Web Element Identification by Combining NLP and Heuristic Search for Web Testing

Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{h-kirink, shinsuke, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—End-to-end test automation is critical in modern web application development. However, test automation techniques used in industry face challenges in implementing and maintaining test scripts. It is difficult to determine and maintain the locators needed by test scripts to identify web elements on web pages. The reason is that locators depend on the metadata of web elements and the structure of each web page. One effective way to solve such a problem of locators is to allow test cases written in natural language to be executed without test scripts. In this study, we propose a technique to identify web elements that should be operated on a web page by interpreting natural-language-like test cases. The test cases are written in a domain-specific language that independents on the metadata of web elements and the structural information of web pages. We leverage natural language processing techniques to understand the semantics of web elements. We also create heuristic search algorithms to explore web pages and find promising test procedures. To evaluate the proposed technique, we applied it to test cases for two open-source web applications. The experimental results show that our technique was able to successfully identify about 94% of web elements to be operated in the test cases. Our approach also succeeded in identifying all the web elements that were operated in 68% of the test cases.**

I. Introduction

In recent years, the prompt updating of software has become increasingly important in order to respond to rapid changes in market conditions. Developers need to verify their software works properly before release. This confirmation includes not only whether new features work properly but also whether existing features work as before. Testing that existing features work as they did previously is called regression testing. The cost of regression testing is overwhelming in software maintenance [1], [2]. Test automation is therefore an important technique to reduce cost.

In web application development, developers commonly use tools that automate end-to-end testing, and they need to implement and maintain test scripts. A test script enables the automation of the operations and verifications performed on web pages that are being tested. The implementation of test scripts is known to be costly. Dobslaw et al. [3] investigated the return on investment (ROI) of end-to-end test automation frameworks. The authors showed that, compared to manual testing, the initial implementation time was close to 90% of the total cost up until reaching the ROI. The study also claimed that the dominant cost is the initial time required to implement test scripts. One of the reasons for the high cost of implementing test scripts is that most end-to-end test automation tools depend on the metadata of web elements and the structure of web pages.

For example, Selenium [4], a de facto standard end-to-end test automation tool, requires locators to identify web elements. Some locators depend on metadata such as `id` or `name` attributes described in HTML document, and other locators use XPath. XPath is a query language for selecting a web element from an XML/HTML document. Developers often have to understand the detailed implementation of a web page to determine locators. In this way, test script implementation is obstructed by the dependence on metadata and the structure of each web page.

This dependence is also an obstacle to maintaining test scripts. Test scripts that use locators are known to be fragile. Some prior works have examined the fragility of locators. Christophe et al. [5] investigated eight open-source software repositories that include Selenium test scripts. The study showed that 75% of Selenium test scripts are changed more than once every nine commits (once every 2.05 days). Hammoudi et al. [6] examined breakages in 453 versions of Selenium IDE test scripts of eight web applications and classified the causes of test breakages. The results of their experiment showed that 73.62% of the breakages were caused by locators. From the above studies, we can see that using locators increases the cost of maintaining test scripts and hinders efficient regression testing.

Another major problem with end-to-end testing is the cost of creating and maintaining test cases. Note that this paper defines a test case as a specification of test procedures and expected results. This paper also defines a test script as an automated program to verify the specification. Writing test cases is important because not all tests are always automated, and not all people involved in testing understand test scripts written in a programming language. The test cases also require maintenance. If both test cases and test scripts exist, developers need to keep them consistent. Therefore, it is costly to create and maintain test cases, especially for fast-evolving applications.

One efficient way to solve the problems mentioned so far is to make test cases executable without test scripts. This is because it relieves developers from the problem of test script implementation and consistency between test cases and test scripts. Our goal is to make it possible to execute test cases written in natural languages without conventional test script implementation using locators. To achieve the goal, it is first necessary to be able to identify web elements from test case descriptions without depending on the implementation of applications. In this study, we propose a technique to identify web elements to be operated on web pages by interpreting test cases. The test cases that we are focusing on are written in a domain-specific language (DSL) without relying on metadata of web elements or the structural information of web pages. We leverage natural language processing (NLP) techniques to understand the semantics of web elements and test cases. We also create heuristic search algorithms to find promising test procedures from possible ones. To evaluate our proposed technique, we applied it to test cases for two open-source web applications. The experimental results show that our technique was able to successfully identify about 94% of web elements to be operated in the test cases. We also succeeded in identifying all the web elements that were operated in 68% of the test cases. Our experimental source code, the test cases, and the outputted test procedure are publicly available[1].

