s of time for Crawljax.

An overview of the results with 600 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	46.3	34.65	36.50	32.712	30.27	32.68	21.64
	M	45.96	33.35	35.52	26.67	30.4	37.71	22.45
	Min	35.02	25.5	24.5	21.33	20.9	18.86	14.97
	SD	7.59	8.73	8.11	9.84	5.8	9.26	4.06
	Skp	0.076	0.75	0.19	0.36	−0.35	0.065	0.205
	$\beta2$	−1.011	−0.595	−1.068	−1.403	−1.086	−1.247	−1.009
DOM diversity	μ	0.18	0.44	0.18	0.42	0.49	0.50	0.39
	M	0.25	0.48	0.16	0.45	0.54	0.59	0.44
	Min	0.09	0.35	0.09	0.36	0.39	0.41	0.32
	SD	0.077	0.077	0.082	0.074	0.098	0.100	0.089
	Skp	−0.285	−0.040	1.039	0.185	−0.139	−0.441	0.114
	$\beta2$	−0.502	−1.201	0.403	−1.062	−1.52	−1.340	−1.478
Navigational coverage	μ	51.2	88.8	66.3	33.4	210.3	15.7	101.4
	M	51.07	88.65	66.7	33.25	210.37	15.6	101.42
	Min	50.28	87.83	65.44	32.47	209.58	14.74	100.6
	SD	0.573	0.658	0.607	0.596	0.484	0.682	0.497
	Skp	0.346	0.075	−0.336	−0.459	0.111	−0.058	−0.120
	$\beta2$	−1.012	−1.475	−1.319	−1.185	−0.687	−1.463	−0.893
Navigational diversity	μ	0.09	.53	00.22	0.20	0.13	0.21	0.28
	M	0.08	0.56	0.24	0.18	0.17	0.23	0.28
	Min	0.01	0.44	0.15	0.1	0.05	0.12	0.18
	SD	0.0535	0.058	0.056	0.056	0.056	0.049	0.068
	Skp	0.160	−0.396	−0.022	0.278	−0.542	−0.189	−0.068
	$\beta2$	−1.241	−1.297	−1.248	−0.825	−0.996	−0.287	−1.453
Error states	μ	3.1	2.3	3.2	2.3	2.0	3.3	0
	M	2	1	1	2	2	2	1
	Min	0	0	1	0	0	1	0
	SD	1.21	1.32	1.41	1.22	1.11	2.121	1.33
	Skp	0.351	−0.138	0.271	0.346	0.231	0.213	−0.113
	$\beta2$	0.231	0.214	0.413	−0.154	0.165	0.124	−0.133
Total number of distinct states	μ	42.4	17.66	29.8	6.1	15.93	5.2	46.26
	M	42	17	29	5	16	6	48
	Min	32	13	20	4	11	3	32
	SD	6.936	4045	6.624	1.846	3.081	1.473	8.688
	Skp	0.075	0.752	0.196	0.362	−0.223	−0.354	0.065
	$\beta2$	−1.011	−0.595	−1.068	−1.403	−1.086	−1.247	−1.009
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	43.85	41.09	39.13	48.21	37.74	41.6	23.54
	M	43.08	43.62	37.57	50.26	39.53	48	21.33
	Min	30.47	28.45	27	34.96	25.41	24	16.28
	SD	8.87	10.47	9.22	9.21	8.57	11.88	5.65
	Skp	0.067	0.21	−0.166	−0.155	−0.24	−0.12	−0.34
	$\beta2$	−0.561	−1.681	−1.297	−1.524	−1.406	−1.575	−1.563
DOM diversity	μ	0.14	0.27	0.13	0.40	0.45	0.43	0.34
	M	0.18	0.34	0.19	0.44	0.49	0.47	0.45
	Min	0.06	0.2	0.03	0.3	0.36	0.35	0.27
	SD	0.086	0.074	0.088	0.081	0.087	0.084	0.074
	Skp	0.195	−0.329	−0.414	−0.117	0.222	0.170	−0.543
	$\beta2$	−1.257	−0.481	−1.117	−1.171	−0.865	−1.04	−0.731
Navigational coverage	μ	73.4	255.3	645.4	42.3	255.3	33.2	135.6
	M	73.35	254.83	645.07	42.2	254.81	33.08	135.34
	Min	72.45	254.33	644.44	41.31	254.32	32.27	134.7
	SD	0.602	0.623	0.505	0.518	0.499	0.581	0.659
	Skp	0.045	0.356	0.067	0.031	1.141	0.192	0.224
	$\beta2$	−0.932	−1.484	−1.191	−0.408	0.833	−1.388	−1.479
Navigational diversity	μ	0.07	0.33	0.15	0.18	0.13	0.17	0.21
	M	0.08	0.36	0.17	0.17	0.1	0.18	0.24
	Min	−0.02	0.23	0.06	0.09	0.04	0.07	0.15
	SD	0.059	0.061	0.066	0.062	0.070	0.061	0.040
	Skp	0.011	−0.548	−0.189	0.093	0.031	−0.122	−0.536
	$\beta2$	−1.083	−0.793	−1.272	−1.239	−1.534	−1.205	−0.535

(*continued on next page*)

behind this RQ is twofold; (1) analyzing the effectiveness of exploration strategy by using random data inputs rather than contextual data and (2) measuring the effectiveness of all the tools in the same manner because Crawljax and WebExplor support random data. Based on the motivation, we average the results for all the 15 runs and followed the guidelines in Fraser and Arcuri (2012) to present the Mean (μ), Median (M), Minimum (Min), Standard Deviation (SD), Skewness (Skp) and Kurtosis ($\beta2$) of the obtained results for Crawljax, WebExplor and QExplore, as shown in Table 5, Table 6 and Table 7 *respectively*. Secondly, we

Table 5 (*continued*).

Error states	μ	3.5	2.4	3.6	2.5	3.6	5.1	0.2
	M	2	1	2	1	1	3	1
	Min	0	1	2	2	1	1	0
	SD	2.42	1.31	1.62	1.22	1.54	3.89	1.27
	Skp	0.341	−0.243	0.344	−0.164	0.412	0.364	0.161
	$\beta2$	0.181	−0.162	0.134	−0.166	0.154	−0.134	−0.122
Total number of distinct states	μ	41.73	21.66	33.33	22.06	26.73	6.93	41.93
	M	41	23	32	23	28	8	38
	Min	29	15	23	16	18	4	29
	SD	8.447	5.524	7.86	4.216	6.076	1.980	10.067
	Skp	0.205	0.06	0.215	−0.166	−0.155	−0.247	0.120
	$\beta2$	−0.561	−1.681	−1.297	−1.524	−1.406	−1.575	−1.563
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	48.39	40.26	47.92	46.08	45.83	38.99	26.43
	M	53.31	36	51	47.44	47.36	37.33	30.13
	Min	31.98	28	30	29.37	38.19	24.89	16.26
	SD	9.43	9.49	9.80	11.73	5.83	10.89	6.24
	Skp	0.88	−0.48	0.007	−0.0405	−0.091	−0.48	−0.28
	$\beta2$	−1.076	−0.567	−1.219	−1.650	−1.105	−1.367	−1.401
DOM diversity	μ	0.17	0.30	0.11	0.33	0.24	0.40	0.30
	M	0.23	0.36	0.11	0.42	0.31	0.52	0.33
	Min	0.09	0.23	0.04	0.24	0.15	0.34	0.25
	SD	0.082	0.080	0.08	0.094	0.090	0.093	0.061
	Skp	−0.143	0.005	0.365	−0.381	−0.073	−0.622	0.24
	$\beta2$	−0.981	−1.342	−1.280	−1.066	−1.148	−0.998	−1.049
Navigational coverage	μ	105.6	322.3	866.3	88.3	323.2	55.6	155.38
	M	105.25	322.19	866.46	88.21	322.99	55.14	155.08
	Min	104.67	321.35	865.39	87.35	322.26	54.61	154.44
	SD	0.701	0.582	0.576	0.580	0.657	0.574	0.471
	Skp	0.249	0.018	−0.288	0.118	0.323	0.781	−0.036
	$\beta2$	−1.698	−1.617	−1.248	−1.248	−1.172	−0.301	−0.820
Navigational diversity	μ	0.07	0.30	0.14	0.18	0.11	0.15	0.17
	M	0.15	0.3	0.12	0.18	0.09	0.16	0.18
	Min	0.02	0.21	0.05	0.1	0.01	0.05	0.07
	SD	0.053	0.061	0.057	0.060	0.056	0.058	0.061
	Skp	−0.562	−0.018	0.361	0.315	0.539	−0.416	−0.122
	$\beta2$	−1.359	−1.375	−0.794	−0.930	−0.193	−0.835	−1.205
Error states	μ	3.5	3.2	4.4	2.8	3.6	5.2	5.1
	M	1	1	2	1	1	2	3
	Min	1	1	2	1	1	2	1
	SD	1.23	2.32	2.33	1.98	1.78	3.43	3.89
	Skp	0.231	0.215	−0.211	0.106	0.213	−0.221	0.364
	$\beta2$	0.164	−0.122	−0.133	0.142	−0.144	0.135	−0.134
Total number of distinct states	μ	49.93	20.13	39.93	20.4	30.0	6.26	6.93
	M	55	18	43	21	31	6	8
	Min	33	14	25	13	25	4	4
	SD	9.735	4.748	8.171	5.193	3.817	1.751	1.980
	Skp	−0.346	0.887	−0.486	0.007	−0.039	−0.091	−0.247
	$\beta2$	−1.076	−0.567	−1.219	−1.651	−1.105	−1.366	−1.575

also applied Mann–Whitney U test to identify the significance of the averaged results from Table 5, Table 6 and Table 7. The results from Mann–Whitney U test are presented in Table 8.

