Using Webpage Comparison Method for Automated Web Application Testing with Reinforcement Learning

Ci-Feng Lai, Chien-Hung Liu, Shingchern D. You*

Department of Computer Science and Information Engineering, National Taipei University of Technology, Taiwan, ROC

Received 08 August 2024; received in revised form 01 November 2024; accepted 02 November 2024

DOI: https://doi.org/10.46604/ijeti.2024.14104

Abstract

Web application testing often uses crawlers to explore the application under test (AUT) and identify potential vulnerabilities. For dynamically generated pages, crawlers must provide test inputs for web forms. A previous tool combines a web crawler with a reinforcement learning agent, which uses code coverage to guide the crawler in filling web forms. This paper aims to improve the applicability of web application testing by using webpage comparison techniques instead of code coverage and source code access, thereby enhancing the handling of multiple web forms on a single page. Experimental results show that this approach explores more pages, reaches greater crawling depths, and achieves better code coverage than the original method. It also interacts more efficiently with multiple web forms and outperforms a random-action Monkey on new, untrained web applications. Therefore, this approach is promising for automated web application testing.

Keywords: automated testing, web crawler, reinforcement learning, webpage comparison

1. Introduction

Due to the rapid development of globalization, many services are now offered as web applications. According to statistics from Siteefy, as of 2023, there are already 1.11 billion websites [1], and approximately 71% of businesses worldwide have a website [2]. With the sharp increase in the number of websites, the importance of web testing has become increasingly significant. Web testing ensures the normal operation and user experience of websites and conduces to the enhancement of business competitiveness.

Web application testing is generally divided into manual testing and automated testing. Manual testing requires testers to personally operate the web application, which demands that testers possess extensive software testing knowledge and an understanding of the web application being tested. The advantage of manual testing is that it enables the operations to test specific scenarios, capturing detailed issues that automated testing might overlook. For example, the operator can more accurately determine the optimal input values for web form fields. However, the disadvantages of manual testing are also evident, viz., the testing process is time-consuming and costly due to the need for extensive manual operations. Not only that, it explicitly and observably manifests when the web application is tremendous, complex, and arduous.

In contrast, automated testing uses web crawlers to automatically crawl and explore web applications, generating visual state diagrams for testers to analyze and troubleshoot issues. The advantage of automated testing is that it significantly improves testing efficiency and reduces the workload of manual operations. This is particularly important for large-scale regression testing or frequently updated web applications.

[*] Corresponding author. E-mail address: scyou@ntut.edu.tw

English language proofreader: Chih-Wei Chang

The crawl-based approach has been utilized for manifold types of web testing, including regression, compatibility, and security [3]. This method employs a crawler, such as Crawljax [4], to examine a web application, interact with user interface elements, and model potential user actions to verify the application's features [5]. While the crawl-based approach can automatically test web applications, called application under test (AUT), the crawler requires input values to fill form fields in dynamic web applications. These input values are often prepared manually or generated randomly, which can lead to inefficiencies and high costs. This poses a challenge for automated testing.

The extent to which the AUT is tested is typically measured by code coverage. According to Brader et al. [6], it is stated that coverage means some of the logic of the code has not been tested. Nevertheless, high coverage indicates that the likelihood of correct processing is rather decent. Therefore, analogously, higher coverage is desirable for software testing. Statement and branch coverage are two common methods for measuring code coverage, which indicate the percentage of statements or branches executed in the AUT. A branch is a decision point in the code, and branch coverage is more comprehensive, subsuming statement coverage. However, finding and selecting input values for a crawler to achieve higher code coverage of the AUT remains a challenging problem.

To address this issue, Liu et al. [7] previously proposed a collaborative framework that integrates a crawler and an agent working together. The crawler explores the AUT and identifies pages requiring form inputs during the crawling process. These identified pages are subsequently handed over to the agent for interaction. The agent, aiming to discover more usage scenarios, selects actions to determine the input values for the form fields. If the form filling is completed, the agent generates a set of directives guiding the crawler to continue its exploration. This process repeats until the crawler confirms the absence of input pages or a timeout occurred. A reinforcement learning (RL) algorithm was used to train the agent, incorporating code coverage as a component of the rewards, yielding promising results.

Based on the aforementioned finding, the previously proposed framework by Liu et al. [7] can explore pages more thoroughly, whereas limitations emerge:

(1) It relies on code coverage as the reward criterion, which depends on the language of the tool. For instance, Istanbul-middleware [8] can measure the code coverage for ES5 and ES2015+ JavaScript codes, but not for other languages like PHP or Python. To compute the coverage of PHP code, different tools must be used. This situation complicates the integration of the testing platform.
(2) It interacts with all the form tags on a page, even if there are multiple forms.
(3) It trains the agent to test only the same AUT, without studying its generalization capability.

To overcome these challenges, this paper improves the previous framework in the following ways:

(1) It proposes an alternative approach that uses webpage comparison techniques [9-10] to derive new metrics—number of input pages, input page depth, and input page breadth increase—instead of code coverage vectors. This enables AUTs to be tested without the reliance on code coverage. Since tools for generating code coverage are language-dependent, the proposed technique standardizes the performance metrics, rendering them independent of the programming language used.
(2) It adds a mechanism to the framework that checks if multiple form tags on a webpage are present, and if so, it interacts with one form at a time. Such a mechanism improves the efficiency of exploration, which is therefore regarded as a necessary step for the agent to learn.
(3) It modifies the reward function in the RL algorithm, enabling the agent to be trained with single or multiple web applications and testing a different web application without further training. This capability is particularly useful for testing new web applications as retraining can be skipped. In contrast, different from the aforementioned advantage, a random-input approach (Monkey) does not possess this capability.

The rest of the paper is organized as follows. Section 2 covers a complete review of the related work. Section 3 presents the proposed approach. Section 4 describes and discusses the experiments and results. Finally, Section 5 is the conclusion and future work.

2. Related Work

Lin et al. [11] suggested a way to test web applications using natural language. The method takes the features of a Document Object Model (DOM) element and its nearby words and makes a vector. The vector is changed to a vector of real numbers using some natural language methods, e.g., bag-of-words. The method subsequently compares the vector with a training set to find a topic for the DOM element. Based on the topic, the method picks a value for the element from a databank. The tests show that the method works as well or better than the old rule-based methods.

Carino and Andrews [12] suggested a way to test application graphical user interfaces (GUIs) using ant colony optimization (ACO). The way uses an ant colony algorithm with Q-learning, called AntQ. Specifically, it can induce event sequences to go through the GUIs and use the GUI state changes from the events as the goal. The tests show that AntQ outperforms random testing and normal ant colony algorithms in finding statements and faults.

Kim et al. [13] suggested a way to use RL instead of human-made algorithms in the search-based software testing (SBST) method. The SBST algorithms try to find the best test data based on the fitness function. The researchers made a test data problem as an RL environment and trained an RL agent with double deep Q-networks (DDQN). Given that the fitness value represents the reward of the agent, when the agent generates a new solution to obtain more reward, the fitness value of the solution is lower. The tests exhibit that the way works and can cover all branches for the C functions.

Liu et al. [14] proposed an interactive crawling approach and a crawler called GUIDE to test web applications using user directives. GUIDE actively inquires the user for directives to inspect web pages when an input field is detected. The tests show that GUIDE can identify more code coverage than the previous web crawler. However, GUIDE still requires human input. This work tries to use RL to train an agent to give inputs and assist web crawlers in the search for more code.

Zheng et al. [15] suggested a way to test web applications automatically. The way is called WebExplor and it can make different actions to find new web pages. It uses a reward function and a deterministic finite automaton (DFA) to guide the exploration. The DFA is a machine that records the visit information. If WebExplor cannot find a new state in some time, it selects a path from the DFA and keeps exploring. The tests reveal that WebExplor can find more faults, code, and work more efficiently than the new techniques. Although this approach employs Q-learning (a type of RL algorithm), the Q-learning is trained based on the states of the AUT. Consequently, this method cannot generalize knowledge from one AUT to another. In contrast, the approach presented in this paper utilizes the RL algorithm in a manner that allows the knowledge gained from testing one AUT to be transferred to another AUT.

Liu et al. [16] proposed an RL approach for workflow-guided exploration. The approach aims to alleviate the overfitting problem when training an RL agent to perform web-based tasks, such as booking flights, by mimicking expert demonstrations. Particularly, the approach includes high-level workflows that can limit the allowable actions at each time step by pruning those bad exploration directions. This empowers the agent to discover sparse rewards faster while avoiding overfitting. The experimental results show that the proposed approach can achieve a higher success rate and significantly improve sample efficiency compared to existing methods.

Sherin et al. [17] suggested using Q-learning for guided checking. This method can check dynamic web applications without a cornucopia of knowledge. The tests show that QExplore outperforms Crawljax and WebExplor in finding more code and pages. Analogous to the approach by Zheng et al., this method cannot extend the knowledge gained from training on one AUT to another. This limitation is thereby a key difference between their approach and the work presented in this paper.

Wang and Tian [18] proposed an efficient method for the automatic generation of linearly independent paths in white-box testing. This approach transforms the source code into a strongly connected graph and subsequently applies an algorithm to find the paths. However, notably, the approach studied in this paper requires no specific analysis of source code during testing. From the above review, it is clear that the presented approach is unique in that the agent is trained on one (or more) AUTs to test other AUTs. However, this capability allows the presented approach to test new AUTs faster than other methods.

3. Proposed Approach

Since the proposed approach modifies a previously available framework, this section outlines the entire framework, highlighting the modified parts. The first subsection details user scenario exploration, which is particularly rudimentary for understanding the framework. The next subsection provides an overview of the framework. This is followed by a description of the webpage comparison method. The subsequent subsection addresses the issue of multiple forms on a webpage. The final subsection discusses the design of RL rewards and the agent.

3.1. Exploring user scenarios

Web applications are typically designed to empower users to perform interactive tasks, such as making queries, manipulating data, or completing transactions. A user scenario of a web application usually involves a sequence of user actions, such as clicking on a hyperlink or filling out a web form, to complete a task. To verify the functionality of such a web application, the user scenarios need to be explored so that the corresponding code can be executed and tested.

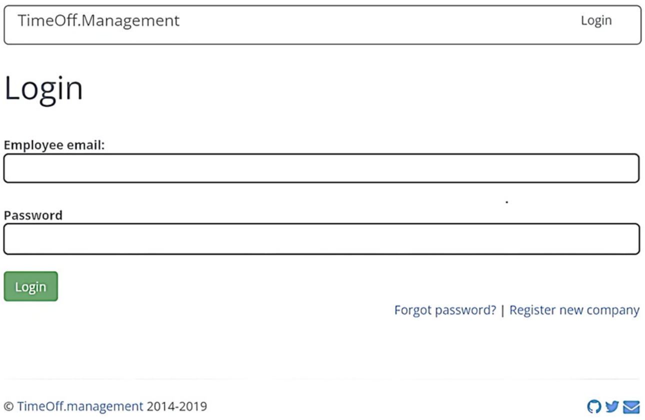

TimeOff.Management
Login
Login
Employee email:
Password
Login
Forgot password? | Register new company
© TimeOff.management 2014-2019

Fig. 1 The login page of TimeOff.Management

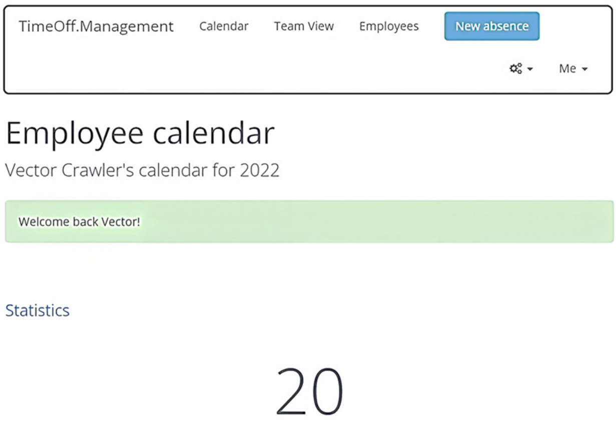
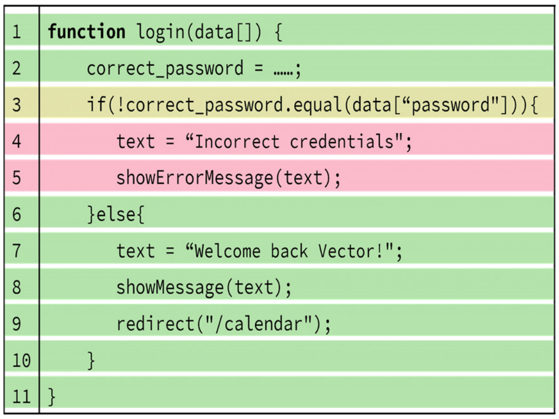