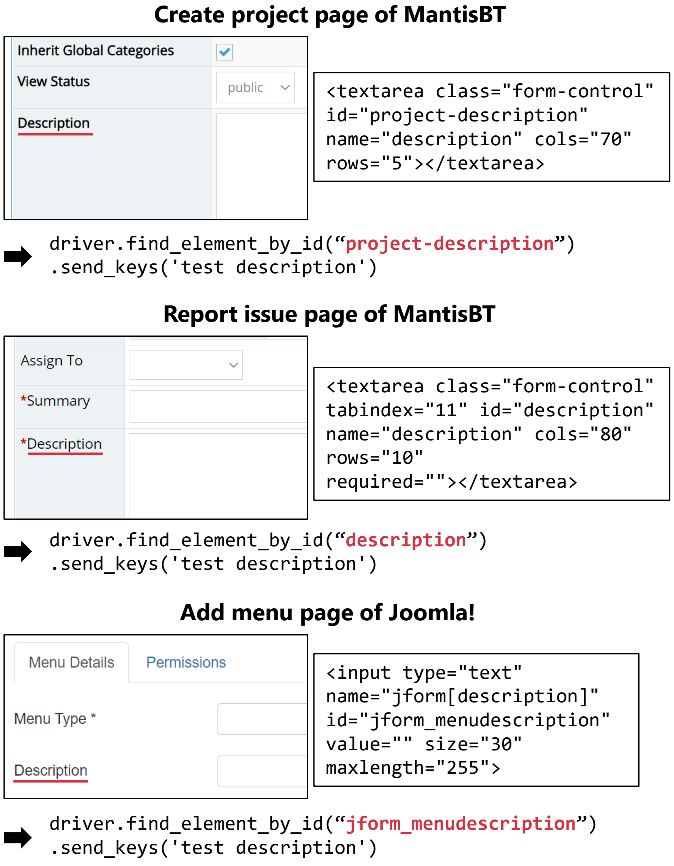
The contributions of this study are as follows:

- We propose a novel technique to identify web elements by interpreting test cases that are close to natural language.
- We propose an algorithm that combines NLP and heuristic search to find promising test procedures.
- Our experiments show the potential for semantic-based identification of web elements and reuse of test cases across multiple contexts and applications.

II. Motivating Example

We explained the problems of the conventional locator-based approach for end-to-end test automation in Section I. In this section, we introduce some examples of such problems when implementing test scripts using locators. Figure 1 shows the three different *description* input fields of Joomla![2] and MantisBT[3] and Python snippets with Selenium to enter the value “test description”. Even though these fields have similar roles, it is necessary to use different locators when scripting them since all of these fields are implemented differently. These differences in test script implementation make it difficult to reuse a part of test scripts across different contexts. If all these web elements could be represented by the word “description”, it would lead to reuse of a part of test scripts.

Figure 2 shows a drop-down list in the log-in module page of Joomla! and a Python snippet to select the value “Icons”. Despite the drop-down list of “Display Label”, the id and name attribute of the web element do not seem to be related to it. This would make it difficult for developers to understand what this snippet means when they read it. If a web element does not have an id or name attribute, the same problem will occur because XPath or CSS selectors would be used as a locator. In this case, we want to suggest using the string ”display labels” to identify this drop-down list.