It can be analyzed looking at Table 3 and Table 7 that Q^C outperforms Q^R in all the subject applications with different time budgets by achieving an overall improvement from 1%–26% in code coverage, 0.0003–0.38 in DOM diversity, 26–649 in navigational coverage and 0.01–0.15 in navigational diversity. Similarly, we also evident an improvement in the total number of distinct states (0–94) and error states (0–4). This shows that the contextual data input method complements the exploration strategy for dynamic web applications and helps explore states which cannot be explored with random data. For example, Fig. 14 shows a DOM state of Timeclock which is explored by Q^C but not Q^R because it needs a correct format and range of date to reach this DOM state of the report module in Timeclock. The context input data method in QExplore helps to complete user scenarios and discover states behind the web forms (as shown in Fig. 9).

This is noticeable that Q^C gives better results on applications that are more data-driven. For example, in the case of Claroline and Flex, we noticed large improvements in the results because these applications have several web forms that require specific input data to pass the validation constraints. The contextual input data method helps to provide the correct contextual data to explore the states behind the web forms, thus, improving the overall functional coverage and exploring more distinct DOM states that Q^R failed to explore. Similarly, Q^C captured more error states than Q^R for the same reason as Q^R was unable to explore those states with the random data.

Finding 5. QExplore with contextual data input method is more effective in exploring dynamic web applications than QExplore with random data input and help completing user scenarios.

When the results from Q^R are compared with C^R from Table 7 and Table 5, we can see that overall Q^R performs better than C^R including DOM coverage, DOM diversity, navigational coverage, navigation diversity and overall distinct states. This is mainly because C^R gets stuck in dynamic regions of web applications (e.g., pick date in Timeclock) and captures duplicate states which

Table 6
Averaged results for 15 runs with random data for 600, 1200 and 1800 s of time for WebExplor.

An overview of the results with 600 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	39.60	36.55	35.43	32.10	32.17	30.04	21.89
	M	40.68	36.21	38.02	34.4	33.55	30.33	22.8
	Min	29.66	25.86	26.84	21.5	20.5	21.67	14
	SD	7.130	8.222	5.253	7.244	7.268	7.761	3.636
	Skp	−0.100	0.374	−0.416	−0.126	−0.125	0.025	−0.482
	$\beta 2$	−1.360	−1.091	−1.468	−1.008	−1.355	−1.689	−0.154
DOM diversity	μ	0.19	0.51	0.34	0.45	0.48	0.50	0.40
	M	0.24	0.58	0.45	0.51	0.49	0.55	0.46
	Min	0.1	0.41	0.27	0.36	0.39	0.41	0.3
	SD	0.090	0.097	0.081	0.078	0.082	0.093	0.079
	Skp	−0.392	−0.212	−0.579	−0.422	0.0137	−0.095	−0.062
	$\beta 2$	−1.063	−1.287	0.726	−0.555	−1.348	−1.214	−0.815
Navigational coverage	μ	231.3	67.2	55.4	55.3	165.2	16.4	67.2
	M	231.4	67.07	55.38	54.66	164.93	16.37	66.85
	Min	230.36	66.23	54.55	54.38	164.28	15.47	66.243
	SD	0.519	0.504	0.626	0.495	0.4628	0.515	0.475
	Skp	−0.482	−0.080	0.227	1.172	0.328	−0.119	0.492
	$\beta 2$	−0.730	−0.792	−1.274	0.173	−1.334	−1.085	−0.375
Navigational diversity	μ	0.21	0.50	0.45	0.41	0.49	0.46	0.36
	M	0.2	0.45	0.4	0.4	0.5	0.41	0.36
	Min	0.12	0.4	0.35	0.36	0.39	0.36	0.27
	SD	0.052	0.055	0.063	0.044	0.063	0.064	0.054
	Skp	0.209	0.414	0.531	0.645	−0.212	0.306	−0.034
	$\beta 2$	−0.755	−1.385	−1.271	−0.674	−1.299	−1.345	−1.178
Error states	μ	3.4	1.5	3.1	2.1	3.9	3.3	0.6
	M	2	1	1	1	2	2	1
	Min	1	0	1	1	1	1	0
	SD	1.2	1.01	1.21	1.1	1.41	1.65	1.77
	Skp	−0.121	−0.354	−0.232	−0.223	0.114	0.162	−0.346
	$\beta 2$	−0.235	−0.410	−0.413	−0.232	−0.243	0.236	−0.242
Total number of distinct states	μ	46.73	21.2	47.53	7.46	17.26	6.93	54.73
	M	48	21	51	8	18	7	57
	Min	35	15	36	5	11	5	35
	SD	8.41	4.768	7.049	1.684	3.899	1.791	9.090
	Skp	−0.100	0.374	−0.417	−0.126	−0.125	0.025	−0.482
	$\beta 2$	−1.36	−1.092	−1.467	−1.008	−1.355	−1.689	−0.154
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	41.71	39.44	33.84	40.04	35.55	36.10	25.36
	M	42.52	39.77	30.58	41.74	34.72	38.05	25.81
	Min	27	28.17	22.5	27.22	25	22.38	17.2
	SD	9.742	9.747	7.889	7.307	6.881	8.785	6.129
	Skp	−0.137	0.472	0.092	−0.390	0.178	−0.469	−0.175
	$\beta 2$	−1.550	−1.041	−1.394	−1.38	−0.395	−1.107	−1.434
DOM diversity	μ	0.16	0.41	0.31	0.40	0.44	0.46	0.37
	M	0.19	0.48	0.31	0.51	0.57	0.55	0.4
	Min	0.07	0.32	0.22	0.34	0.34	0.39	0.29
	SD	0.081	0.078	0.08	0.085	0.092	0.088	0.090
	Skp	0.124	−0.352	0.409	−0.309	−0.535	−0.260	0.170
	$\beta 2$	−1.191	−0.553	−1.447	−1.358	−1.017	−1.198	−1.446
Navigational coverage	μ	340.3	121.1	798.8	125.4	434.3	88.9	157.2
	M	340.37	120.57	798.91	125.04	434.49	89.05	157.21
	Min	39.47	120.26	798.14	124.5	433.43	88.18	156.43
	SD	0.485	0.554	0.549	0.562	0.620	0.557	0.502
	Skp	0.012	1.41	0.070	0.710	−0.155	−0.265	−0.177
	$\beta 2$	−0.397	0.809	0.809	−1.348	−0.420	−1.350	−1.508
Navigational diversity	μ	0.19	0.47	0.42	0.33	0.44	0.41	0.31
	M	0.17	0.47	0.45	0.34	0.46	0.42	0.34
	Min	0.1	0.39	0.34	0.24	0.35	0.31	0.23
	SD	0.052	0.049	0.056	0.069	0.065	0.067	0.054
	Skp	0.204	−0.374	−0.503	−0.085	−0.176	−0.257	−0.128
	$\beta 2$	−0.768	−1.219	−1.056	−1.519	−1.459	−1.306	−1.068

(*continued on next page*)

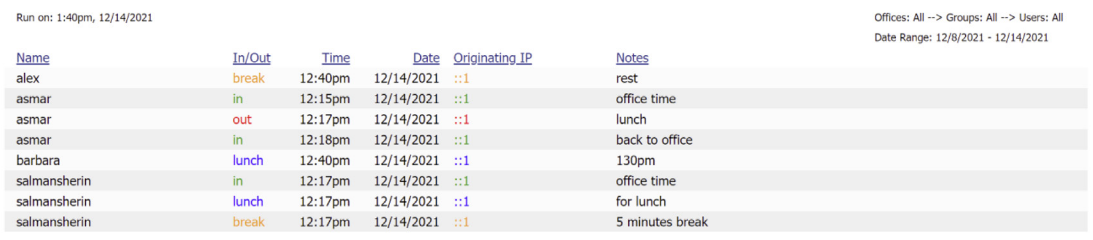
limit the exploration and reduce the overall diversity of DOM and navigational diversity.

Finding 6. QExplore with random data is more effective in exploring dynamic web applications than Crawljax.

In comparison to W^R, Q^R performed better in achieving coverage, navigational coverage, navigational diversity, distinct DOM states and error states, as shown in Table 7 and Table 6. This is mainly because of the difference in the exploration approach of both Q^R and W^R. In W^R, a state representation is based on

Table 6 (*continued*).