Fig. 2 The login success page of TimeOff.Management

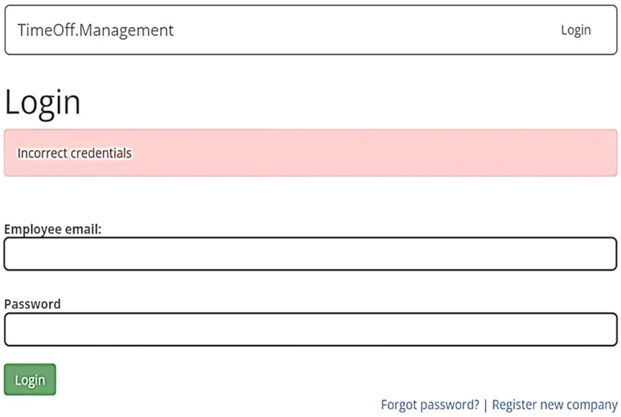


```
function login(data[]) {
    correct_password = ……;
    if(!correct_password.equal(data[“password"])){
      text = “Incorrect credentials";
      showErrorMessage(text);
    }else{
      text = “Welcome back Vector!";
      showMessage(text);
      redirect("/calendar");
    }
}
```

Fig. 3 The source code of a login success

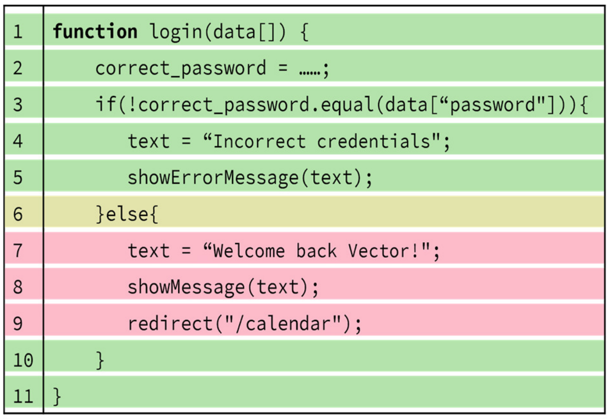
To illustrate this concept, consider an open-source web application for employee absence management called TimeOff.Management [19]. In Fig. 1, the application initially displays the login page. In a successful login scenario, the user needs to enter the correct username and password, after which the application will display the calendar page, as shown in Fig.

2. The source code covered for this scenario is displayed in Fig. 3, where the green-highlighted sections indicate the code covered once a login is successful. Conversely, in a login failure scenario, the user enters an incorrect username or password, and the application remains on the login page with an error message, as shown in Fig. 4. The corresponding source code for this scenario is shown in Fig. 5 with the green-highlighted sections indicating the code covered when the user fails to login.



Incorrect credentials

Fig. 4 The login fail page of TimeOff.Management



```
function login(data[]) {
    correct_password = ……;
    if(!correct_password.equal(data[“password"])){
        text = “Incorrect credentials";
        showErrorMessage(text);
    }else{
        text = “Welcome back Vector!";
        showMessage(text);
        redirect("/calendar");
    }
}
```

Fig. 5 The source code of a login-fail

3.2. Overview of USAGI

The implemented system is a modification of the USAGI framework, which stands for Using Agents to Automatically Choose Input Data [20]. It is worth noting that the study by Liu et al. [7] is also based on the USAGI framework. Regarding completeness, the following section briefly describes the design of the entire system and then discusses the modified parts. USAGI consists of a crawler and an agent. Since the crawler and the agent are separate programs, a job pool, referred to as the learning pool, is employed to facilitate their interaction and communication, as illustrated in Fig. 6.

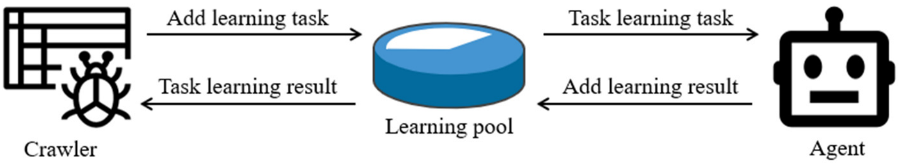


Fig. 6 The USAGI framework

USAGI systematically crawls an AUT using a web crawler. By providing the URL of a page (e.g., the home page of the AUT), the crawler automatically requests the page, interacts with it through clicking or typing, and continuously explores the subsequent linked pages. When encountering a page containing a web form (referred to as an input page), the crawler is required to fill the form with inputs that are either manually prepared or randomly generated owing to the mechanism. However, manually preparing inputs is labor-intensive and time-consuming, while randomly generating inputs often results in unsuccessful form submissions.

To automate form filling when the crawler encounters an input page, USAGI transfers the input page along with pertinent information (called a learning task) to an RL agent via a learning pool. Subsequently, the agent learns to perform a sequence of actions to complete and submit the form. The sequence of agent actions (called the learning result) is returned to the crawler afterward. This empowers the crawler to use these actions to interact with the input page, complete the form submission, and continue exploring, to improve the code coverage of the AUT. To further delineate, a detailed description of the agent is given in Section 3.5.

USAGI determines whether the action sequence found by the agent directs to a previously unexplored page by using the code coverage information of the AUT. Specifically, as mentioned in Section 3.1, provided that the code coverage of the AUT increases by performing the action sequence, USAGI considers that the action sequence can lead to a page not previously explored within a user scenario. In such cases, USAGI converts the action sequence found by the agent into a directive for guiding the crawler's behavior pertinent to input page exploration. This directive is submitted from the agent to the crawler to construct a directive tree. The directives on each path of the tree form a sequence of actions that enable the crawler to navigate from the home page to different target pages, thereby continuing the exploration.

Fig. 7 illustrates an example of a partial directive tree, comprising a root, directives, and input pages. The root is a dummy node that does not contain any information and only links to the input pages that the crawler initially explored without filling any web forms. Meanwhile, an input page node can have several directive child nodes, which include the corresponding hyperlink-clicking actions performed automatically by the crawler and the form-filling actions identified by the agent when interacting with the input page to explore possible user scenarios.

For instance, the login input page has two child nodes corresponding to the directives for login success and login failure scenarios. To continuously explore possible user scenarios after an input page has been previously crawled, the directives on the directive tree path are used by the crawler to navigate to the target page and resume crawling. Thereafter, to continue exploring the pages after a successful login, the directive of the child node (click register link) for the login input page will be used by the crawler to navigate to the register input page.

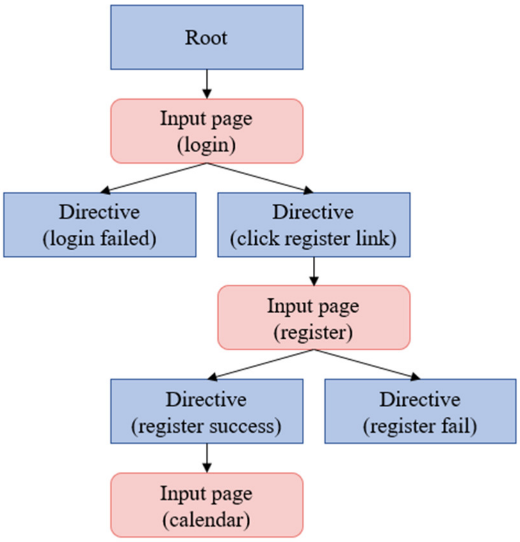


Fig. 7 An example of a partial directive tree

The process of using USAGI can be divided into two stages: model training and model testing. During the model training stage, input pages that were successfully explored earlier, along with their associated input data, are collected to form a training set. This training set is employed to create the RL environment for training a new agent. In the model testing stage, the agent trained in the previous stage is utilized to explore input pages while collecting new training data. These two stages are alternated to enhance the agent's ability to choose actions for input pages, thereby improving the code coverage of the AUT. Initially, USAGI employs a monkey to explore and interact with input pages while gathering the training data set.

3.3. Directing crawler's exploration with the changes in DOM structure

The code coverage of the AUT was used to assess whether the agent's actions completed form submissions [20]. Despite the straightforwardness of using code coverage to map scenarios, collecting this data can be arduous and cumbersome, especially when AUTs are developed with programming languages where code coverage information is not readily available. This limitation can also restrict the applicability of the approach. Therefore, in this study, a webpage comparison method is proposed to determine whether a sequence of agent actions can effectively fill and submit the form on an input page. The proposed webpage comparison method involves using static analysis to obtain information about the DOM [21] of a webpage.

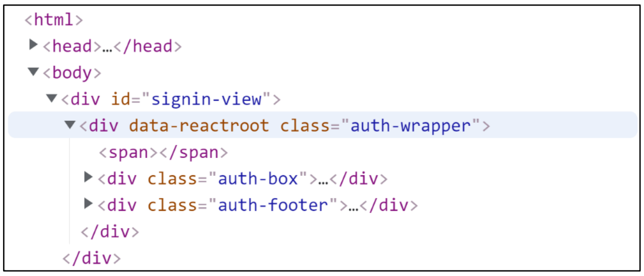
By comparing the DOM differences of the pages before and after submitting a form, the method can assess once the form submission is successful. If so, it is assumed that more code coverage is obtained. To discuss further, the following sections describe the proposed approach, including changes in the web page's DOM structure and the webpage comparison algorithm.

3.3.1. Changes in DOM structure of AUT

The DOM of a web page is typically comprised of HTML, CSS, and JavaScript code, and the browser renders a page according to its DOM structure. When a user interacts with a page, the DOM structure of the AUT is often modified or replaced entirely to visually reflect the results of the interaction. The agent assessed the effectiveness of interactions by utilizing server-side code coverage of AUT [20]. However, this information is not completely available. Thus, instead of using code coverage, this study utilizes the changes in AUT's DOM structure to assess the effectiveness of interactions.

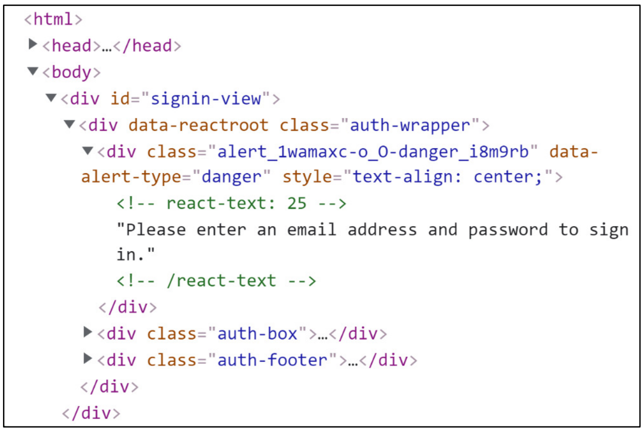
When server-side code is executed, the DOM structure of the AUT is usually changed or replaced to visually display the execution results. Therefore, when a certain user scenario is explored by the crawler, not only the server-side code can be covered, but the DOM structure of the AUT will also be altered or replaced by a new DOM. To illustrate, consider the login page example in Section 3.1: login success and login failure result in different DOMs and thus different visual appearances, as shown in Figs. 2 and 4. However, the DOM structure can also be changed by client-side JavaScript even if no server-side code is executed. Nevertheless, form submission is typically handled by server-side business logic. Therefore, if the DOM structure of an input page is significantly changed or entirely replaced, such changes can be attributed to the execution of server-side code, indicating an increase in server-side code coverage. Consequently, the focus is on detecting changes in the DOM structure. This is achieved by comparing the contents of the DOM before and after form submission.