Create project page of MantisBT

```
<textarea class="form-control"
id="project-description"
name="description" cols="70"
rows="5"></textarea>
```

```
driver.find_element_by_id(“project-description”)
.send_keys('test description')
```

Report issue page of MantisBT

```
<textarea class="form-control"
tabindex="11" id="description"
name="description" cols="80"
rows="10"
required=""></textarea>
```

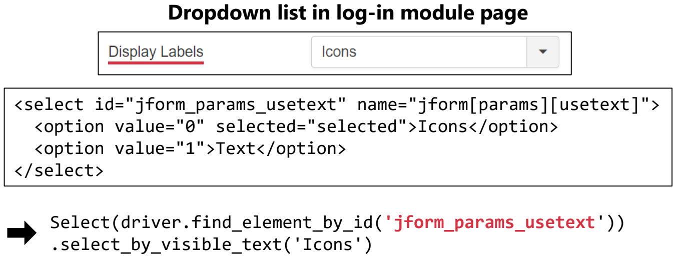
```
driver.find_element_by_id(“description”)
.send_keys('test description')
```

Add menu page of Joomla!

```
<input type="text"
name="jform[description]"
id="jform_menudescription"
value="" size="30"
maxlength="255">
```

```
driver.find_element_by_id(“jform_menudescription”)
.send_keys('test description')
```

Fig. 1. *Description* input fields and Python snippets to enter the value “test description”



Dropdown list in log-in module page

```
<select id="jform_params_usetext" name="jform[params][usetext]">
  <option value="0" selected="selected">Icons</option>
  <option value="1">Text</option>
</select>
```

```
Select(driver.find_element_by_id('jform_params_usetext'))
.select_by_visible_text('Icons')
```

Fig. 2. A drop-down list in the log-in module page of Joomla! and a Python snippet to select the value “Icons”

III. Related Work

In the following, we present related studies that have improved upon the existing test automation approaches. In order to reduce the cost of test script implementation, many researchers have attempted to automatically generate test scripts for web applications [7], [8], [9], [10]. These approaches allow us to generate effective test scripts under certain circumstances.

Some researchers have sought to eliminate the fragility of test scripts to improve their maintainability. One effective solution is to make locators more robust by using metadata that is unlikely to change [11], [12], [13] or by using images of web elements as locators [14], [15]. Leotta et al. proposed a robust

[1] https://github.com/knukio/saner2022-experiment

[2] https://www.joomla.org/

[3] https://www.mantisbt.org/

XPath algorithm, ROBULA+ [11]. The study showed that ROBULA+ reduced the fragility of Selenium IDE locators by 63%, but these algorithms still depend on metadata and the structural information of web pages. Yandrapally et al. [12] proposed a technique to identify web elements by using other prominent elements on a web page (e.g., Click on “LabelA” near “LabelB”). The technique uses web elements that have labels and images as prominent elements. The study showed that, compared to existing tools, their scripts were more resilient to changes in metadata and the structure of pages. However, their technique can be fragile to changes in labels or images, and the expression of their test cases is limited by the labels. Some other researchers have also attempted to repair broken locators to handle locator fragility [16], [17], [18]. On the other hand, our approach aims for greater flexibility in terms of test case expression and for semantic-based identification of web elements by using NLP.

Yeh et al. [14] proposed SIKULI, a visual approach to automate operations on a screen by using images to identify web elements. The advantage of visual locators is that they are not dependent on the metadata or the structure of web pages, and target elements are easy to understand visually. Stocco et al. [15] proposed a technique called PESTO, which converts conventional locators to visual locators, but such visual locators are fragile to changes in user interfaces.

Several other researchers have leveraged NLP into testing or operating web applications. Thummalapenta et al. [19] proposed a technique to interpret test cases written in natural language. Their technique requires that a test step includes all necessary information for mechanically interpreting the step. Therefore, as distinct from our technique, targets to be operated must be described in the test cases in an identifiable form. DSL is effective in making test cases both unambiguity and readability. Dwarakanath et al. [20] proposed to introduce DSL to test cases for accelerating test automation. However, their technique also requires locators to uniquely identify web elements.

Lin et al. [21] proposed a technique to identify the topic of input fields for crawling-based test automation techniques, which can be applied to mine behavioral models, etc. They showed that their technique improved the accuracy of input topic identification by up to 22% compared to a rule-based approach. However, their technique only considers input fields and only identifies pre-trained topics. Pasupat et al. [22] proposed a machine-learning-based technique to convert a natural language command (e.g., clicking on the second article) into the web element to be operated on the page. Their technique can be applied to end-to-end testing, but many of the commands given in their study are indirect and difficult to interpret with their model. The conversion accuracy of the technique is therefore not high. On the other hand, our technique incorporate heuristic search to explore a system under test by limiting the target to automated testing. Bajammal et al. utilized NLP techniques for accessibility testing [23].