Error states	μ	4.02	3.2	4.8	3.2	4.7	5.2	1.1
	M	2	1	2	1	2	2	1
	Min	2	1	1	0	1	2	0
	SD	1.32	2.1	2.54	2.41	2.97	3.32	0.96
	Skp	−0.321	0.212	−0.364	−0.341	−0.134	−0.142	0.134
	$\beta2$	0.127	−0.135	0.143	−0.123	0.145	−0.143	−0.132
Total number of distinct states	μ	61.8	23.8	58.66	22.06	25.6	16.13	61.93
	M	63	24	53	23	25	17	63
	Min	40	17	39	15	18	10	42
	SD	14.433	5.882	13.678	4.026	4.954	3.925	14.959
	Skp	−0.137	0.472	0.092	−0.390	0.178	−0.469	−0.174
	$\beta2$	−1.550	−1.041	−1.394	−1.238	−0.393	−1.108	−1.435
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	46.19	45.55	50.18	48.88	46.23	38.81	27.81
	M	46.24	49.4	52.52	50.75	50.23	34.14	28.42
	Min	32.84	28.81	30.64	31.5	29.3	26.55	19.41
	SD	10.676	10.309	9.816	11.253	10.584	11.217	5.460
	Skp	0.233	−0.398	−0.647	−0.256	−0.238	0.449	−0.205
	$\beta2$	−1.274	−1.406	−0.748	−1.463	−1.466	−1.456	−1.343
DOM diversity	μ	0.173	0.47	0.30	0.37	0.37	0.41	0.33
	M	0.24	0.51	0.37	0.41	0.49	0.45	0.46
	Min	0.1	0.37	0.2	0.28	0.32	0.32	0.25
	SD	0.097	0.098	0.106	0.088	0.075	0.1	0.099
	Skp	−0.140	0.152	−0.338	0.089	−0.751	−0.073	−0.487
	$\beta2$	−1.284	−1.308	−1.166	−1.003	−0.456	−1.349	−1.230
Navigational coverage	μ	543.4	412.1	1305.4	178.3	487.1	134.2	187.9
	M	543.54	412.19	1305.53	178.69	486.9	134.1	187.73
	Min	542.47	411.32	13.4.44	177.53	486.11	133.23	186.97
	SD	0.586	0.548	0.581	0.603	0.574	0.503	0.676
	Skp	−0.269	−0.140	−0.101	−0.502	0.086	0.150	0.454
	$\beta2$	−0.916	−1.383	−1.574	−0.926	−1.013	−0.846	−0.772
Navigational diversity	μ	0.17	0.35	0.37	0.31	0.37	0.35	0.26
	M	0.14	0.35	0.41	0.33	0.37	0.39	0.28
	Min	0.07	0.28	0.28	0.23	0.29	0.28	0.16
	SD	0.062	0.062	0.069	0.059	0.049	0.044	0.066
	Skp	0.0623	−0.302	−0.138	−0.299	−0.152	−1.292	−0.243
	$\beta2$	−0.705	−1.385	−1.641	−1.207	−1.196	−1.196	−1.373
Error states	μ	4.3	4.5	7.7	5.2	4.4	5.3	2.1
	M	2	2	3	2	3	3	1
	Min	0	0	3	2	1	2	0
	SD	1.67	2.56	4.66	3.53	2.35	3.88	1.13
	Skp	0.131	0.135	−0.213	−0.412	−0.334	0.221	−0.334
	$\beta2$	0.146	−0.132	−0.121	0.214	−0.213	−0.114	−0.134
Total number of distinct states	μ	68.93	33.2	68.8	27.93	33.13	20.46	80.26
	M	69	36	72	29	36	18	82
	Min	49	21	42	18	21	14	56
	SD	15.934	7.513	13.459	6.430	7.58	5.914	15.759
	Skp	0.233	−0.397	−0.647	−0.256	−0.238	0.449	−0.204
	$\beta2$	−1.274	−1.406	−0.748	−1.463	−1.466	−1.456	−1.343


Run on: 1:40pm, 12/14/2021

Offices: All --> Groups: All --> Users: All
Date Range: 12/8/2021 - 12/14/2021

Name	In/Out	Time	Date	Originating IP	Notes
alex	break	12:40pm	12/14/2021	::1	rest
asmar	in	12:15pm	12/14/2021	::1	office time
asmar	out	12:17pm	12/14/2021	::1	lunch
asmar	in	12:18pm	12/14/2021	::1	back to office
barbara	lunch	12:40pm	12/14/2021	::1	130pm
salmansherin	in	12:17pm	12/14/2021	::1	office time
salmansherin	lunch	12:17pm	12/14/2021	::1	for lunch
salmansherin	break	12:17pm	12/14/2021	::1	5 minutes break

Fig. 14. An example DOM state from Timeclock capture by Q^C.

the structure of the HTML page. The problem with structural representation of the state in dynamic web applications is that different business logic can have the same structural representation restricting the exploration strategy to consider DOM states with different business logic as one state. However, Q^R and Q^C represent each state in terms of a set of interactable elements which helps in exploring diverse and distinct DOM states instead of similar or duplicate states. Moreover, we found that the reward function of W^R does not consider the potential of a DOM state that can lead to several new DOM states. This means that W^R assigns

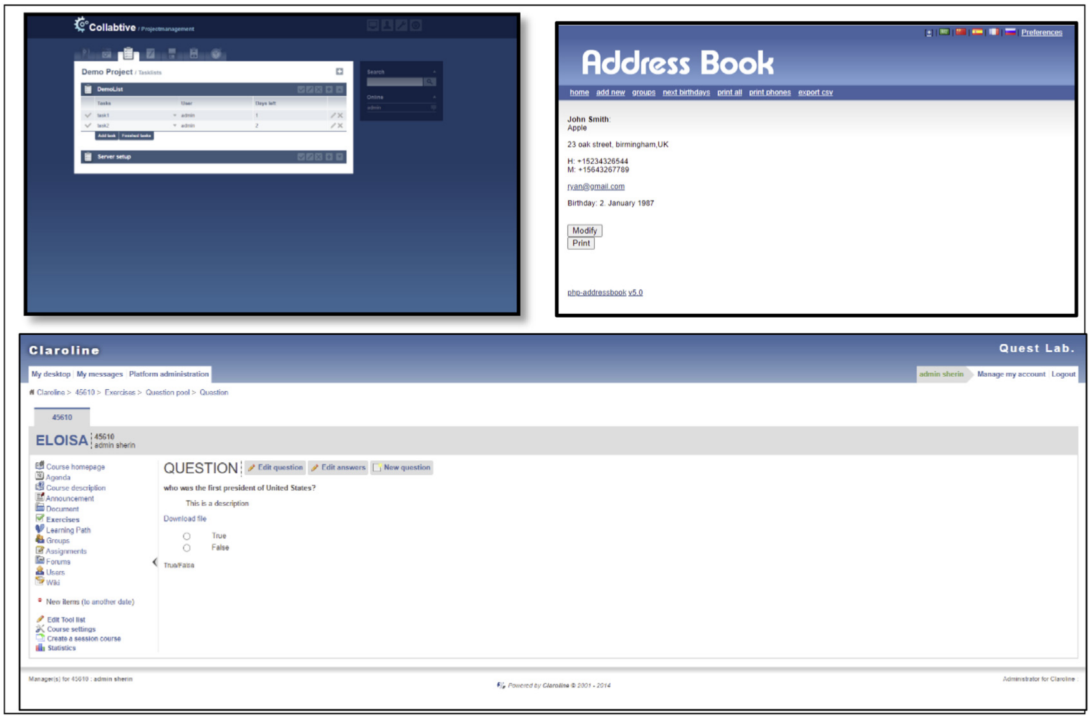


Fig. 15. Examples of DOM states captured only with Q^C.

the same weight to two DOM states having different number of interactable elements. However, a DOM state with a greater number of interactable elements has the potential to lead to new DOM states more than the one with fewer interactable elements. Fig. 15 shows different DOM states from our open-source subject applications captured only by Q^R within the time budget of 600 seconds based on exploring DOM states having the potential of several other DOM states.

Finding 7. QExplore with random data input is more effective than WebExplor in terms of code coverage, navigational coverage, distinct and error DOM states.

The above results show that QExplore outperformed Crawljax and WebExplor even if the results from their best runs out of 15 runs are compared. It can be seen from the results in Table 8 that Q^R significantly outperformed C^R and W^R in all the cases (except few) calculated with Mann–Whitney U test at confidence level 0.05. In cases where the results are not statistically significant are highlighted in italic in Table 8. In code coverage, there are only two cases where the difference is not significant when Q^R is compared to C^R. However, in all the other cases the results are statistically significant which means the Q^R performed better than other approaches in achieving better coverage. Similarly, in DOM diversity there are four cases out of 21 between Q^R and W^R and two cases between Q^R and C^R in which the difference is not statistically significant between. However, looking at the averaged results Q^R performed better then W^R and C^R in all 21 cases. One of the problems with W^R is that it gives equal wait to all the web element in its reward function. This makes it difficult for W^R to distinguish between clicking on a single action vs actions in a sequence. Such difference is important to maintain order dependency between the actions specifically in data-driven applications. For example, W^R assigns equal weight to *username*, *password* and *submit* button on a login page which makes it difficult for W^R to login without maintaining specific order for the actions to perform. Moreover, W^R requires more time for learning big applications like Claroline and takes more time in exploration of the application. Consequently, there is only one case in the number of error states and two cases in the number of distinct states between Q^R and W^R in which there is no significant improvements, however, in all the other cases the difference is found to be statistically significant. This means that in most of the cases the results are improved significantly by Q^R followed by W^R and C^R.