To better understand the correlation among the DOM structure, the exploration of user scenarios, and the server-side code coverage of the AUT, the DOM structure of the login page in KeystoneJS [22] serves as an example. Fig. 8 depicts the DOM structure of the login page, while Fig. 9 displays the modified DOM structure when the crawler explores the login failure scenario. By comparing these two figures, it is evident that Fig. 9 replaces the <span> </span> in Fig. 8 with <div class="alert_1wamaxc-o_O-danger_i8m9rb" ...>...</div>. This confirms that changes in the AUT's DOM structure can be used to determine if a new part of a user scenario has been explored and, consequently, if new server-side code has been covered.



```
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="signin-view">
     ▼<div data-reactroot class="auth-wrapper">
         <span></span>
       ▶<div class="auth-box">…</div>
       ▶<div class="auth-footer">…</div>
       </div>
     </div>
```

Fig. 8 The original DOM structure of the login page



```
<html>
 ▶<head>…</head>
 ▼<body>
   ▼<div id="signin-view">
     ▼<div data-reactroot class="auth-wrapper">
       ▼<div class="alert_1wamaxc-o_O-danger_i8m9rb" data-
       alert-type="danger" style="text-align: center;">
           <!-- react-text: 25 -->
           "Please enter an email address and password to sign
           in."
           <!-- /react-text -->
         </div>
       ▶<div class="auth-box">…</div>
       ▶<div class="auth-footer">…</div>
       </div>
     </div>
```

Fig. 9 The DOM structure of a login fail

3.3.2. Scenario-guided exploration

From the above descriptions, it is evident that a relationship emerges between the DOM structure, the exploration of user scenarios, and the server-side code coverage of the AUT. Therefore, this paper proposes using the results from a webpage comparison method, instead of code coverage, as the criterion to determine whether the action sequence performed by the agent can lead to new user scenarios. The proposed method entails using static analysis to obtain the webpage's DOM information, comparing the differences between the DOMs of the page before and after the interaction (i.e., executing agent actions), and determining whether the page after the interaction is in a new state (page).

Fig. 10 shows the block diagram of the proposed scenario-guided exploration. Initially, the crawler at the bottom center of Fig. 10 finds a new input page. To simplify the discussion, it is assumed to have only one form on the input page. Handling multiple forms on a page will be discussed in Section 3.4. The RL agent subsequently interacts with the input page, and a valid directive is obtained. Meanwhile, multiple user scenarios may involve the same input page. To prevent the agent from repeating the same user scenario, it is necessary to determine whether the action sequence (or directive) generated by the agent can lead to a new page explored by the crawler. If the new page has a similar DOM structure to one in the database collection, this directive will be discarded, and the agent will generate a new directive. Otherwise, it is added to the collection, and a new user scenario is considered detected. This process repeats until a predefined timestep is reached.

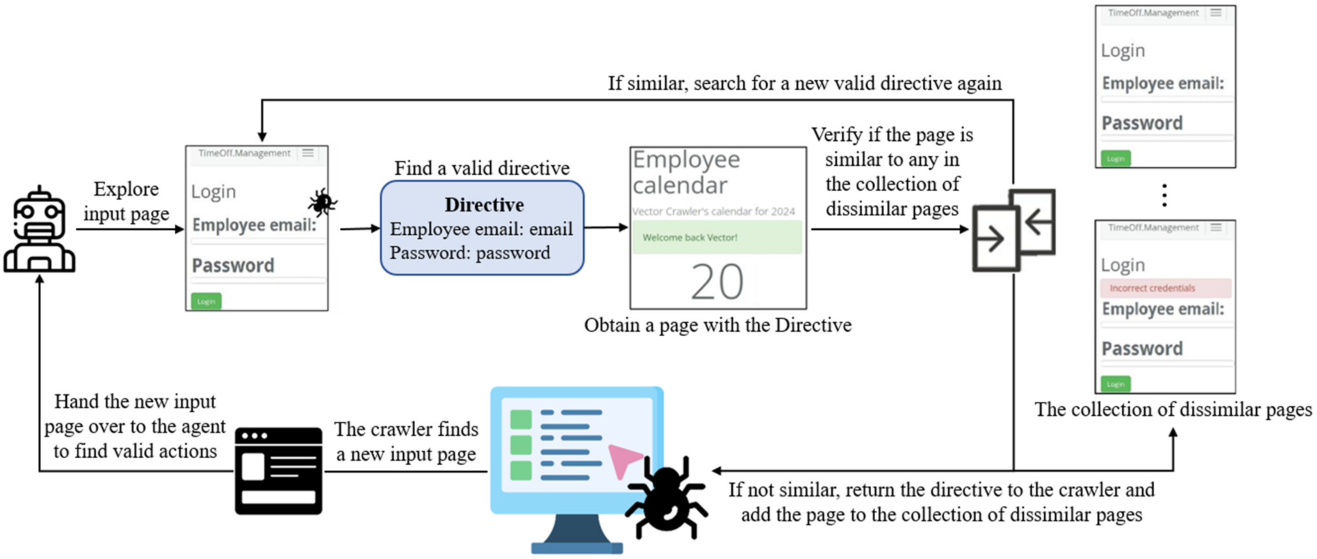

If similar, search for a new valid directive again
Explore input page
TimeOff.Management
Login
Employee email:
Password
Login
Find a valid directive
Directive
Employee email: email
Password: password
Employee calendar
Vector Crawler's calendar for 2024
Welcome back Vector!
20
Obtain a page with the Directive
Verify if the page is similar to any in the collection of dissimilar pages
Incorrect credentials
The collection of dissimilar pages
Hand the new input page over to the agent to find valid actions
The crawler finds a new input page
If not similar, return the directive to the crawler and add the page to the collection of dissimilar pages

Fig. 10 Block diagram of the proposed scenario-guided exploration

3.3.3. Proposed webpage comparison algorithm

The DOM consists of various elements (or components), each with different attributes. According to the methods proposed by Gowda and Mattmann [9] and Griazev and Ramanauskaitė [10], the similarity of webpages can be determined by comparing their DOM elements, including tags, structure, style, text content, etc. If two pages are similar, the actions conducted by Monkey are not collected and used for training (see also Section 3.5.1).

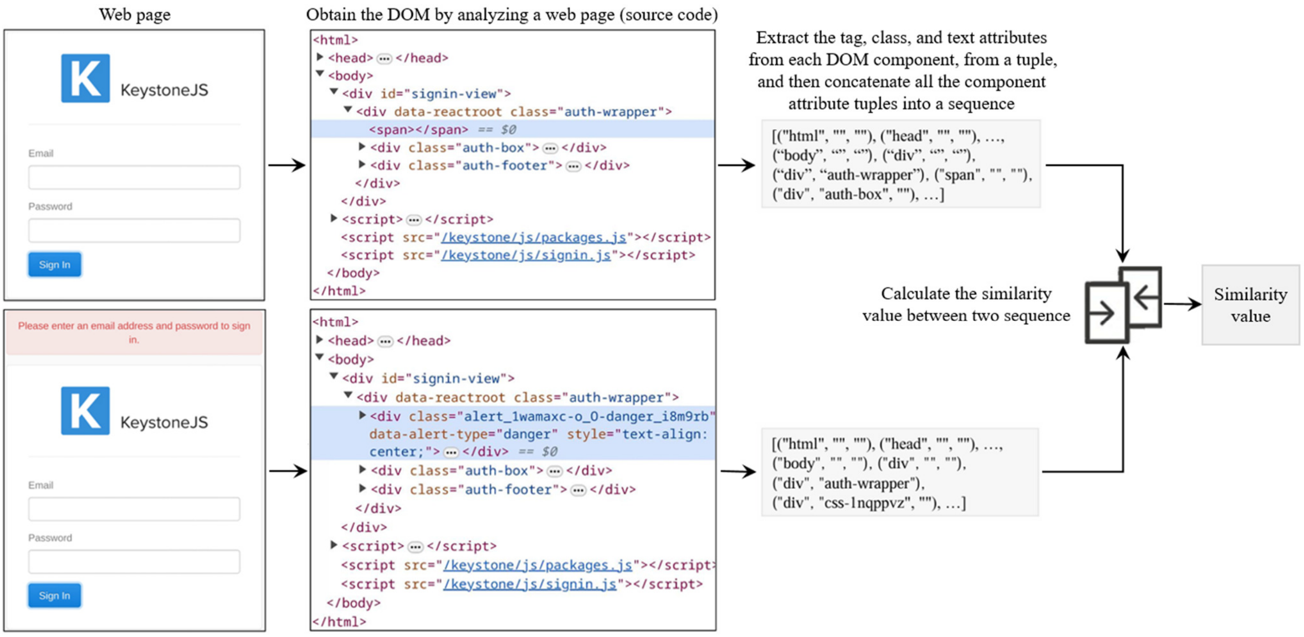
Among the DOM elements, tags, classes, and text, in particular, convey important information about a page. Meanwhile, comparison efficiency can be improved by excluding the remaining elements. Excluding these less important elements improves comparison efficiency. For instance, the original Page Compare tool [23] uses only the "tag" element, and its comparison performance is acceptable. In this paper, in addition to the tag element, class, and text elements are also used in comparisons. A preliminary experiment was conducted to determine the threshold. It was found that if two pages have a similarity of less than 0.95, they are not the same page, with an accuracy ranging from 84.2% to 95.1% based on human inspection.

The proposed similarity calculation method is illustrated in Fig. 11, using the login screen of KeystoneJS as an example. The webpage's screenshot and source code are also shown in the figure. The method involves analyzing the webpage to obtain the DOM information, extracting the tag, class, and text attributes for each DOM element, and forming a tuple. These element-attribute tuples are concatenated into a sequence thereafter.

By analyzing two web pages using this method, two sequences of element-attribute tuples are generated. A sequence comparison algorithm is subsequently used to calculate the similarity value, and a threshold is set to determine whether these two web pages are similar.

Based on the proposed method, an algorithm has been designed to determine whether a page obtained after applying an action sequence of the agent is similar to the existing ones explored by the crawler. The idea is to compare a page with all other pages, considering them similar if their similarity ratio exceeds a given threshold.

Initially, the crawler obtains the current page after applying the action sequence. Then, the webpage comparison method is applied to compare the DOM of the current page with stored pages to obtain the maximum similarity values among all pages. If the current page is not similar to any page in the collection, it is added to the dissimilar collection.


Web page
Obtain the DOM by analyzing a web page (source code)
KeystoneJS
Email
Password
Sign In
Please enter an email address and password to sign in.
KeystoneJS
Email
Password
Sign In
Extract the tag, class, and text attributes from each DOM component, from a tuple, and then concatenate all the component attribute tuples into a sequence
Calculate the similarity value between two sequence
Similarity value

```
<html>
 ▶ <head> ⋯ </head>
 ▼ <body>
   ▼ <div id="signin-view">
     ▼ <div data-reactroot class="auth-wrapper">
         <span></span> == $0
       ▶ <div class="auth-box"> ⋯ </div>
       ▶ <div class="auth-footer"> ⋯ </div>
       </div>
     </div>
   ▶ <script> ⋯ </script>
     <script src="/keystone/js/packages.js"></script>
     <script src="/keystone/js/signin.js"></script>
   </body>
</html>
```

[("html", "", ""), ("head", "", ""), …, (“body”, “”, “”), (“div”, “”, “”), (“div”, “auth-wrapper”), ("span", "", ""), ("div", "auth-box", ""), …]

```
<html>
 ▶ <head> ⋯ </head>
 ▼ <body>
   ▼ <div id="signin-view">
     ▼ <div data-reactroot class="auth-wrapper">
       ▶ <div class="alert_1wamaxc-o_O-danger_i8m9rb"
         data-alert-type="danger" style="text-align:
         center;"> ⋯ </div> == $0
       ▶ <div class="auth-box"> ⋯ </div>
       ▶ <div class="auth-footer"> ⋯ </div>
       </div>
     </div>
   ▶ <script> ⋯ </script>
     <script src="/keystone/js/packages.js"></script>
     <script src="/keystone/js/signin.js"></script>
   </body>
</html>
```

[("html", "", ""), ("head", "", ""), …, ("body", "", ""), ("div", "", ""), ("div", "auth-wrapper"), ("div", "css-1nqppvz", ""), …]

Fig. 11 Proposed similarity calculation method

Algorithm 1 presents the function calculatePagesSimilarity() used for the similarity calculation method. Its core operations are located between lines 7 and 17. In this code block, attributes such as tag, class, and text of all elements in the DOM of the entire page are extracted and organized into tuples. These tuples are subsequently concatenated into a sequence, as mentioned previously. After processing the sequences of DOM elements in the pages (lines 1-5), a sequence comparison algorithm is employed to calculate the similarity value between two distinct sequences. The function calculateSequenceSimilarity() represents this sequence-comparison algorithm.