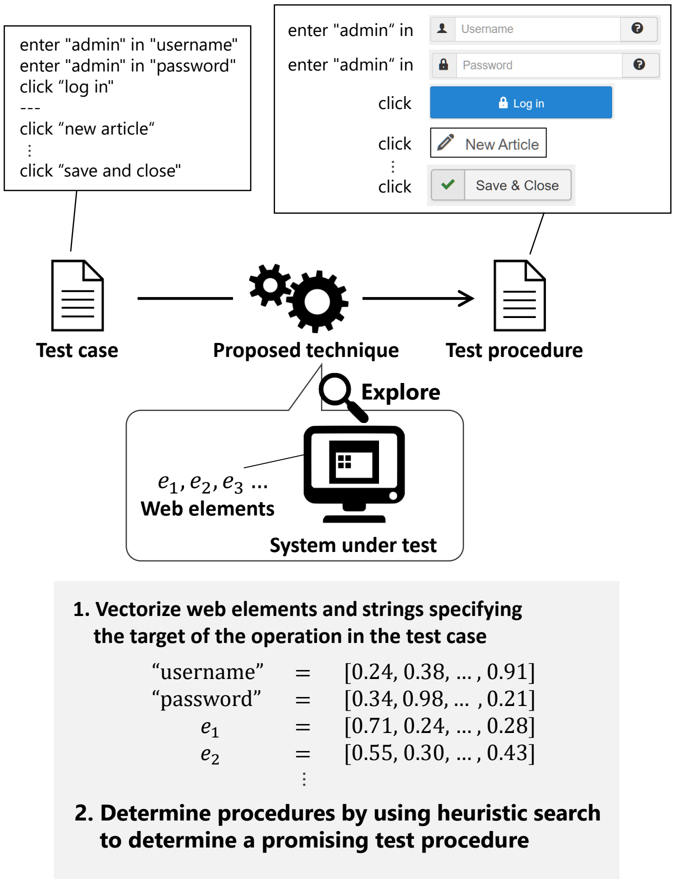

enter "admin" in "username"
enter "admin" in "password"
click "log in"
---
click "new article"
⋮
click "save and close"
enter "admin" in
Username
enter "admin" in
Password
click
Log in
click
New Article
⋮
click
Save & Close
Test case
Proposed technique
Test procedure
Explore
e_1, e_2, e_3 ...
Web elements
System under test
1. Vectorize web elements and strings specifying the target of the operation in the test case
“username” = [0.24, 0.38, ... , 0.91]
“password” = [0.34, 0.98, ... , 0.21]
e_1 = [0.71, 0.24, ... , 0.28]
e_2 = [0.55, 0.30, ... , 0.43]
⋮
2. Determine procedures by using heuristic search to determine a promising test procedure

Fig. 3. An overview of our approach

IV. Our Approach

The proposed technique interprets test cases written in a DSL close to natural language and determines a promising test procedure. A test case written in the DSL is a sequence of test steps. In this case, a test step is the smallest operational unit, as shown in the following example:

```
enter "admin" in "username"
```

Our technique interprets this test step as a operation that identify a web element that might be represented by “username” and enters “admin” into it.

Figure 3 shows an overview of our approach. The proposed technique interprets a test case and determines a test procedure by exploring the page transitions of the system under test. Our approach vectorizes web elements and strings specifying the target of the operation to understand their semantics by using NLP. Our approach also determines a test procedure by using a heuristic search to consider multiple test procedures for finding the most promising one.

Table I shows the specification of our DSL. Our DSL can currently handle only simple operations such as clicking, inputting, and selecting. *open* opens a specified URL in a browser and is generally called at the beginning of the test case. *click*, *enter* and *select* operate a certain web element.