QExplore is designed to work with contextual input data rather than random data. However, we empirically evaluated QExplore with both, contextual input data and random data. Our experiment shows that QExplore with the random data performs better when the results from all the runs are averaged (shown in Table 5, Table 6 and Table 7). Also, the results obtained are statistically significant in most of the cases or equal in only a few cases. The better performance of QExplore with random data indicates the effectiveness of the crawling strategy proposed in the paper. In terms of contextual data, we analyzed that QExplore with contextual data outperformed Q^R followed by W^R and C^R despite with the results from their best run, as shown in Table 3.

6. Discussion and research findings

The main application of our dynamic exploration approach is in the automated testing of dynamic web applications. It automatically derives an SFG that can be used to automate several web analysis and software testing techniques such as invariant-based testing, regression testing and cross-browser testing. Our approach can detect DOM elements with their particular DOM properties and systematically exercise each DOM element. It also supports the contextual data input method which helps in the detection of DOM states behind the forms. The contextual data input method helps in capturing complete sequences from the index to the terminal DOM state. Our approach uses semantic similarity

Table 7
Averaged results for 15 runs with random data for 600, 1200 and 1800 s of time for QExplore with random data.

An overview of the results with 600 s of exploration time

		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	61.10	50.44	52.23	44.32	38.98	48.43	27.54
	M	59.53	48.36	51.16	36.73	37.54	39.09	26.85
	Min	58.3	43.71	48.9	30.72	32.72	31.01	25.99
	SD	0.8771	2.298	1.018	4.795	2.152	4.586	0.473
	Skp	−0.133	−0.525	−0.399	−0.015	−0.331	0.207	−0.411
	$\beta2$	−1.159	−1.123	−0.804	−1.346	−1.361	−0.255	−0.872
DOM diversity	μ	0.21	0.60	0.43	0.53	0.51	0.57	0.47
	M	0.29	0.73	0.48	0.57	0.58	0.67	0.52
	Min	0.18	0.53	0.36	0.44	0.41	0.47	0.38
	SD	0.070	0.094	0.076	0.079	0.088	0.101	0.093
	Skp	−0.288	−0.573	0.295	−0.235	−0.119	−0.405	−0.031
	$\beta2$	−1.158	−1.151	−0.510	−0.812	−1.394	−1.076	−1.137
Navigational coverage	μ	432.6	187.6	108.5	204.7	288.4	48.9	154.6
	M	432.62	187.42	107.94	204.72	288.52	48.98	154.87
	Min	431	186	107	203	287	47	153
	SD	0.526	0.585	0.622	0.686	0.521	0.619	0.587
	Skp	0.158	0.209	0.514	0.008	0.192	−0.087	−0.268
	$\beta2$	−0.719	−1.287	−1.377	−1.342	−1.370	−1.420	−1.091
Navigational diversity	μ	0.42	0.69	0.82	0.84	0.71	0.80	0.41
	M	0.41	0.74	0.83	0.84	0.73	0.78	0.41
	Min	0.34	0.6	0.72	0.78	0.62	0.71	0.31
	SD	0.057	0.54	0.063	0.050	0.058	0.050	0.047
	Skp	0.0457	−0.948	−0.368	0.156	−0.058	0.381	−0.532
	$\beta2$	−1.185	−0.164	−1.037	−1.473	−1.054	−0.605	−0.894
Error states	μ	4.8	2.8	3.7	2.9	4.8	4.7	1.7
	M	4.2	2.5	3.5	2.6	4.7	4.4	1.5
	Min	4	2	3	2	4	4	1
	SD	2.01	1.11	1.7	1.3	2.1	2.23	0.45
	Skp	−0.116	−0.421	−0.341	−0.164	−0.216	−0.341	−0.412
	$\beta2$	−0.213	−0.524	−0.346	−0.492	−0.745	−1.346	−0.145
Total number of distinct states	μ	48.8	24.0	51.88	9.0	21.33	9.0	59.63
	M	50	24	52	9	22	8	60
	Min	47	22	50	7	19	7	58
	SD	0.71	1.167	1.048	1.182	1.277	1.037	1.259
	Skp	−0.132	−0.524	−0.399	−0.016	−0.331	0.207	−0.417
	$\beta2$	−1.161	−1.125	−0.803	−1.345	−1.362	−0.256	−0.870

An overview of the results with 1200 s of exploration time

		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	66.41	55.45	56.78	63.23	51.11	52.23	33.31
	M	64.43	52.85	55.08	59.02	48.37	45.77	32.32
	Min	62.85	49.64	53.82	54.95	46.36	42.56	31.7
	SD	1.155	1.432	1.092	2.354	1.442	3.665	0.529
	Skp	0.288	−0.446	0.193	0.296	0.138	0.315	0.474
	$\beta2$	−1.040	0.323	−0.508	−0.727	−1.021	−1.524	−0.984
DOM diversity	μ	0.20	0.58	0.40	0.51	0.50	0.55	0.45
	M	0.21	0.62	0.45	0.51	0.49	0.57	0.47
	Min	0.1	0.52	0.31	0.43	0.41	0.45	0.37
	SD	0.096	0.0946	0.071	0.067	0.066	0.084	0.084
	Skp	0.222	0.132	−0.357	0.922	0.268	−0.048	0.690
	$\beta2$	−1.244	−1.665	−1.076	−0.002	−1.015	−1.392	−0.708
Navigational coverage	μ	633.1	434.5	1405.2	234.6	568.2	134.9	204.4
	M	633.04	1405.5	434.99	234.15	568.46	134.8	204.44
	Min	632	433	1404	233	567	134	203
	SD	0.650	0.552	0.656	0.689	0.578	0.545	0.595
	Skp	0.255	−0.585	−0.388	0.523	−0.398	0.555	−0.089
	$\beta2$	−1.203	−0.663	−1.029	−1.275	−0.935	−1.56	−0.987
Navigational diversity	μ	0.42	0.66	0.81	0.73	0.67	0.75	0.40
	M	0.42	0.62	0.79	0.74	0.7	0.75	0.43
	Min	0.34	0.57	0.72	0.63	0.57	0.67	0.3
	SD	0.056	0.057	0.050	0.058	0.051	0.055	0.067
	Skp	−0.001	0.645	0.920	−0.357	−0.922	0.236	−0.210
	$\beta2$	−1.218	−0.809	0.479	−1.108	0.184	−1.205	−1.346

(continued on next page)

Table 7 (*continued*).

Error states	μ	4.8	5.4	7.4	5.2	5.4	8.4	3.3
	M	4.3	5	7	4.8	5.2	8	3
	Min	4	5	6	5	5	7	2
	SD	1.5	2.2	3.2	2.06	2.3	3.4	1.2
	Skp	−0.614	−0.251	−0.616	−0.252	−0.341	−0.174	−0.180
	β2	−0.512	−0.413	−0.523	−0.612	−0.341	−0.523	−0.412
Total number of distinct states	μ	66.87	34.4	63.98	25.65	37.76	18.7	68.65
	M	66	34	63	25	37	18	68
	Min	65	32	62	23	35	16	67
	SD	1.199	0.935	1.268	1.015	1.116	1.451	1.119
	Skp	0.288	−0.443	0.191	0.296	0.137	0.315	0.482
	β2	−1.041	0.323	−1.323	−1.509	−0.727	−1.019	−1.525
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	μ	71.32	60.40	64.66	72.77	61.60	63.22	35.98
	M	70.21	58.94	63.23	68.67	59.21	60.47	35
	Min	68.29	55.57	61.62	65.72	57.37	55.26	34.46
	SD	1.014	1.655	0.881	2.300	1.303	2.362	0.477
	Skp	−0.304	−0.333	−0.110	0.494	−0.106	−0.494	0.411
	β2	−1.049	−1.414	−0.701	−0.968	−1.194	−0.709	−1.141
DOM diversity	μ	0.19	0.53	0.38	0.50	0.39	0.44	0.40
	M	0.24	0.52	0.45	0.56	0.42	0.54	0.52
	Min	0.09	0.43	0.29	0.48	0.3	0.35	0.33
	SD	0.106	0.085	0.080	0.079	0.103	0.097	0.084
	Skp	−0.141	0.369	−0.177	0.206	0.151	−0.341	−0.425
	β2	−1.334	−0.867	−1.575	−1.587	−1.560	−1.239	−0.975
Navigational coverage	μ	765.7	601.4	1703.6	368.4	756.3	188.4	288.6
	M	765.44	601.29	1703.65	368.35	755.95	188.25	288.67
	Min	764	600	1702	367	755	187	287
	SD	0.631	0.507	0.465	0.600	0.525	0.449	0.621
	Skp	0.014	0.332	−0.461	0.077	0.461	0.220	−0.162
	β2	−1.557	−1.325	−0.810	−1.581	−0.847	−0.578	−1.048
Navigational diversity	μ	0.37	0.63	0.74	0.68	0.61	0.72	0.38
	M	0.39	0.61	0.74	0.67	0.65	0.75	0.39
	Min	0.27	0.54	0.64	0.59	0.52	0.65	0.29
	SD	0.051	0.049	0.058	0.053	0.073	0.055	0.058
	Skp	−1.023	0.215	0.133	−0.124	−0.135	−0.277	−0.169
	β2	−0.045	−1.066	−0.990	−1.258	−1.857	−1.297	−1.456
Error states	μ	6.5	6.9	8.9	6.3	5.7	8.4	3.7
	M	6	6.5	8.5	6	5.5	8	3.5
	Min	5	5	8	5	5	6	3
	SD	1.8	2.7	3.2	2.33	2.04	3.5	2.13
	Skp	−0.177	−0.170	−0.193	−0.484	−0.437	−0.517	−0.367
	β2	−0.461	−0.524	−0.356	−0.526	−0.414	−0.134	−0.164
Total number of distinct states	μ	77.32	39.55	77.22	38.57	42.22	26.3	88.46
	M	78	40	77	38	42	26	87
	Min	75	38	75	36	40	24	86
	SD	1.127	1.137	1.079	1.282	0.929	1.049	1.198
	Skp	−0.303	−0.333	−0.112	0.494	−0.106	−0.494	0.419
	β2	−0.978	−1.050	−1.414	−0.698	−0.969	−1.193	−0.710

that allows users to define their crawl rules by using keywords rather than locators such as ID or XPath which requires reading of the source code. Moreover, our approach is incremental in nature that reduces the overall exploration time and supports multiple values for a single field, thus, overcoming the limitations of the existing approaches.