The implementation of the calculatePagesSimilarity() function is based on modifications to the Page Compare project [23]. Given, that in the original method, only the tag attribute of DOM elements in a page was extracted, the extraction of the class and text attributes was added herein. The core of the calculateSequenceSimilarity() function is based on the SequenceMatcher class from Python’s built-in difflib library, which is used “for comparing pairs of sequences of any type, as long as the sequence elements are hashable” [24].

The implemented method can find the longest contiguous matching subsequence that excludes uninterested elements, such as blanks or whitespace. The similarity is then computed based on the length of this identified subsequence. The SequenceMatcher objects have many methods, such as set_seq1(a), set_seq2(b), and ratio(). The ratio() method “returns a measure of the sequences’ similarity as a float in the range [0,1]” [24]. With the SequenceMatcher class, it is easy to compute the similarity between two sequences. Indeed, the SequenceMatcher class was also used by the Page Compare project [23] to calculate the similarity between two different sequences.

```
Algorithm 1: calculatePagesSimilarity
Input: Page sourcePage, Page targetPage
Output: double similarity
begin
    sourceElements ← getElements(sourcePage)
    targetElemenets ← getElements(targetPage)
    return calculateSequenceSimilarity(sourceElements, targetElements)
end

procedure getElements(Page page)
begin
    dom ← page.getDOM()
    elements ← []
    for each element in dom do
        tag ← element.getTag()
        class ← element.getClass()
        text ← element.getText()
        elements.append({tag, class, text})
    end
    return elements

procedure calculateSequenceSimilarity(Elements source, target)
Begin
    diff ← diff_lib.SequenceMatcher()
    diff.set_seq1(source)
    diff.set_seq2(target)
    ratio ← diff.ratio() * 100
return ratio
```

3.3.4. Performance metrics

As some AUTs cannot use Istanbul-middleware to measure code coverage, additional metrics are necessary. This paper introduces the number of input pages, input page depth, and the breadth of input page coverage increase (ICI breadth) as alternatives to code coverage. Fig. 12 shows a simple directive tree. Based on this figure, the number of input pages, the input page depth, and the ICI breadth are determined as follows.

(1) Number of input pages: The input nodes in the directive tree of Fig. 12 represent input pages. As shown in the diagram, there are four such nodes, indicating that the number of input pages in this example is 4.

(2) Input page depth: This subsection describes the number of nodes on the longest path from the root to the deepest input page node in the directive tree. For example, the input page with ID 1821633496 has a depth of 1. Extending from this input page node, there is a directive with ID 77eb5790 that leads to an input page with ID 1068395108, which is defined as having a depth of 2. Therefore, the input page depth in the example is 2.

(3) ICI breadth: The number of directive nodes under the input page nodes in the directive tree. According to Fig. 12, three input page nodes extend to directive nodes, indicating that the ICI breadth in the example is 3.

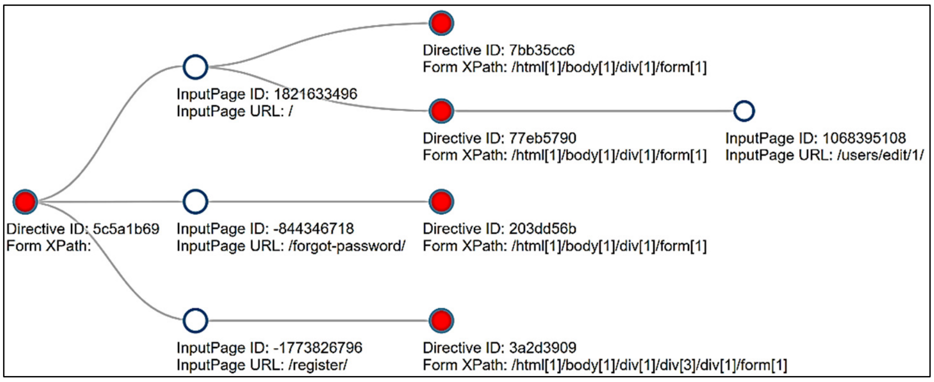
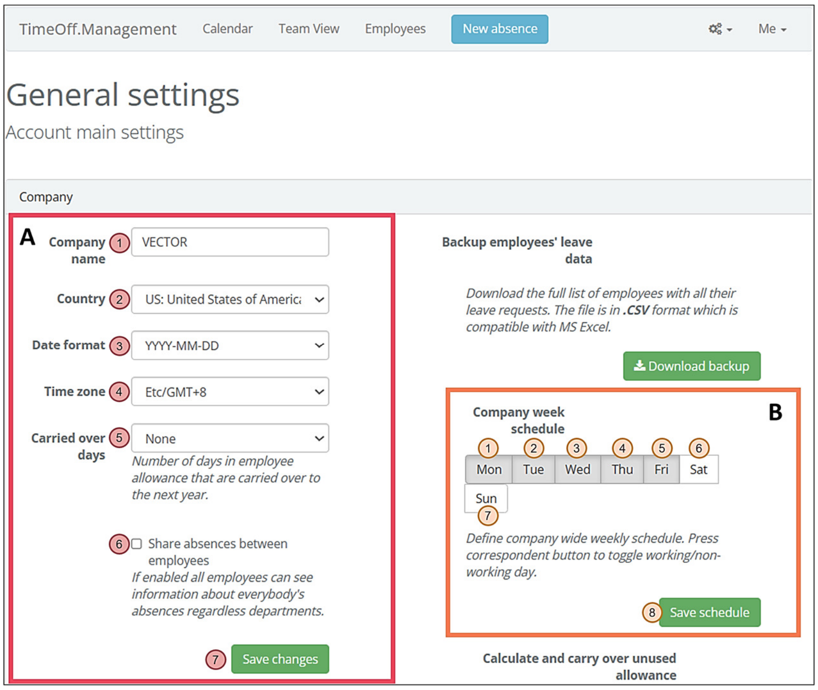

Directive ID: 7bb35cc6
Form XPath: /html[1]/body[1]/div[1]/form[1]
InputPage ID: 1821633496
InputPage URL: /
Directive ID: 77eb5790
Form XPath: /html[1]/body[1]/div[1]/form[1]
InputPage ID: 1068395108
InputPage URL: /users/edit/1/
Directive ID: 5c5a1b69
Form XPath:
InputPage ID: -844346718
InputPage URL: /forgot-password/
Directive ID: 203dd56b
Form XPath: /html[1]/body[1]/div[1]/form[1]
InputPage ID: -1773826796
InputPage URL: /register/
Directive ID: 3a2d3909
Form XPath: /html[1]/body[1]/div[1]/div[3]/div[1]/form[1]

Fig. 12 Example of a directive tree

3.4. Multiple web forms on a page

Webpages consist of a series of HTML tags, which can be used repeatedly in web design. Web forms, in particular, use the form tag, with form elements nested under it. Consequently, a single page can possess multiple form tags, which is common in settings pages of web applications. In the prior approach, the agent identifies an input page as a page containing a form tag. A form tag can include elements such as input fields or buttons. Web design does not restrict an input page to having only one form. Therefore, if an input page has multiple form tags, the agent may fail to perform a valid operation due to its inability to identify the specific form it is interacting with. This situation is particularly apparent when the agent is Monkey. As Monkey randomly chooses one field to fill, it is possible that Monkey fills one field of Form A, jumps to fill one field of Form B, and subsequently clicks "submit" for an incomplete form. In this way, efficiency is compromised. With the proposed method, Monkey will interact with only one form at a time, thus improving efficiency.

This issue can be illustrated using the TimeOff.Management page. As shown in Fig. 13, multiple form tags are used to separate different setting interfaces on this page. In this partial view, two form tags, referred to as Form A and Form B, are used. The corresponding source code of the webpage is shown in Fig. 14. Form A contains seven interactive elements, while Form B contains eight interactive elements.



General settings

Account main settings

Company

Company name: VECTOR

Country: US: United States of Americ

Date format: YYYY-MM-DD

Time zone: Etc/GMT+8

Carried over days: None

Number of days in employee allowance that are carried over to the next year.

□ Share absences between employees

If enabled all employees can see information about everybody's absences regardless departments.

Backup employees' leave data

*Download the full list of employees with all their leave requests. The file is in **.CSV** format which is compatible with MS Excel.*

Company week schedule

Define company wide weekly schedule. Press correspondent button to toggle working/non-working day.

Calculate and carry over unused allowance

Fig. 13 Partial screenshot of the settings page in TimeOff.Management

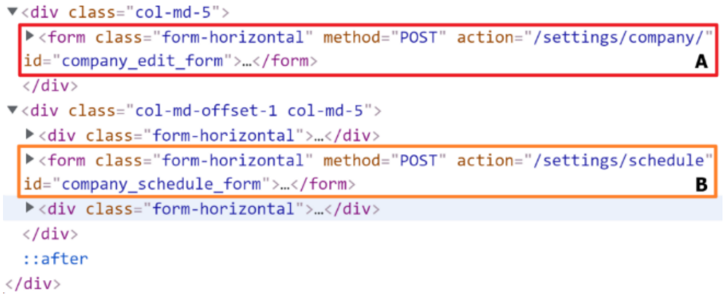

```
▼<div class="col-md-5">
  ▶<form class="form-horizontal" method="POST" action="/settings/company/"
  id="company_edit_form">…</form>                                         A
  </div>
▼<div class="col-md-offset-1 col-md-5">
  ▶<div class="form-horizontal">…</div>
  ▶<form class="form-horizontal" method="POST" action="/settings/schedule"
  id="company_schedule_form">…</form>                                     B
  ▶<div class="form-horizontal">…</div>
  </div>
  ::after
</div>
```

Fig. 14 Partial source code of the settings page in TimeOff.Management

In the prior approach, the agent interacted with the elements of both forms without distinction, resulting in interactions with all 15 elements when filling out form values. However, these two forms are independent and should be tested individually. If the agent interacts with or fills out the fields of Form A but clicks the "Save schedule" button in Form B, the settings of Form A will not be saved. The settings of Form A will only be retained if the agent interacts with Form A and clicks the "Save changes” button in Form A. Therefore, this paper proposes restricting the agent to interact only with the elements of specified form tags on a single form using XPath. This approach aims to enhance the agent’s efficiency in completing form-filling operations within a limited number of steps.

The method is illustrated in Fig. 15. One XPath from a form will be treated as one learning task, enabling the agent to interact with the identified form using the XPath. This approach can handle input pages with either a single form or multiple forms. In the prior approach, handling the input page was treated as a single task. In the proposed method, handling the input page is divided into multiple tasks based on the number of form tags, which are subsequently assigned to the agent for learning. This method can improve the agent’s efficiency in form filling, but increasing the number of forms requires more learning tasks to handle the input page. Despite such a requirement, the proposed approach remains more efficient for an input page with multiple forms. Notably, the main advantage of the proposed multi-form handling approach is to reduce the training steps, as demonstrated in Experiment 2. It does not significantly improve code coverage or other metrics compared to the original approach.

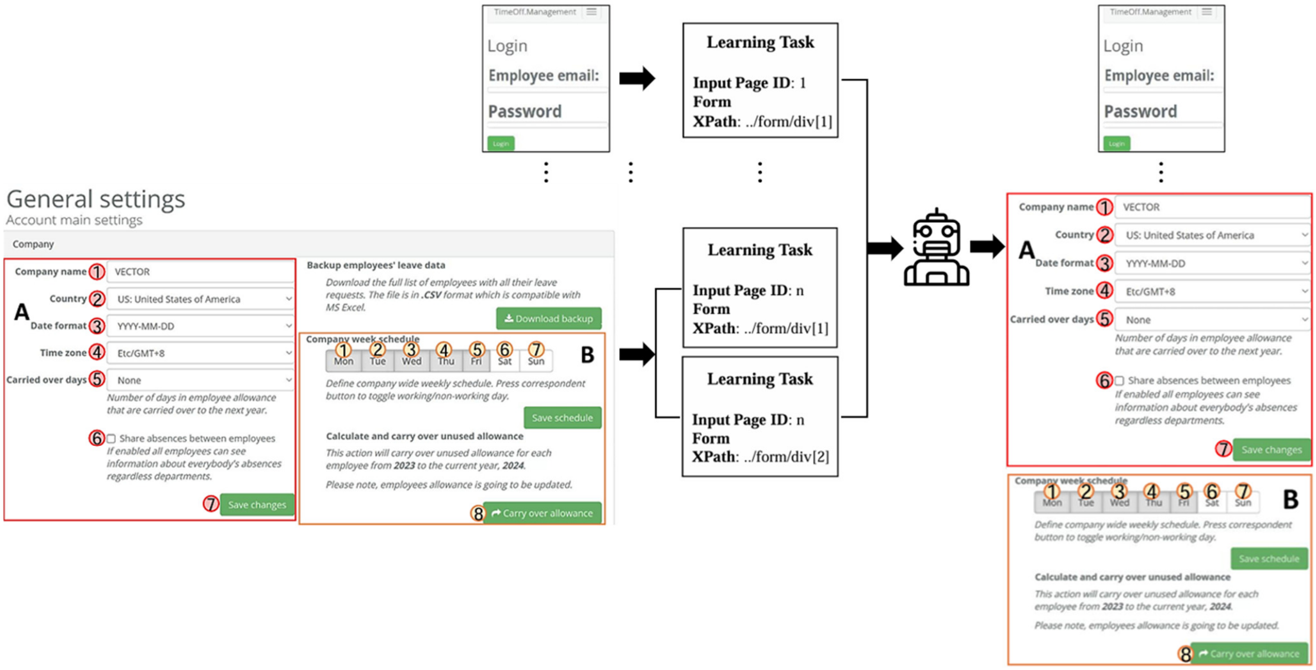


Fig. 15 Illustration of handling multiple forms by using one XPath as one learning task

3.5. *Reinforcement learning* (*RL*)

Typically, there are several different machine-learning approaches available in the literature, such as supervised learning, unsupervised learning, and RL. Unlike supervised and unsupervised learning, RL differs from the other two approaches as it relies on the use of rewards to determine suitable actions for an agent. It has been highly successful in playing video games [25]. As mentioned previously, the agent used in the proposed approach is trained through RL. It is typically modeled as a Markov decision process defined by the tuple ($\boldsymbol{S}, \boldsymbol{A}, P_a, R_a$), where $\boldsymbol{S}$ is a set of states, $\boldsymbol{A}$ is a set of actions, $P_a(S, S')$ symbolizes the transition probability from state S to state S', and R_a is the immediate reward received after taking action a to change state from S to S'. In the present case, the state is derived from an input form, and the action is one of the possible ways to interact with the form, such as filling in the name or clicking the “submit” button. The objective of the agent is to maximize the expected reward through the learning process.

This section covers a detailed description of the training process, the list of actions, the chosen reward function, and the agent implemented with fastText and neural networks. The deep Q-network (DQN) [25] RL algorithm is used to train the agent in the experiments.

3.5.1. Training process for RL agent

Fig. 16 illustrates the process implemented to train the RL agent. The modifications of USAGI are marked with a border and background for emphasis. The process consists of two phases: exploration and training. During the exploration phase, the agent employs Monkey, which takes random actions to navigate the web application, identifying input pages and directives to be included in the training set. Specifically, Monkey repeatedly selects one action from Table 1 at random to interact with the form until it captures an effective directive (to be covered in the next paragraph). Once an effective directive is found, it is added to the training set. This process continues with random trials until the maximum number of steps is reached. After the exploration phase is properly performed, the training phase commences, where the first generation of the agent is trained using the collected dataset. If necessary, a second exploration phase is conducted, following the same procedure but targeting a different web application to further enrich the training set with additional input pages and directives.

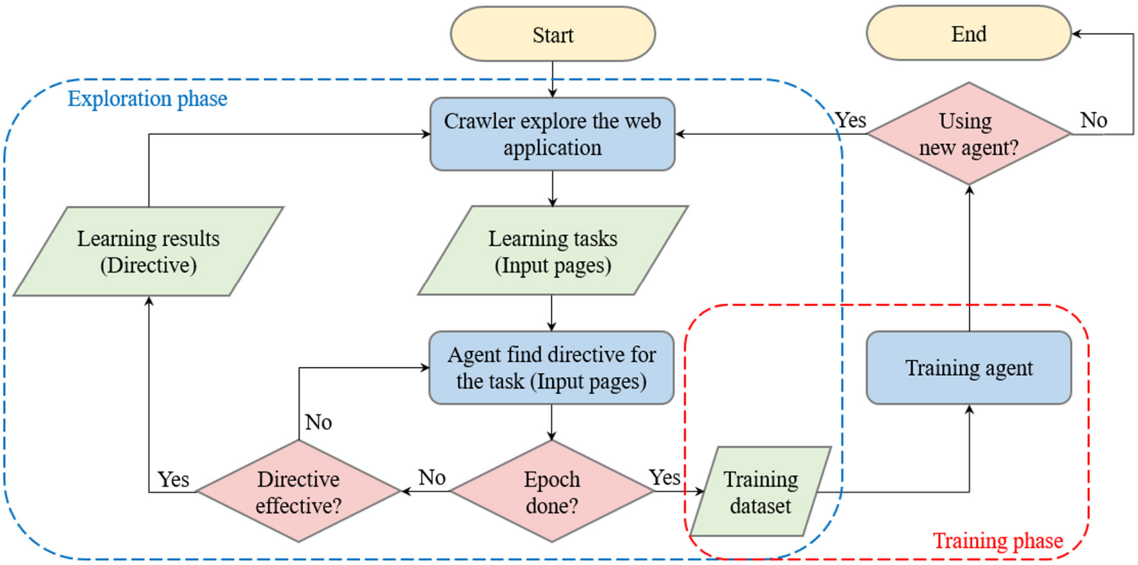

Start
End
Exploration phase
Crawler explore the web application
Yes
Using new agent?
No
Learning results (Directive)
Learning tasks (Input pages)
Agent find directive for the task (Input pages)
Training agent
No
Yes
Directive effective?
No
Epoch done?
Yes
Training dataset
Training phase

Fig. 16 Schematic diagram of the training process

In the USAGI framework, a directive is defined as effective if it leads to an increase in code coverage. This paper redefines it based on the appeared new screen page after the agent interacts with the web page. Only if the similarity between the new page and the stored pages is lower than a threshold, the directive is deemed effective. If a set of actions (e.g., filing forms and clicking the submit button) leads to an effective directive, the actions are stored for training. To implement this new definition, the web page comparison method described in Section 3.3 is utilized, replacing the original implementation of directive effectiveness. To implement the multi-form method outlined in Section 3.4, the XPath attributes of each form will be added to the learning task. This enables the crawler to create the appropriate number of learning tasks based on the number of forms presented on the input page. Each learning task will be interacted with the agent one by one.

Once training is complete, the flow outlined in Fig. 10 can be used to perform automated AUT testing. However, during the testing phase, a different web application is to be used to study the generalization capability. Initially, the crawler locates an input page and adds it to the learning pool (see also Fig. 6). Learning tasks are subsequently derived from the input page. The agent interacts with one form in each learning task at a time. If the agent identifies a valid directive, it inspects whether the page after an interaction is new. The results are recorded, and the process is repeated until the allotted time expires.

3.5.2. Actions

The agent in RL selects an action from the action set and sends it to the environment. The design of these actions significantly impacts the overall learning performance. In the present application, continuous-valued actions are not applicable, necessitating the use of a finite set of actions. However, the number of actions affects training efficiency. The number of actions is directly proportional to the required training time while defining a small set of actions that can effectively explore web applications is challenging due to the wide variety of interactive elements on web pages, such as input fields, buttons, and hyperlinks. Each interaction, such as filling a value or clicking a hyperlink, requires a distinct action. Furthermore, different input fields need their respective corresponding actions, as one action is associated with only one value. Since input fields can include names, addresses, telephone numbers, emails, etc., the action set is inevitably massive. Consequently, identifying a concise set of actions that can train the RL agent to interact with a web application requires careful design.

This paper introduces a set of actions designed for the proposed approach. These actions are meticulously crafted to handle various input fields. The action types include click, change focus, and input. The input type encompasses six specific actions: email, number, password, random string, date, and full name, as detailed in Table 1. Each action type derives distinct effects. For instance, change focus moves the agent to the next interactive element, input fills in the value for the currently focused element, and click triggers a button element.

Table 1 List of actions

Action id	Action type	Input value
a_1	Click	Click
a_2	Change focus	Change focus
a_3	Input	Email
a_4		Number
a_5		Password
a_6		Random string
a_7		Date
a_8		Full name

3.5.3. Rewards

The design of the reward function in RL is crucial. To improve the efficiency of the agent, the following goals are considered. The first goal is to enable the agent to mimic the actions collected by Monkey that yield effective directives. The second goal is to ensure that the agent clicks the “submit” button as soon as it completes filling all fields. To attain the first goal, the reward function is designed to encourage the agent to use the same actions collected during training. This means the agent learns how to interact with previously explored web pages, originally collected by Monkey. If the action of the agent corresponds to the same action in the action history record for this particular form field, a positive reward is given.

Otherwise, a negative reward is assigned. It should be noted that only action sets leading to effective directives are stored in the training dataset, and the directive effectiveness is determined based on the page-compare method. Therefore, a positive reward indicates that the agent follows one of the action sequences leading to an effective directive. In other words, through training, the agent would eventually choose a sequence of “effective” actions, leading to a new webpage. Additionally, to encourage the agent to submit the form directly after filling it out, without performing extra actions, a positive reward is provided based on the ratio of the number of steps completed in the current episode to the total number of episode steps. With the above designs, the trained agent can fill out a form and submit it with fewer actions.

The reward for action *a* is computed as:

$$reward(a) = \begin{cases} a, \; if \;\; the \; action \; meets \; the \; history \; record \\ -a, \; if \;\; the \; action \; does \; not \; meet \; the \; history \; record \\ \beta \times \dfrac{episode \; step}{episode \; step \; fraction}, \; if \;\; the \; page \; is \; submitted, \; and \; meets \; the \; history \; record \end{cases} \tag{1}$$

where α and β are positive integers.

3.5.4. State and agent

The state of the environment serves as the input to the agent. Since the agent operates as a feedforward neural network (as depicted in Fig. 17), the state must be represented by vectors. To capture the content within a form, features extracted through word embedding are utilized. In this design, the tag of the web element in focus and its corresponding text (label) in the form are used as inputs for fastText [26]. The fastText is defined as "a library for efficient learning of word representations and sentence classification" [26].

In the experiments, a pre-trained model was adopted. This model, when given a word, would produce an array of 300 floating-point numbers, representing the word as a point in an internal representation space. Consequently, this process generates two 300-dimensional vectors representing the label and tag in the word vector. These vectors are concatenated to form a single 600-dimensional vector, which is followed by a hidden layer containing 64 nodes. Finally, the output layer consists of 8 nodes, corresponding to the 8 actions listed in Table 1.

In summary, the state is the label and tag of a field in a form, and the action is the value to be filled in the field. The reward is calculated based on Eq. (1). The agent receives a positive reward if the chosen action matches the one previously used by Money to successfully find a new effective directive. Monkey uses the page-compare method to determine if a new page is discovered or not. If so, the new directive is saved.