The results show that QExplore substantially outperforms Crawljax in terms of diversity, coverage, DOM states, and error states within the time constraints of 600 s, 1200 s and 1800 s. QExplore also outperforms WebExplor in terms of code coverage, navigational coverage and diversity and distinct and error DOM states in all the three runs. The derived state-flow graph from QExplore with better code and navigational coverage will help to increase the chances of bug detection in different parts of the web application. Similarly, the increased diversity in the state-flow graph will help to reduce the presence of irrelevant DOM states (discussed in detail in Section 1). The effectiveness of software testing techniques, particularly model-based testing, is as good as the test model itself. If the derived test model omits any DOM state, then the bugs in that part of the application will go unknown/unexplored. The following are the lessons that we learned while conducting this study.

Lessons learned and future research directions

- *DOM equivalence/comparison*
  DOM equivalence is a crucial step in the dynamic exploration approach in order to identify different DOM states. Primarily a new DOM tree is considered as a new DOM state. There are different ways used in the existing studies to compare and differentiate between the DOM states. For instance, in most Crawlers, the string representations of the DOM states are used to identify the difference in states. The strings are compared by using string-based Levenshtein distance (Mesbah et al., 2008) or by computing the hash value of the complete contents (Duda et al., 2009). But the main problem with these approaches is that they ignore

Table 8
Mann–Whitney U Test results for 600, 1200 and 1800 s.

		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
An overview of the results with 600 s of exploration time								
Coverage	Q^R - C^R	0.00076	0.00024	0.00133	*0.07822*	0.00071	0.00665	0.00273
	Q^R - W^R	0.00008	0.00015	0.00765	0.01117	0.04301	0.00026	0.00155
DOM diversity	Q^R - C^R	0.01647	0.00508	0.00013	0.00057	*0.35999*	0.01814	0.01718
	Q^R - W^R	*0.11340*	0.00193	0.01572	0.00782	0.02596	0.00967	0.01881
Navigational coverage	Q^R - C^R	0.00077	0.00010	0.00020	0.00003	0.00040	0.00005	0.00008
	Q^R - W^R	0.00006	0.00012	0.00074	0.00004	0.00001	0.00003	0.00009
Navigational diversity	Q^R - C^R	0.00038	0.00106	0.00112	0.00032	0.00011	0.00142	0.00010
	Q^R - W^R	0.00060	0.00050	0.00063	0.00057	0.00050	0.00048	0.01114
Error states	Q^R - C^R	0.04243	0.02322	0.03443	0.03422	0.04335	0.03325	0.02112
	Q^R - W^R	0.00334	0.00556	0.04112	0.00558	0.00985	0.02248	0.04526
Total number of distinct states	Q^R - C^R	0.00545	0.00027	0.00769	0.00534	0.00015	0.00722	0.00131
	Q^R - W^R	*0.27949*	0.01815	0.01250	0.00193	0.00020	0.00071	0.00490
An overview of the results with 1200 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	Q^R - C^R	0.00765	0.00048	0.00121	0.00865	0.00632	*0.25418*	0.01939
	Q^R - W^R	0.00076	0.00087	0.00076	0.00075	0.00066	0.00553	0.00476
DOM diversity	Q^R - C^R	0.04065	0.00023	0.00075	0.00229	*0.20841*	0.00181	0.04298
	Q^R - W^R	0.04836	0.00071	0.00571	0.04528	*0.49245*	0.04574	0.01806
Navigational coverage	Q^R - C^R	0.00021	0.00002	0.00001	0.00003	0.00006	0.00012	0.00011
	Q^R - W^R	0.00011	0.00006	0.00021	0.00044	0.00034	0.00090	0.00045
Navigational diversity	Q^R - C^R	0.00025	0.00120	0.00005	0.00002	0.00009	0.00044	0.00024
	Q^R - W^R	0.00019	0.00024	0.00066	0.00082	0.00014	0.00044	0.00031
Error states	Q^R - C^R	0.04324	0.02232	0.03321	0.04413	0.04061	0.03612	0.03011
	Q^R - W^R	0.03010	0.02025	0.02103	0.01081	0.02013	0.02042	0.03031
Total number of distinct states	Q^R - C^R	0.00767	0.00762	0.00763	0.01362	0.00752	0.00722	0.00769
	Q^R - W^R	*0.21978*	0.00062	0.00141	0.00830	0.00009	0.04663	0.01588
An overview of the results with 1800 s of exploration time								
		Schoolmate	Timeclock	Addressbook	Collabtive	Claroline	Brotherhood	Flex
Coverage	Q^R - C^R	0.00762	0.00659	0.00770	0.00750	0.00739	0.00082	0.00074
	Q^R - W^R	0.00070	0.00390	0.00092	0.00073	0.00093	0.00076	0.00003
DOM diversity	Q^R - C^R	0.00149	0.00021	0.00075	0.00042	0.00027	0.04251	0.00031
	Q^R - W^R	*0.31834*	0.04101	0.04135	0.00023	*0.92694*	0.04182	0.02174
Navigational coverage	Q^R - C^R	0.00074	0.00004	0.00001	0.00003	0.00006	0.00022	0.00015
	Q^R - W^R	0.00036	0.00071	0.00070	0.00003	0.00040	0.00015	0.00018
Navigational diversity	Q^R - C^R	0.00056	0.00077	0.00033	0.00045	0.00022	0.00019	0.00043
	Q^R - W^R	0.00006	0.00082	0.00042	0.00053	0.00038	0.00055	0.00041
Error states	Q^R - C^R	0.02074	0.03404	0.04401	0.03603	0.03806	0.02322	0.04015
	Q^R - W^R	0.01131	0.02064	0.03051	*0.05232*	0.03531	0.03012	0.02244
Total number of distinct states	Q^R - C^R	0.00763	0.00755	0.00770	0.00750	0.00739	0.00073	0.00076
	Q^R - W^R	0.00682	0.00119	0.03982	0.00073	0.00011	0.00607	0.04315

the DOM tree structure of the states. To consider the tree structure of the DOM, *tree edit distance* is used to cope with state comparison. However, the problem with the *tree edit distance* is the identification of recurring DOM states which leads the Crawlers to stuck in an infinite loop. For example, consider a simple dynamic To-do list application with DOM states for every possible to-do item. Therefore, tree edit distance might not be suitable for the exploration of dynamic web applications where there can be infinite DOM states leading towards the state explosion problem. In our approach, we use a modified technique of computing the hash value for state comparison. We capture the interactable elements from the DOM state and transform them into an intermediate representation to calculate the hash value from selective attributes of the interactable elements (discussed in Section 3.2). This method resolved the problems of the previous studies up to some extent. But the problem with this method is that the DOM states with no interactable elements (e.g., the report page in Timeclock application) might have the same hash value resulting in overwriting of the DOM states. Therefore, we additionally use the *tree edit distance* to compare such DOM states to avoid state overwriting. Though, our approach overcomes the problem of DOM comparison by using the combination of both techniques (hash computing and tree edit distance). But we observed few issues with this technique which requires further research. For example, consider a DOM state with hidden elements in the DOM. If an event occurs that makes the hidden elements visible on the interface with no explicit change in the tree structure of the DOM or its contents. Then, it is difficult identify the difference between such two DOM states either by computing hash or *tree edit distance* because for both techniques the DOM states are similar. However, there is a difference from user perspective. One possible solution to this problem is the use of image processing techniques to compare the screenshots of the two DOM states to compare and analyze the difference between them. We believe that comparing DOM states is challenging that requires more research work and attention from researchers in the area.