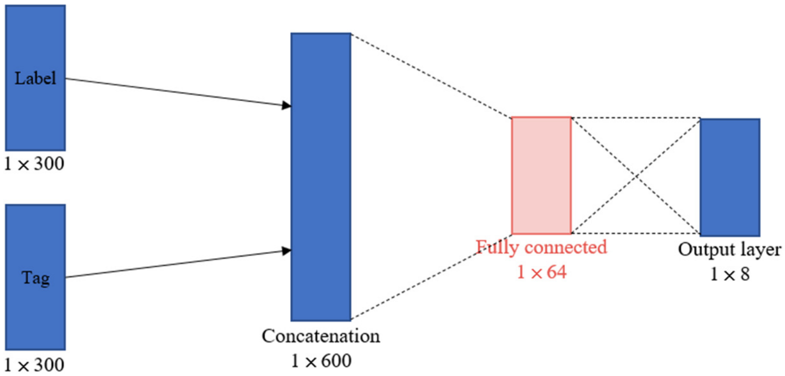


Fig. 17 Network architecture of the agent

4. Experiments and Results

The proposed approach is evaluated through the following three experiments:

Experiment 1: The webpage comparison method is compared with the code coverage vector as guidance criteria for performance.

Experiment 2: The performance improvement of using multiple-form handling over the original USAGI is measured. It is used to demonstrate the efficiency improvement during exploration when encountering multiple forms on a web page.

Experiment 3: The performance gap between using a random-action Monkey and the proposed approach on a new, untrained web application is tested.

4.1. Performance evaluation

To evaluate the performance of the proposed method, the focus is on whether the directives generated by the agent can guide the crawler to explore new states. The evaluation criteria include: (1) assessing code coverage, which encompasses statement coverage and branch coverage; (2) evaluating the input page of the directive tree, considering the number of input pages, the depth of input pages, and ICI breadth achieved by the directives generated by the agent.

4.2. Experimental environment

The experiments are conducted using a modified version of USAGI, as detailed in Section 3. Table 2 lists the computer equipment and software versions used in the experiments. Table 3 provides information on the AUT (Applications Under Test) and their respective versions. Five web applications were selected for three key reasons:

(1) these selected web applications are all open source

(2) TimeOff.Management, NodeBB [27], and KeystoneJS can achieve code coverage with Istanbul-middleware [8]

(3) Django Blog [28] and Spring Petclinic [29] empower testing the applicability of the proposed method on web applications without code coverage.

It is noteworthy that Istanbul-middleware is unable to compute the code coverage for Django Blog (written by Python) and Spring Petclinic (written by Java). While other code-coverage tools may exist for these two applications, it nevertheless complicates the integration of the experimental platform and increases the platform development time. Table 4 outlines the parameters for the model testing phase, while Table 5 details the hyperparameters for the agent training phase.

Table 2 List of experimental equipment and software versions

Equipment/Software	Specifications/Model/Version
CPU	Intel Core I7-6700
RAM	16GB
OS	Ubuntu 18.04
GPU	GeForce GTX 780
Docker	Version: 20.10.15
Python	Version: 3.6.9

Table 3 Web application versions

Application name	Version	GitHub stars count	Lines of code	Number of branches
TimeOff.Management [19]	V0.10.0	751	2698	1036
NodeBB [27]	V1.12.2	12758	7334	3241
KeystoneJS [22]	V4.0.0-beta.5	-	5267	3444
Django Blog [28]	V1.0.0	23	-	-
Spring Petclinic [29]	V2.6.0	5741	-	-

Table 4 Parameter list for the test model

Parameter type	Value
Episode steps	16
Number of actions	8
Starting URL	/
Crawljax crawling depth	5
Probability of not using the agent for action selection	50%

Table 5 Hyperparameter setting list

Hyperparameter	Value
Agent training steps	Number of pages × 50,000
Target network update frequency (steps)	250
Learning rate	0.0005
Initial ε value	1
Final ε value	0.1
Exploration fraction	0.5

4.3. Experiment 1

In this experiment, the directive effectiveness method, determined based on the page-comparison algorithm, is compared with the code coverage vector as guidance criteria for performance comparison. Both the modified and original USAGI frameworks use the same training method to ensure a fair comparison. The experimental steps are as follows:

(1) Use Monkey to explore TimeOff.Management with code coverage and training an agent, called agent_org.

(2) Use Monkey to explore TimeOff.Management with the proposed directive effectiveness determination method and train another agent called agent_new.

(3) Test KeystoneJS three times with the trained agents and average the results. The agent's testing time is limited to 30 minutes. It is worth noting that the training and testing applications are different.

Experimental results are presented in Fig. 18 and Fig. 19 for statement coverage and branch coverage, respectively. The results indicate that agent_new renders better code coverage than the original version, agent_org. This confirms that the proposed method to determine directive effectiveness can be used during Monkey exploration.

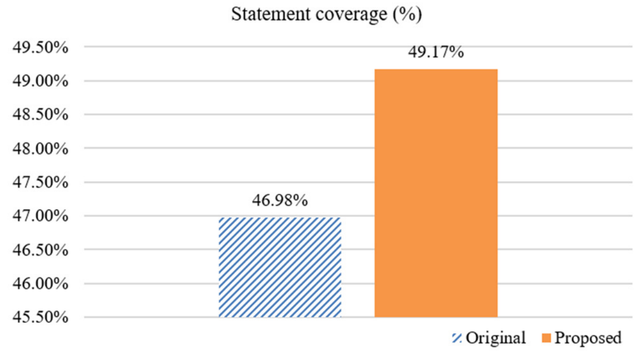


Fig. 18 Comparison of statement coverage (%) for both agents

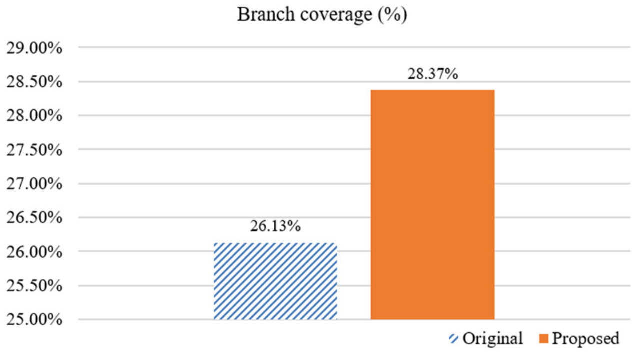


Fig. 19 Comparison of branch coverage (%) for both agents

In addition to code coverage, the number of input pages, input page depth, and ICI breadth are also measured. The results, shown in Fig. 20 to Fig. 22, demonstrate that the proposed metrics (number of pages, etc.) have the same ordering as the code coverage does. In other words, an agent that leads to higher code coverage also has a higher number of input pages, input page depth, and/or ICI breadth. Therefore, these metrics can be used to measure the performance of the agents.

Based on the above observation, the conclusions of this experiment are:

(1) The method to determine directive effectiveness, based on page compare, can replace code coverage for Monkey to explore training applications

(2) The introduced metrics contain correlation with code coverage well and can be used to measure the agent's performance.

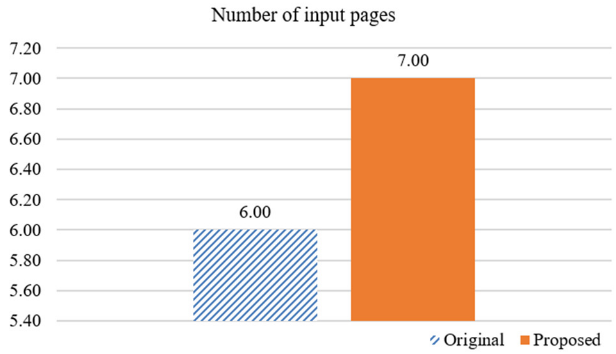


Fig. 20 Comparison of the number of input pages for both agents

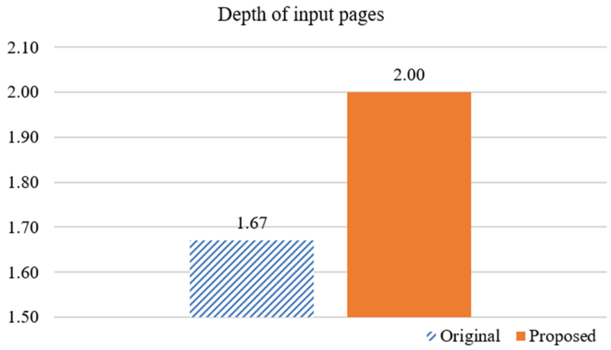


Fig. 21 Comparison of input page depth for both agents

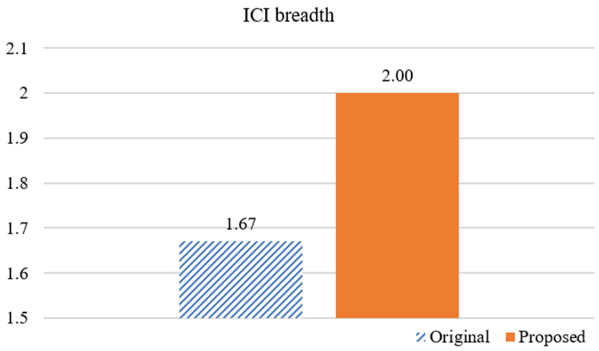


Fig. 22 Comparison of ICI breadth for both agents

4.4. Experiment 2

This experiment measures the performance improvement of the proposed multi-form handling over the original version. As shown in Fig. 16, the exploration phase uses the webpage comparison method to determine whether a user scenario has been explored. The experiment steps are as follows:

(1) Monkey is employed to explore TimeOff.Management by interacting with all form tags on a single input page for 10,000 steps, with data recorded every 50 steps.

(2) The process is repeated, except that Monkey interacts only with a specified form on the page.

Notably, TimeOff.Management contains multiple forms on a single page. Therefore, it is suitable for measuring the improvement of the multi-form handling approach.