- *Coping with dynamism in locating interactable elements*
Locating interactable elements in the dynamic web application is challenging. This is because in dynamic web applications the attributes (ID, name, class) that are used to locate different interactable elements are highly dynamic that changes frequently. This dynamism in the attributes of interactable elements makes it difficult for the Crawler to locate them in the DOM. To cope up with dynamism, we determine the DOM actions at run time for every DOM state. To locate different DOM actions, we use specific attributes for locating various types of interactable elements (as discussed in Section 3.2). Though, our approach copes up with dynamism up to some extent, however, there are still exceptions that are difficult to handle and requires further research. For example, the attributes that we are using in our approach for locating DOM actions changes dynamically at a particular DOM state. Then, it will be treated as new DOM actions every time the same DOM state appears. This will result in multiple string representations of the same interactable elements in the Q-table which increases the action space unnecessarily. One possible solution is the use of fine-grained heuristics for the identification of similar DOM actions at a particular DOM state. Another possible solution for the identification of similar DOM elements is the use of machine learning models. We believe that identification of interactable elements in dynamic web application is challenging and still requires further research work.
- *Identification of error states*
Identification of failure in the application is also critical to software testing. Currently, we detect common DOM errors or server side errors propagated to the client side like existing studies (Benedikt et al., 2002; Wang et al., 2009). However, there are several bugs such as relevant to domain or business logic that still needs sophisticated techniques for detection and localization. There are only few works available on addressing this problem. For instance, Groeneveld et al. (2010) proposed a technique for automatic detection of invariants in dynamic web applications that can serve as an oracle for the fault detection. Similarly, Ocariza Jr. et al. (2012) proposed a technique to automatically localize faults in client-side JavaScript code. Liu et al. (2011) proposed a technique to automatically generate assertions for the test scripts by using data mining techniques. Research in this area is still under way and more primary works are expected to improve the identification of failures by using machine learning models for prediction and detection.

7. Limitations & threats to validity

7.1. Limitations

QExplore supports the exploration of dynamic web applications and achieved better results than the state-of-the-art Crawljax. However, there are several limitations specific to only QExplore or common to all dynamic exploration approaches. We discuss the limitations in detail as follows:

7.1.1. Interactable elements

The identification of interactable elements is always challenging for researchers due to the diversity and complexity of the web applications (Maras et al., 2013). The area of identifying complex interactable elements is still under research such as drag and drop event and its execution. Similarly, the detection of DOM states based on time events is also not supported by QExplore. The list of interactable elements supported by QExplore is given in Table 1. Currently, all the existing dynamic exploration approaches support a limited number of interactable elements and QExplore is not an exception to it. More research is underway on the detection of complex interactable elements for exploration of dynamic web applications. In future, we aim to extend QExplore to identify interactable elements programmed with JavaScript.

7.1.2. Accuracy of contextual input data method

QExplore rely on using fastText4 to map the data from the generator to the input fields in the web forms by matching semantic similarity (as discussed in Section 3.6). It helps the crawlers to automatically feed the data into the input fields and explore DOM states behind the complex web forms. However, we do not claim that this is the best ML model to use for mapping the data to the input fields and there are other ML models such as word2Vec (Mikolov et al., 2013) and glove embeddings (Pennington et al., 2014) that can be used for the same purpose. However, we empirically found the model to be working effectively in QExplore which outperforms Crawljax.

7.2. Threats to validity

This section reported various internal and external threats to the validity of the study and the ways to overcome them.

7.2.1. Internal threats

7.2.1.1. Code instrumentation. In order to measure the code coverage in each subject application, we instrumented the code of the target subject application in a manner that does not affect the actual behavior of the code. To instrument the code in our subject applications, we followed the guidelines suggested in other empirical studies to carefully instrument the application code (Sherin et al., 2021; Zou et al., 2014).

7.2.1.2. Evaluation metrics. The metrics we used for the evaluation of the effectiveness of a test model might be a threat to internal validity. However, all these metrics capture various properties of a test model and are used by other studies for the same purpose (Fard and Mesbah, 2013; Qi et al., 2019; Benedikt et al., 2002). Therefore, we believe that the identified metrics measure the effectiveness of a test model which reflects the effectiveness of the dynamic exploration approach used.

7.2.1.3. Reliability of results. We believe that the results gathered from our experiment are reliable and reproducible because QExplore and all the subject applications except the “FLEX”, which is an industrial case study, are available online.[13] The automated scripts used to evaluate the code coverage, DOM diversity, navigation coverage, navigation diversity, error states and the number of states is also made available for other researchers to reproduce the same results.[14]

7.2.1.4. Experimenter bias. Another threat to the internal validity might be to run both Crawljax and QExplore on the target subject applications with the same databases. For example, if we run Crawljax on Timeclock and then run QExplore on the same Timeclock without truncating the database then QExplore will automatically result in more states. This is because the data forms filled by the Crawljax will automatically increase the number of interactive elements which will increase the number of states resulting in the experimenter bias. To overcome this threat, we have run the same subject applications on different machines for

13 http://www.github.com/salmansherin/QExplore

14 http://www.github.com/salmansherin/QExplore/results

Crawljax and QExplore. Both the machines were having the same hardware specifications (mentioned in Section 5.2) but to reduce the bias of sharing the same database by the subject applications for Crawljax and QExplore.

7.2.2. External threats

7.2.2.1. Generalizability. One of the threats to external validity is the generalizability of the outcomes to other dynamic web applications than the target subject applications in this study. We admit that more dynamic web applications should be examined to support the claims drawn from the results. However, we overcome this threat by selecting seven open-source dynamic web applications from different domains and sizes used in several other studies (Sherin et al., 2021; Imtiaz and Iqbal, 2021; Zou et al., 2014; Mirzaaghaei and Mesbah, 2014). Consequently, we also evaluated our approach on an industrial case study 'FLEX' to measure the effectiveness in a more realistic context.

Another external threat to QExplore is the use of Mocker data generator which provides data to the web forms during the exploration of the application. However, our approach is not limited to only using the Mocker data generator and other automated data generation techniques such as Biagiola et al. (2017) and Bozkurt and Harman (2011) can be used with our dynamic exploration technique. Moreover, Mocker data generator is completely open-source and accessible for other researchers in the area.

7.2.2.2. Replication of results. To replicate the results of our study, we made QExplore publicly available for other researchers along with all the automated scripts used in the empirical evaluation for measuring the metrics. The availability of our tool will motivate other researchers to replicate the results but also to contribute to the area of exploring dynamic web applications.

8. Related work

Dynamic exploration approaches are being studied extensively in the past two decades (Van Deursen et al., 2015; Li et al., 2014; Doğan et al., 2014). Different approaches and algorithms are proposed in the literature to improve the effectiveness and efficiency of exploring web applications. The advent of web 2.0 poses several challenges to the software testing that made the dynamic analysis of modern web applications indispensable. Therefore, recent studies focus on the exploration of dynamic web applications to address the challenges in crawling these applications which is comparatively a new research area (Muñoz et al., 2018; Liu et al., 2020; Qi et al., 2019; Yandrapally et al., 2020). In general, dynamic exploration approaches used for the construction of a test model can be classified into two broad categories, i.e., generic exploration approaches (e.g. Crawljax Mesbah et al., 2008) and guided or feedback directed exploration approaches (e.g. FeedEx Fard and Mesbah, 2013). Generic exploration approaches use generic exploration strategies such as DFS or BFS to explore the state-space of a web application as a whole whereas guided approaches restrict the scope of exploration to specific scenarios or goals. For this reason, guided approaches are also called scoped crawling (Olston and Najork, 2010; Gao et al., 2013). Generic exploration approaches require less or no prior knowledge of the target web application and therefore can be used for exploring unknown applications. In contrast, guided exploration approaches require prior knowledge of the web application under test, thus, may not be effective when given an unknown application. In fact, one of the main advantages of the dynamic exploration approach is to explore the state-space of an unknown application. Therefore, we propose a generic dynamic exploration approach for the exploration of dynamic web applications. A large number of studies exist that focus on the exploration of dynamic web applications. However, there exist several limitations in the existing approaches that QExplore aims to alleviate. In this section, we discuss the most relevant literature on exploring dynamic web applications to report their limitations and differences with our approach.

8.1. Generic exploration approaches

Generic approaches require no or less prior knowledge of a web application to derive a test model and can be run on dynamic applications having dynamic features from either client-side (JavaScript) or Server-side scripting languages (e.g., PHP). There are several approaches proposed in the literature which are discussed as follow:

Benedikt et al. (2002) presented an approach called VeriWeb to explore the execution paths in traditional web applications. Wang et al. (2009) proposed a combinatorial strategy for the exploration of dynamic web applications and handling of web forms. A more related to our approach is the state-of-the-art approach, Crawljax, proposed by Mesbah et al. (2008) in 2008. Crawljax automatically explores the state-space of AJAX-based web applications and reverse engineers a state-flow graph that automates several web analysis and testing approaches. For example, invariant-based testing (Mesbah et al., 2011) and regression testing (Mirshokraie and Mesbah, 2012). Similarly, DOM-based coverage criteria called DOMCovery are also based on Crawljax (Mirzaaghaei and Mesbah, 2014).

A common limitation of generic exploration techniques including Crawljax is the use of generic searches such as DFS or BFS which get stuck in a loop in dynamic regions of web applications (discussed in Section 2 in detail). This is mainly due to two reasons: (1) the crawling strategy used by generic approaches have no feedback to guide the crawling process and (2) the existing generic approaches differentiate one DOM state from another based on small DOM changes in the DOM which may not be important from a testing perspective. In contrast, QExplore is inspired from Q-learning as a crawling strategy that uses a reward function and a Q-value function as feedback to guide the crawling process. Moreover, QExplore differentiates the DOM states on the basis of interactive DOM elements rather than small DOM changes in the DOM during the crawling process.

8.2. Guided exploration approaches

Guided exploration approaches partially explore a web application by limiting the scope of exploration to derive a partial test model instead of a complete test model. Guided exploration approaches are mostly built on generic exploration techniques. For instance, Fard and Mesbah (2013) proposed a feedback-directed exploration approach to guide the exploration process by using heuristics i.e. coverage, navigational diversity, DOM diversity by reducing the test model size. Similarly, Dincturk et al. (2012) proposed a probabilistic strategy for exploring the state-space of a web application. Wang et al. (2009) proposed a combinatorial testing approach to build navigational graphs for dynamic web applications. The approach targets the problem of providing a set of input values required for testing dynamic web applications. Thummalapenta et al. (2013) proposed a guided test generation to automatically generate tests against the specified business rules. Qi et al. (2019) proposed a keyword-guided approach called keyjaxTest to explore the state-space of a dynamic web application by using similarity matching algorithms. Liu et al. (2020) proposed a guided approach for exploring dynamic web applications against specific scenarios allowing the tester to provide input data at runtime during the crawl process.

The motivation behind guided approaches is to explore the application against specific scenarios or goals, which require prior knowledge of the application. For example, keyjaxTest (Qi et al., 2019) requires the keywords that describe the target functionality in the application. If the tester is not familiar with the application, then it is difficult to guess those keywords. Similarly, FeedEx (Fard and Mesbah, 2013) requires prior knowledge of the application to set up values manually for the code coverage, DOM diversity and navigational diversity in the proposed algorithm. These parameters can vary from one application to another that requires the tester to give the algorithm several runs before reaching an optimal combination of these parameters.

The closest in comparison to our approach is WebExplor (Zheng et al., 2021), which uses curiosity driven reinforcement learning approach to explore the state space of a web application. However, there are some key differences in QExplore and WebExplor. The state representation in WebExplor is based on the structure of HTML page whereas QExplore represents each state in terms of a set of interactable elements. The problem with structural representation of the state in dynamic web applications is that different business logic can have the same structural representation restricting the exploration strategy to consider DOM states with different business logic as one state. Another difference is that the reward function of WebExplor does not consider the potential of a DOM state that can open-up several new DOM states. This means that WebExplor assigns the same weight to two DOM states having different number of interactable elements. However, a DOM state with greater number of interactable elements has the potential to lead to new DOM states more than the one with less number of interactable elements. In contrast, QExplore considers the potential of a DOM state in terms of interactable elements in its reward function, thus, complementing the exploration process to capture new and diverse DOM states. Another major difference is the data input method used by both approaches. QExplore uses contextual data method that helps in exploring states behind the web forms whereas WebExplor uses random data limiting the exploration of data-driven states.

There are several other limitations common to both generic and guided approaches that QExplore alleviates. For example, existing approaches use absolute XPath or ID to locate the input field during crawling. However, dynamic web applications use dynamic IDs that change every time a webpage is refreshed. Similarly, an absolute XPath is also prone to breakage because a slight change in the DOM can make the absolute XPath unable to locate the target input field. In a part of the solution, QExplore provides user-specific inputs by locating the target input fields using user-defined keywords (e.g., user name) rather than XPath or ID. Consequently, existing approaches provide manual or random data to fill the web forms. QExplore provides contextual data automatically during exploration to pass validation constraints and explore states behind the web forms. Furthermore, existing approaches work in a regressive manner rather than incremental reducing the efficiency of the dynamic exploration approaches. QExplore works in an incremental manner which allows testers to interrupt and resume the exploration process in order to define user-specific inputs to uncover certain DOM states. This makes the QExplore more efficient than other existing dynamic approaches. Finally, QExplore allows testers to provide more than one value for the user-specific inputs (e.g., user name and password) which enables the discovery of DOM states available to different user roles.

Table 9 shows an overview of our approach in comparison to others. Our approach falls in the category of generic exploration approaches that require no or less prior knowledge from the tester to explore an unknown dynamic web application. Unlike other approaches, QExplore does not get stuck in dynamic regions of a web application. We focus on improving the crawling strategy by proposing a novel crawling strategy inspired by Q-learning to explore more states and executable paths under certain time constraints as compared to other existing dynamic exploration approaches. QExplore automatically feeds the input fields by using contextual data to uncover states behind the web forms and improve the functional coverage. Moreover, QExplore performs exploration on the basis of interactive DOM elements and therefore works for web applications having dynamic features either because of client-side languages (e.g., JavaScript) or server-side scripting languages such as PHP or JSP. Unlike existing approaches, QExplore identifies DOM elements by using keywords instead of application-specific locators such as XPath or ID.

Besides the works presented above, there are several studies that apply reinforcement learning to other areas of software testing. For example, Böttinger et al. (2018) propose an RL fuzzer for traditional softwares to learn seeding mutations that receive higher reward. Zheng et al. (2019) uses RL with multi-objective optimization for game testing. There are several studies on applying RL to the testing of mobile applications (Adamo et al., 2018; Koroglu et al., 2018; Pan et al., 2020). However, these areas are different than web applications testing, which has a different set of challenges than the other areas. Thus, our approach is fundamentally different than the other RL approaches and is applied to a different domain, i.e., exploration of dynamic web applications.

9. Conclusion & future work

The most commonly used technique to automate web application testing and support web analysis is dynamic exploration. In this study, we propose a guided exploration strategy for dynamic web applications inspired from Q-learning. The approach uses a contextual data input method to feed the input fields during the exploration and discover DOM states behind the web forms. The approach uses a novel definition of DOM action and DOM state to overcome the limitations of the previous approaches and support the exploration of dynamic web applications. Unlike existing dynamic approaches, our approach is incremental in nature and allows testers to define crawl rules by using keywords. We implemented our approach in a tool called QExplore which is evaluated on seven commonly used open-source subject applications and one industrial application called Flex. The results show that QExplore is effective than state-of-the-art Crawljax and WebExplor by achieving better code coverage, DOM diversity, navigational coverage, navigation diversity and discovering distinct DOM states. Similarly, QExplore discovered more error states in our target subject applications than WebExplor and Crawljax. We also measured the impact of contextual data input on the exploration of dynamic web applications by comparing the results from QExplore (with contextual data input method) with Crawljax, WebExplor and QExplore (with random data). It is evident that QExplore with contextual data input is more effective in exploring dynamic web applications followed by QExplore with random data, WebExplor and Crawljax. In the future, we will further improve the reward function to support scenarios-based crawling of dynamic web applications. Refining the reward function will add features of the guided exploration approach to QExplore. For example, getting specific keywords as an input from the user to feed the reward function and guide the crawling process towards exploring specific scenarios. We also plan to use reinforcement learning techniques such as DQN to incorporate learning into the approach during exploration. Moreover, we will continue to conduct more experiments with different types of dynamic web applications to further evaluate the effectiveness of exploration approach and contextual data input method.

Table 9
An overview of the existing exploration approaches.

	Thummalapenta et al. (2013)	Panthi and Mohapatra (2017)	Wang et al. (2009)	Dincturk et al. (2012)	Benedikt et al. (2002)	Fard and Mesbah (2013)	Liu et al. (2020)	Qi et al. (2019)	Mesbah et al. (2008)	Zheng et al. (2021)	QExplore
Year	2013	2017	2009	2012	2002	2013	2020	2019	2008	2021	2021
Crawling strategy	DFS	Model - based	DFS	Probabilistic	Random	DFS or BFS	DFS	Random Search	DFS or BFS	RL	Q-Learning
Generic or guided	Guided	Guided	Guided	Guided	Generic	Guided	Guided	Guided	Generic	Generic	Generic
Input data	Random	Random	Random	Random	Random	Random	Random	Random	Random	Random	Contextual
Regressive or incremental	Regressive	Regressive	Regressive	Regressive	Regressive	Regressive	Incremental	Regressive	Regressive	Regressive	Incremental
Server side or client side	Client side	Server side	Server side	Server side	Client side	Client side	Both	Both	Client side	Client side	Both
Static or dynamic	Static	Static	Static	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Single or multiple user-specific inputs	Single	Single	Single	Single	Single	Single	Multiple	Single	Single	Single	Multiple
Identification of target input fields	Absolute XPath	N/A	Absolute XPath	Absolute XPath	Absolute XPath	Absolute XPath	XPath	Keywords	Absolute XPath	XPath	Multiple
Require source code access	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
Tool	WATEG	N/A	Tansuo	N/A	VeriWeb	FeedEx	GUIDE	KeyjaxTest	Crawljax	WebExplor	QExplore