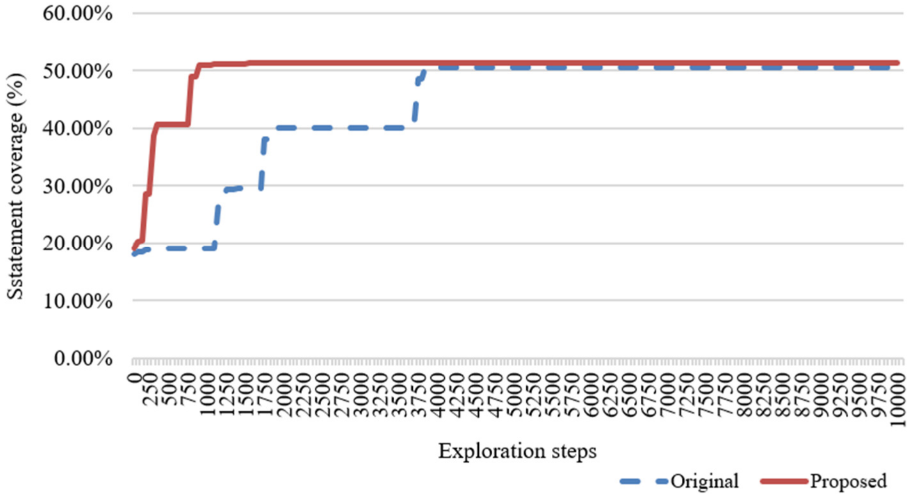


Fig. 23 Comparison of statement coverage (%) using both methods

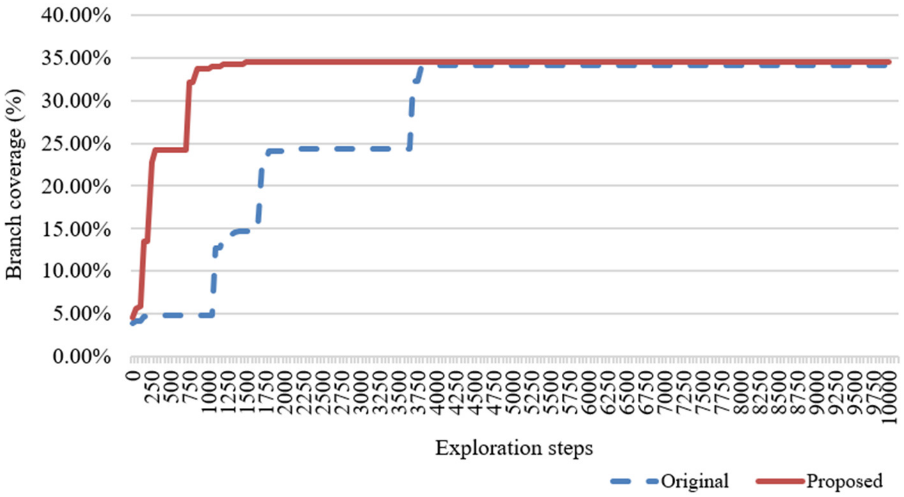


Fig. 24 Comparison of branch coverage (%) using both methods

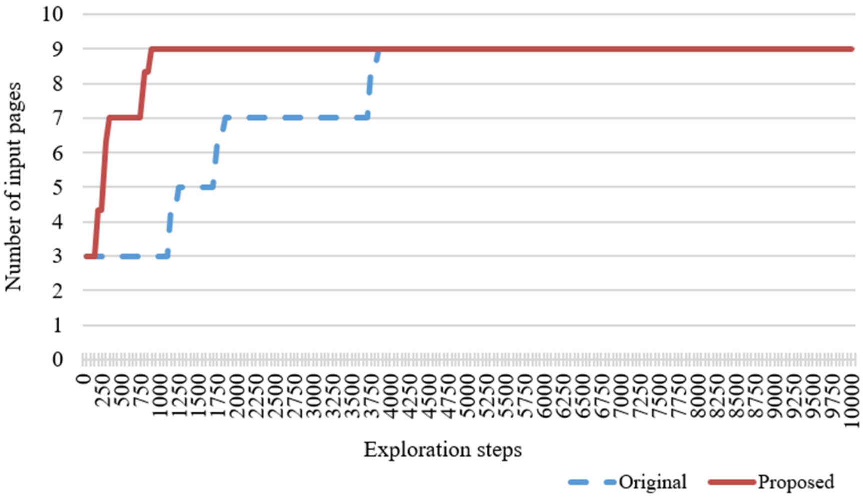


Fig. 25 Comparison of the number of input pages using both methods

The experimental results are presented in Figs. 23 to 27. Specifically, Fig. 23 shows the statement coverage, Fig. 24 displays the branch coverage percentage, and Figs. 25 to 27 illustrate the number of input pages, input page depth, and ICI breadth, respectively. The results indicate that the proposed method outperforms the original method in exploring the AUT. The proposed approach reaches the same metric values (coverage, depth, etc.) with fewer exploration steps or shorter training time. This observation suggests that fewer exploration steps can be used in the future. Consequently, the third experiment will employ the proposed multi-form handling method for Monkey's exploration.

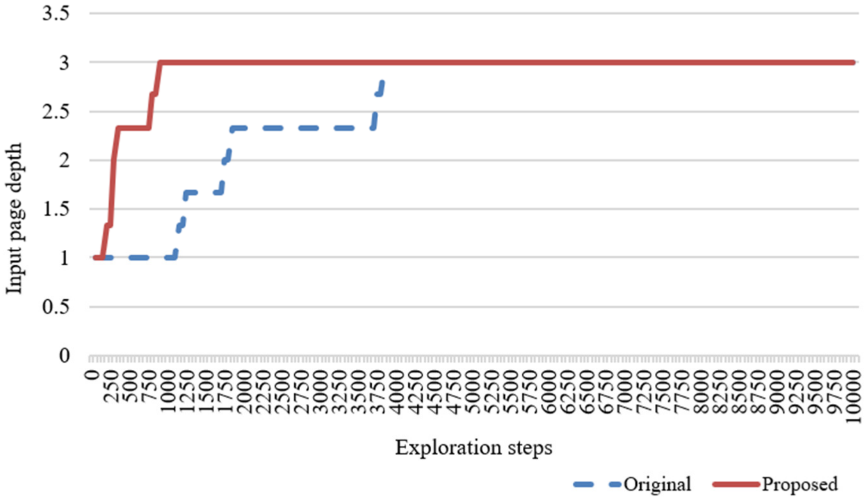


Fig. 26 Comparison of input page depth using both methods

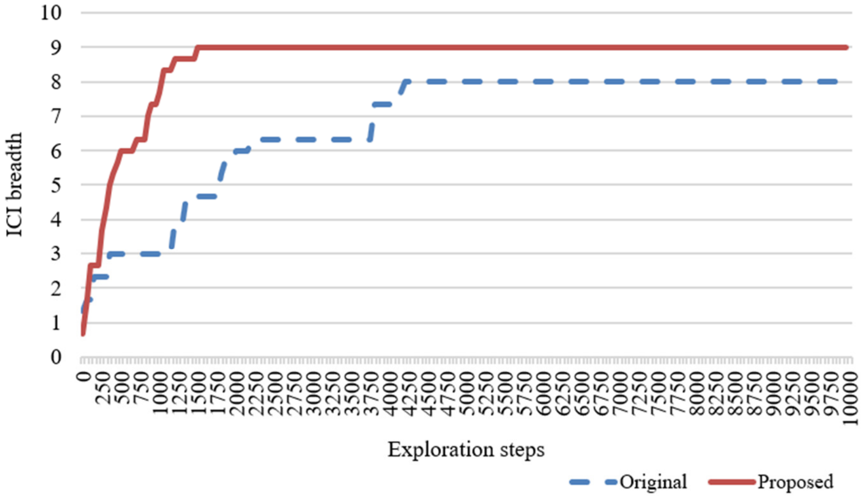


Fig. 27 Comparison of ICI breadth using both methods

4.5. Experiment 3

Table 6 Detailed training information of agents

Agent abbreviation	Web applications used in training	Number of training pages	Exploration + training time (hh:mm)
timeoff	TimeOff.Management	10	07:08
nodebb	NodeBB	4	42:19
keystonejs	KeystoneJS	37	20:45
dejangoblog	Django Blog	4	06:38
petclinic	Spring Petclinic	16	12:05

Experiment 3 aims to compare the performance of Monkey with that of the trained agents on a new, previously untrained AUT. The training procedure mirrors that of Experiment 1, incorporating the multi-form handling capability. The training details for the agents are provided in Table 6. The "Agent abbreviation" field assigns a simple name to each trained agent. For instance, the agent trained with the TimeOff.Management web application is referred to as “timeoff.” The number of trial steps for Monkey is 10,000 steps (exploration phase in Fig. 16), and the number of agent training steps is displayed in Table 5 (training phase in Fig. 16).

In this experiment, the test AUTs are selected from one of the five web applications listed in Table 3, using the other four applications as the training sources. For instance, if the AUT is TimeOff.Management, all agents except Agent_timeoff are used for testing. For AUTs with available coverage information, only branch coverage is reported for brevity. For AUTs that lack coverage information, the number of input pages discovered is used as the performance metric. Each test agent is limited to 250 trial steps, with data recorded every 50 steps.

For TimeOff.Management, the branch coverage information is available and presented in Fig. 28. Notably, the line representing Monkey is closely aligned with that of Petclinic. The results show that all agents, except Agent_petclinic, significantly outperform Monkey. This demonstrates that the proposed approach effectively trains an agent to test a new, previously unseen AUT. Agent_petclinic, however, performs similarly to Monkey. Upon closer inspection, it was found that this application lacks complex forms on its web pages, preventing Agent_petclinic from learning how to interact with more intricate forms. This suggests that the web application used as the training source should be somewhat similar to the AUT in terms of form fields.

The above experimental steps were repeated to test NodeBB and KeystoneJS, both of which have code coverage values. The experimental results are presented in Fig. 29 and Fig. 30, respectively. The findings indicate that when the training application is similar to the AUT, the agent demonstrates a notable level of generalization capability, enabling it to perform automated testing effectively, even on an AUT never encountered before.

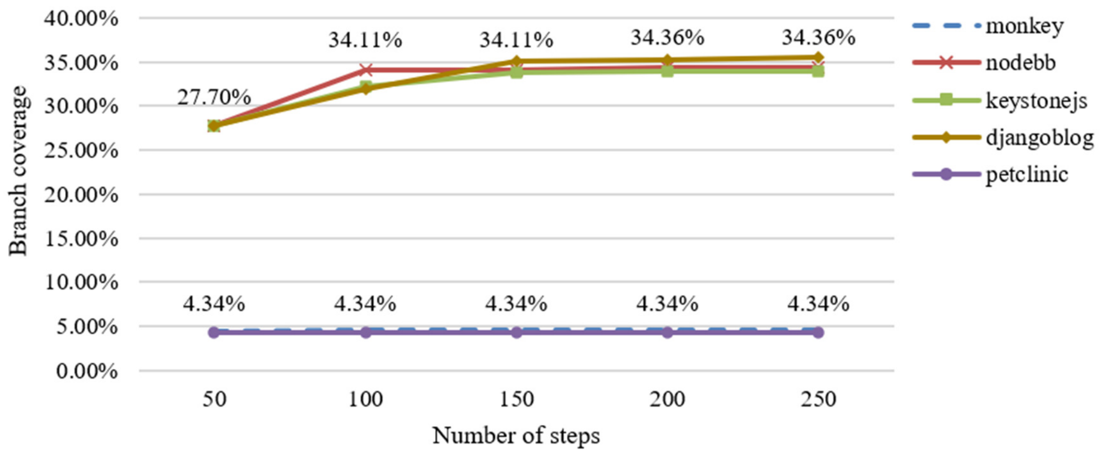


Fig. 28 Branch coverage (%) comparison for TimeOff.Management

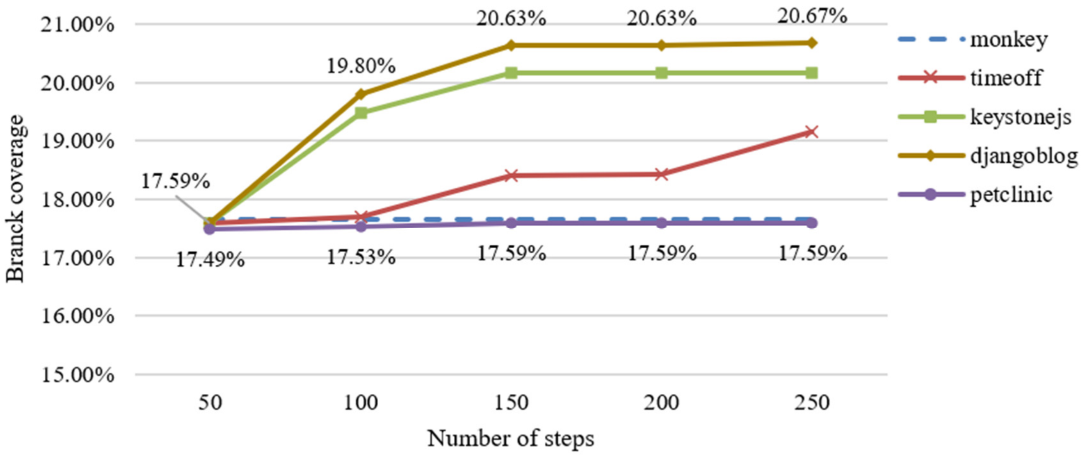


Fig. 29 Branch coverage (%) comparison for NodeBB

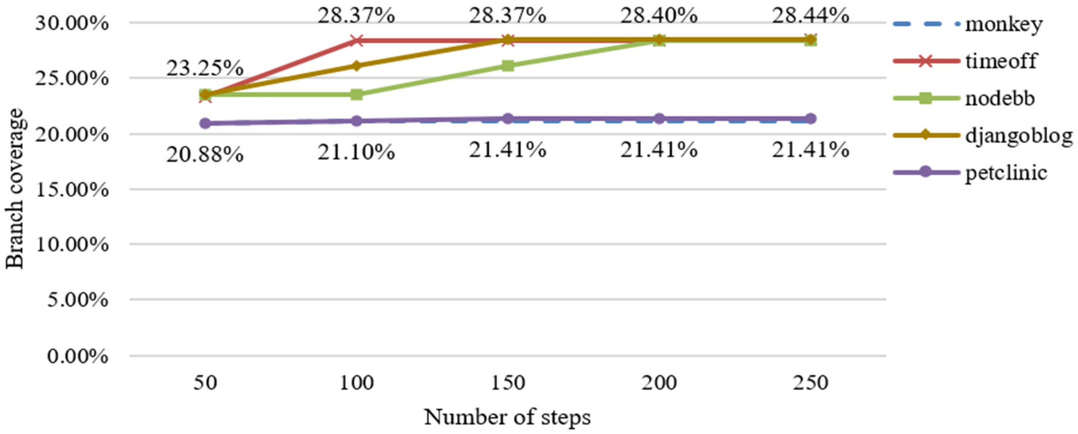


Fig. 30 Branch coverage (%) comparison for KeystoneJS

The experimental results for Django Blog are presented in Fig. 31. It should be noted that the lines of Monkey, petclinic, and nodebb are closely aligned to each other. Since code coverage values are not available for this AUT, only the number of input pages is reported. Fig. 30 highlights the advantages of the proposed approach when a suitable training application is used. In this case, TimeOff.Management proves to be the most effective training application due to its numerous forms, each containing many fields. After training with this application, the agent efficiently fills out forms in the AUT.