CRediT authorship contribution statement

Salman Sherin: Conceptualization, Methodology, Software, Validation, Investigation, Writing – original draft, Visualization. **Asmar Muqeet:** Software, Validation, Investigation, Writing – review & editing. **Muhammad Uzair Khan:** Resources, Writing – review & editing, Supervision, Funding acquisition. **Muhammad Zohaib Iqbal:** Resources, Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Adamo, D., et al., 2018. Reinforcement learning for android gui testing. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.
Augsten, M.P.N., 2016. Tree edit distance: Robust and memory-efficient. Inf. Syst. 56, 157–173.
Belshe, M., Peon, R., Thomson, M., 2015. Hypertext transfer protocol version 2 (HTTP/2). RFC 7540.
Benedikt, M., Freire, J., Godefroid, P., 2002. VeriWeb: Automatically testing dynamic web sites. In: Proceedings of 11th International World Wide Web Conference (WW W'2002). Citeseer.
Beroual, O., Guérin, F., Hallé, S., 2017. Searching for behavioural bugs with stateful test oracles in web crawlers. In: 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing. SBST, IEEE.
Biagiola, M., Ricca, F., Tonella, P., 2017. Search based path and input data generation for web application testing. In: International Symposium on Search Based Software Engineering. Springer.
Biagiola, M., et al., 2020. Dependency-aware web test generation. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification. ICST, IEEE.
Bojanowski, P., et al., 2017. Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. 5, 135–146.
Böttinger, K., Godefroid, P., Singh, R., 2018. Deep reinforcement fuzzing. In: 2018 IEEE Security and Privacy Workshops. SPW, IEEE.
Bozkurt, M., Harman, M., 2011. Automatically generating realistic test input from web services. In: Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System. SOSE, IEEE.
Brunelle, J.F., et al., 2016. The impact of JavaScript on archivability. Int. J. Digit. Libr. 17 (2), 95–117.
Choudhary, S.R., et al., 2011. Water: Web application test repair. In: Proceedings of the First International Workshop on End-to-End Test Script Engineering.
Dincturk, M.E., et al., 2012. A statistical approach for efficient crawling of rich internet applications. In: International Conference on Web Engineering. Springer.
Doğan, S., Betin-Can, A., Garousi, V., 2014. Web application testing: A systematic literature review. J. Syst. Softw. 91, 174–201.
Duda, C., et al., 2009. Ajax crawl: Making ajax applications searchable. In: 2009 IEEE 25th International Conference on Data Engineering. IEEE.
Faheem, M., 2012. Intelligent crawling of Web applications for Web archiving. In: Proceedings of the 21st International Conference on World Wide Web.
Fard, A.M., Mesbah, A., 2013. Feedback-directed exploration of web applications to derive test models. In: ISSRE.
Ferrucci, F., et al., 2011. A crawljax based approach to exploit traditional accessibility evaluation tools for AJAX applications. In: Information Technology and Innovation Trends in Organizations. Springer, pp. 255–262.
Fraser, G., Arcuri, A., 2012. Sound empirical evidence in software testing. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE.
Gao, K., Wang, W., Gao, S., 2013. Modelling on web dynamic incremental crawling and information processing. In: 2013 5th International Conference on Modelling, Identification and Control. ICMIC, IEEE.
Groeneveld, F., Mesbah, A., Van Deursen, A., 2010. Automatic invariant detection in dynamic web applications. Technical Report Series TUD-SERG-2010-037.
Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Pearson Education.
Imtiaz, J., Iqbal, M.Z., 2021. An automated model-based approach to repair test suites of evolving web applications. J. Syst. Softw. 171, 110841.
Koroglu, Y., et al., 2018. QBE: Qlearning-based exploration of android applications. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation. ICST, IEEE.
Li, Y.-F., Das, P.K., Dowe, D.L., 2014. Two decades of web application testing—A survey of recent advances. Inf. Syst. 43, 20–54.
Liu, C.-H., Chen, W.-K., Sun, C.-C., 2020. GUIDE: an interactive and incremental approach for crawling web applications. J. Supercomput. 76 (3), 1562–1584.
Liu, L., et al., 2011. Automatic generation of assertions from system level design using data mining. In: Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011). IEEE.
Maras, J., et al., 2013. Identifying code of individual features in client-side web applications. IEEE Trans. Softw. Eng. 39 (12), 1680–1697.
Marchetto, A., Tonella, P., Ricca, F., 2008. State-based testing of Ajax web applications. In: 2008 1st International Conference on Software Testing, Verification, and Validation. IEEE.
Mariani, L., et al., 2011. AutoBlackTest: a tool for automatic black-box testing. In: 2011 33rd International Conference on Software Engineering. ICSE, IEEE.
Mesbah, A., Bozdag, E., Van Deursen, A., 2008. Crawling Ajax by inferring user interface state changes. In: 2008 Eighth International Conference on Web Engineering. IEEE.
Mesbah, A., Prasad, M.R., 2011. Automated cross-browser compatibility testing. In: Proceedings of the 33rd International Conference on Software Engineering.
Mesbah, A., Van Deursen, A., 2008. Exposing the hidden-web induced by ajax. Technical Report Series TUD-SERG-2008-001.
Mesbah, A., Van Deursen, A., Roest, D., 2011. Invariant-based automatic testing of modern web applications. IEEE Trans. Softw. Eng. 38 (1), 35–53.
Mikolov, T., et al., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
Mirshokraie, S., Mesbah, A., 2012. JSART: JavaScript assertion-based regression testing. In: International Conference on Web Engineering. Springer.
Mirzaaghaei, M., Mesbah, A., 2014. DOM-based test adequacy criteria for web applications. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis.
Moosavi Byooki, S.A., 2014. Component-Based Crawling of Complex Rich Internet Applications. Université d'Ottawa/University of Ottawa.
Muñoz, F.R., Cortes, I.I.S., Villalba, L.J.G., 2018. Enlargement of vulnerable web applications for testing. J. Supercomput. 74 (12), 6598–6617.
Ocariza Jr., F.S., Pattabiraman, K., Mesbah, A., 2012. AutoFLox: An automatic fault localizer for client-side JavaScript. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE.
Olston, C., Najork, M., 2010. Web Crawling. Now Publishers Inc.
Pan, M., et al., 2020. Reinforcement learning based curiosity-driven testing of Android applications. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.
Panthi, V., Mohapatra, D.P., 2017. An approach for dynamic web application testing using MBT. Int. J. Syst. Assur. Eng. Manag. 8 (2), 1704–1716.
Park, J.H., et al., 2017. Novel assessment method for accessing private data in social network security services. J. Supercomput. 73 (7), 3307–3325.
Pawlik, M., Augsten, N., 2011. RTED: a robust algorithm for the tree edit distance. arXiv preprint arXiv:1201.0230.
Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).
Qi, X.-F., et al., 2019. Leveraging keyword-guided exploration to build test models for web applications. Inf. Softw. Technol. 111, 110–119.
Sherin, S., et al., 2021. Comparing coverage criteria for dynamic web application: An empirical evaluation. Comput. Stand. Interfaces 73, 103467.
Stocco, A., Yandrapally, R., Mesbah, A., 2018. 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.
Tai, K.-C., 1979. The tree-to-tree correction problem. J. ACM 26 (3), 422–433.
Tanida, H., et al., 2011. Automated system testing of dynamic web applications. In: International Conference on Software and Data Technologies. Springer.
Thummalapenta, S., et al., 2013. Guided test generation for web applications. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE.
Van Deursen, A., Mesbah, A., Nederlof, A., 2015. Crawl-based analysis of web applications: Prospects and challenges. Sci. Comput. Program. 97, 173–180.
Wang, W., et al., 2009. A combinatorial approach to building navigation graphs for dynamic web applications. In: 2009 IEEE International Conference on Software Maintenance. IEEE.
Watkins, C.J.C.H., 1989. Learning from delayed rewards.
Yandrapally, R., Stocco, A., Mesbah, A., 2020. Near-duplicate detection in web app model inference. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.
Ye, M., Li, G., 2017. Internet big data and capital markets: a literature review. Financial Innov. 3 (1), 1–18.
Zheng, Y.L., Yi, Xie, Xiaofei, Liu, Yepang, Ma, Lei, Hao, Jianye, Liu, Yang, 2021. Automatic web testing using curiosity-driven reinforcement learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE.

Zheng, Y., et al., 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE.

Zou, Y., et al., 2014. Virtual DOM coverage for effective testing of dynamic web applications. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis.

Salman Sherin is a Ph.D. scholar at Software Quality Engineering and Testing (QUEST) Laboratory, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He received his MS SE degree in 2017. His research interests include Web application testing, machine learning, evidence-based software engineering and mutation analysis.

Asmar Muqeet is a research engineer at Software Quality Engineering and Testing (Quest) Laboratory. He is doing his MS SE degree from National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. His research areas include Machine Learning, Model-based Testing, End-to-End Test Automation and Testing Cockpit Display Systems.

Muhammad Uzair Khan is currently an Assistant Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He is heading the Software Quality Engineering and Testing (QUEST) Laboratory, and is a founding member of Pakistan Software Testing Board. He completed his Ph.D. research work at INRIA, France and received his Ph.D. degree in computer science from University of Nice, France in 2011. His research interests include model-driven engineering, empirical software engineering, aspect-oriented software engineering, real time operating systems, and engineering dependable software systems.

Muhammad Zohaib Iqbal is currently an Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He is also the chief scientist at Software Quality Engineering and Testing (QUEST) Laboratory and President of Pakistan Software Testing Board. He received his Ph.D. degree in software engineering from University of Oslo, Norway in 2012. Before joining Fast-NU, he was a research fellow at Simula Research Laboratory, Norway. His research interests include model-driven engineering, engineering mission & safety critical systems, software testing, and dependable avionics systems.