The experimental results for Spring Petclinic are shown in Fig. 32. The number of input pages increases with the number of trial steps because this AUT accepts most value types in each form field with minimal restrictions, making it relatively easy to test. Despite this, trained agents generally exhibit higher efficiency, measured in steps, when testing this application. Overall, based on the results from Figs. 28 to 32, it can be concluded that the proposed method outperforms Monkey in exploring web applications within a fixed number of steps.

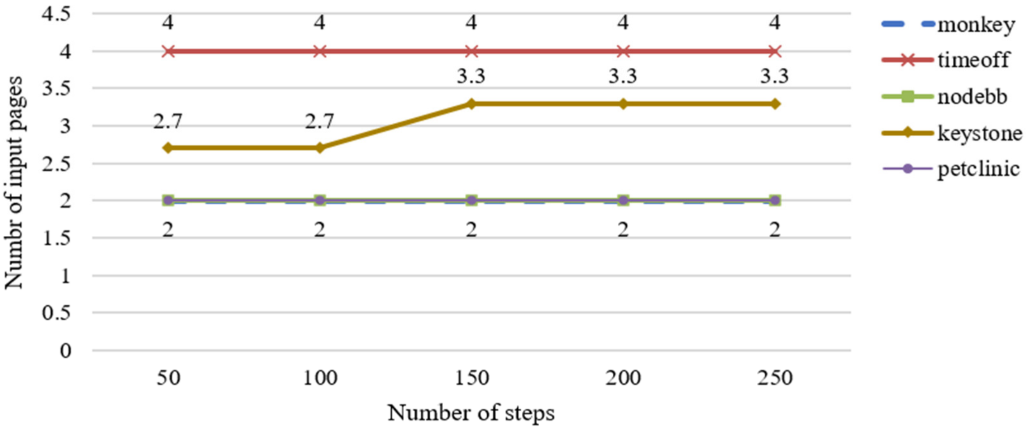


Fig. 31 Input pages comparison for Django Blog

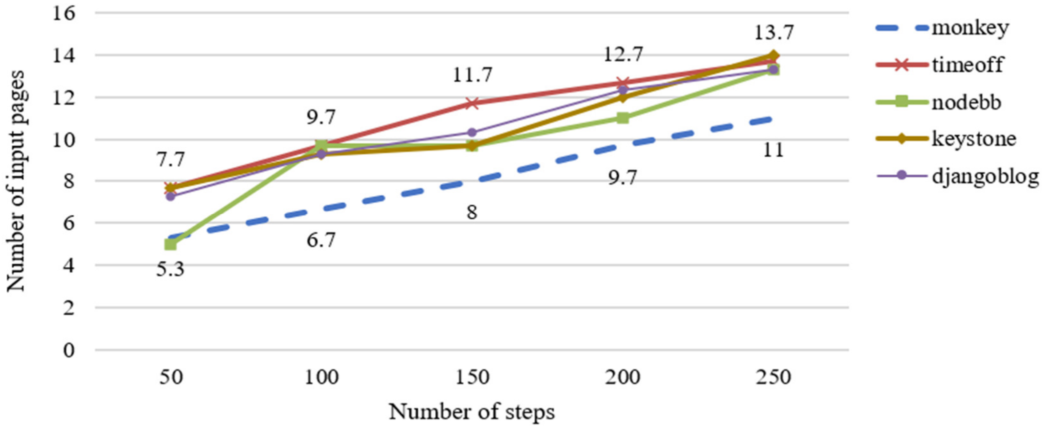


Fig. 32 Input pages comparison for Spring Petclinic

The training times for the agents are listed in Table 3, revealing a significant variation in the exploration-plus-training time. This variation is primarily due to the frequency of communications between the agent (Monkey) and Crawljax, as illustrated in Fig. 6. Notably, Crawljax sometimes exhibits long response times (up to one minute), especially when interacting with AUT. Since these experiments were conducted as a proof of concept, efficiency was not optimized. Therefore, the execution times in Table 3 are provided for reference only. Improving the communication method with Crawljax could significantly reduce these execution times.

The test times typically range from less than 20 minutes to over one hour. Exact values are not included in Table 3 because each agent interacts with four AUTs. Moreover, the most critical issue is the response time of Crawljax. The agent per se can execute within seconds. When compared with the exploration-plus-training time, the test time for each AUT is much shorter. In contrast, other approaches, such as QExplorer [17], typically require more than 4 hours of execution to achieve satisfactory code coverage for a single AUT, and the trained knowledge cannot be reused. The proposed approach is advantageous since it enables “offline” training during the development phase of an application.

4.6. Threats to validity

The threats to internal validity arise from the implementation of the platform and how the experiments were conducted. Firstly, the agent in the current design has a limited number of actions, hindering its ability to fill certain fields, such as credit card numbers or zip codes. Additionally, the experimental platform is unable to perform comprehensive testing with all possible valid and invalid inputs, as well as boundary values. For example, when filling in the email field, either a valid email is used or values from other actions (such as Date or Password) are used. The test does not include any incomplete forms of email, such as “scyou@.” Generating a complete list of all possible values would result in excessive actions for the agent. Another potential threat is that the actions of Monkey are purely random. A Monkey with a better built-in strategy might yield different results in Experiment 2.

The threats to external validity are related to the AUTs selected for the experiments. The experiments were conducted on only five AUTs, which have a limited variety of field types in their forms. Consequently, further studies are required to determine whether the observations can be generalized to other AUTs. Additionally, the performance of the agent is influenced by the behavior of the training applications. For instance, if the training application accepts a random string as an email, the agent trained with this application may be unable to correctly fill the email field for other applications.

5. Conclusions

This paper presents a new webpage comparison method to replace code coverage for detecting new pages, enabling USAGI to test applications without relying on code coverage. Additionally, it presents a method for interacting with only one form on a webpage at a time, allowing Monkey to efficiently generate Directives for input pages. The experiments show that the modified framework outperforms the random-action Monkey in testing new, untrained applications, thus providing a promising starting point for fully automatic webpage testing.

Future improvements can include training the agent with a wider range of applications to evaluate its generalization ability for testing new applications, exploring alternative methods for detecting web screen changes beyond webpage comparison, and designing a new set of actions to accommodate various input types in forms. These enhancements would further improve the proposed framework.

Acknowledgment

This research was supported by the National Science and Technology Council, Taiwan, under grant number NSTC 112-2221-E-027-049-MY2.

Conflicts of Interest

The authors declare no conflict of interest.

References

[1] Siteefy, “How Many Websites Are There in the World?” https://siteefy.com/how-many-websites-are-there/, 2024.

[2] S. Jordan, “Online Presence Management Tips for Small Businesses,” https://clutch.co/resources/online-presence-management, 2024.

[3] A. Van Deursen, A. Mesbah, and A. Nederlof, “Crawl-Based Analysis of Web Applications: Prospects and Challenges,” Science of Computer Programming, vol. 97, part 1, pp. 173-180. 2015.

[4] S. Bennetts, “Crawljax,” https://github.com/zaproxy/crawljax, 2024.

[5] A. Mesbah, A. Van Deursen, and S. Lenselink, “Crawling AJAX-Based Web Applications Through Dynamic Analysis of User Interface State Changes,” ACM Transactions on the Web, vol. 6, no. 1, article no. 3, 2012.

[6] L. Brader, H. F. Hilliker, and A. C. Wills, Testing for Continuous Delivery with Visual Studio 2012, Redmond, Washington: Microsoft, 2012.

[7] C. H. Liu, S. D. You, and Y. C. Chiu, “A Reinforcement Learning Approach to Guide Web Crawler to Explore Web Applications for Improving Code Coverage,” Electronics, vol. 13, no. 2, article no. 427, 2024.

[8] K. Anantheswaran, “Istanbul-Middleware,” https://github.com/gotwarlost/istanbul-middleware, 2024.

[9] T. Gowda and C. A. Mattmann, “Clustering Web Pages Based on Structure and Style Similarity (Application Paper),” IEEE 17th International Conference on Information Reuse and Integration, pp. 175-180, 2016.

[10] K. Griazev and S. Ramanauskaitė, “HTML Block Similarity Estimation,” IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering, pp. 1-4, 2018.

[11] J. W. Lin, F. Wang, and P. Chu, “Using Semantic Similarity in Crawling-Based Web Application Testing,” IEEE International Conference on Software Testing, Verification and Validation, pp. 138-148, 2017.

[12] S. Carino and J. H. Andrews, “Dynamically Testing GUIs Using Ant Colony Optimization,” 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 138-148, 2015.

[13] J. Kim, M. Kwon, and S. Yoo, “Generating Test Input with Deep Reinforcement Learning,” Proceedings of the 11th International Workshop on Search-Based Software Testing, pp. 51-58, 2018.

[14] C. H. Liu, W. K. Chen, and C. C. Sun, “GUIDE: An Interactive and Incremental Approach for Crawling Web Applications,” The Journal of Supercomputing, vol. 76, no. 3, pp. 1562-1584, 2020.

[15] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, et al., “Automatic Web Testing Using Curiosity-Driven Reinforcement Learning,” IEEE/ACM 43rd International Conference on Software Engineering, pp. 423-435, 2021.

[16] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang, “Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration,” https://doi.org/10.48550/arXiv.1802.088022018.

[17] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, “QExplore: An Exploration Strategy for Dynamic Web Applications Using Guided Search,” Journal of Systems and Software, vol. 195, article no. 111512, 2023.

[18] X. Wang and W. Tian, “An Efficient Method for Automatic Generation of Linearly Independent Paths in White-Box Testing,” International Journal of Engineering and Technology Innovation, vol. 5, no. 2, pp. 108-120, 2015.

[19] Pavlo, “TimeOff.Management,” https://github.com/timeoff-management/timeoff-management-application, 2024.

[20] S. H. Chou, “Using Agents to Automatically Choose Input Data for Web Crawler to Increase Code Coverage,” Master thesis, Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei, Taiwan, ROC, 2020.

[21] W3C, “Document Object Model (DOM) Technical Reports,” https://www.w3.org/DOM/DOMTR, 2023.

[22] E. Hamilton, “Keystone,” https://github.com/keystonejs/keystone, 2024.

[23] M. E. Haase, “Page Compare,” https://github.com/TeamHG-Memex/page-compare, 2024.

[24] The Python Software Foundation, “Difflib — Helpers for Computing Deltas,” https://docs.python.org/3/library/difflib.html, 2024.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, et al., “Human-Level Control Through Deep Reinforcement Learning,” Nature, vol. 518, pp. 529-533, 2015.

[26] J. Janzen and Facebook Community Bot, “FastText,” https://github.com/facebookresearch/fastText, 2024.

[27] B. S. Uşaklı, “NodeBB,” https://github.com/NodeBB/NodeBB, 2023.

[28] Dušan, “Django Blog Demo,” https://github.com/reljicd/django-blog, 2023.

[29] D. Syer, “Spring PetClinic Sample Application,” https://github.com/spring-projects/spring-petclinic, 2023.