TABLE I
THE SPECIFICATION OF OUR DSL

Operation	Description
open *url*	Open a specified *url*
click *target*	Click a button, link, etc., specified with *target*
enter *value* in *target*	Enter *value* in an input field specified with *target*
select *value* from *target*	Select *value* from a drop-down list specified with *target*
--- (page separator)	A separator between pages for the heuristic search algorithm explained in Section IV-B

These operations contain a *target* to specify a web element to be operated. Let us call such a string to specify a web element *target string*. Target strings can be any user-specified string, regardless of the implementation of a web page. *enter* and *select* include a *value* to be entered into the input field or selected from a drop-down list.

A. Vectorization

In order to determine a test procedure, we need to identify the web element that corresponds to the target string specified in the test case. For this purpose, we measure the similarity between the web elements and target strings. One important idea is to vectorize both web elements and target strings to represent their semantics.

Word embedding techniques (e.g., Word2Vec, fastText, GloVe, etc.) are often used to represent the semantics of a word or a sentence as a vector. We devise an approach to represent the semantics of a web element because web elements often contain information that is irrelevant to the semantics. First, we separately extract the values of attributes and visible texts from a web element. Visible texts include inner text and labels associated with the web element by `for` attributes. The `for` attribute specifies which web element a label is bound to, so we can identify the label that represents the element. The reason for separating attributes and visible texts is that we assume visible texts represent the semantics of the web element more directly and are more important than attribute values. Here, we ignore some attributes that are mainly used for visual layout such as `class`, `style` and so on. The following describes the procedure for preprocessing the obtained values.

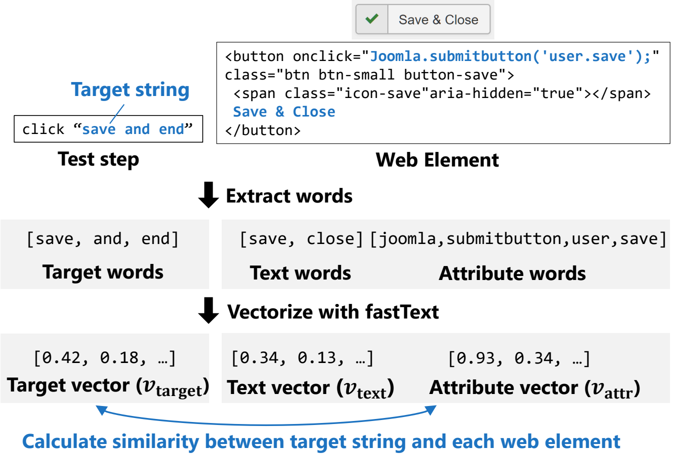
1) Split the values into words based on white space or symbols.
2) Convert the words into lowercase.
3) Remove stop words such as prepositions and articles.

Figure 4 shows an example of vectorizing a web element and a target string in Joomla!. In this example, we have a web element, a button labeled “Save & Close”. The text “Save & Close” that is rendered on the button is extracted as *text words*. Only the value of the `onclick` attribute is extracted as *attribute words*. The value of the `onclick` attribute is often important information because it is often the name of a JavaScript function and represents the feature of the web element. The other attributes (e.g., `class, area-hidden`) are ignored. Thus, we obtain the text words:

$$[\text{save}, \text{close}]$$

and the attribute words:

$$[\text{joomla}, \text{submitbutton}, \text{user}, \text{save}]$$



```
<button onclick="Joomla.submitbutton('user.save');"
class="btn btn-small button-save">
 <span class="icon-save"aria-hidden="true"></span>
 Save & Close
</button>
```

Fig. 4. An example of web element vectorization

Next, we convert these words into vectors representing their semantics. Among the available word-embedding algorithms, we selected fastText [24] because of its ability to handle unknown words using subword embedding. The fastText model has one million word vectors trained on Wikipedia 2017, UMBC WebBase corpus, and statmt.org news dataset[4]. Since web elements often contain abbreviations and proper nouns, we believe that a technique using subwords is suitable. The proposed technique vectorizes each word and takes their mean to obtain a text vector from the text words and an attribute vector from the attribute words. The text vector represents the semantics of the text words, and the attribute vector represents those of the attribute words.

In addition, we introduce tf-idf to weight each word. Intuitively, if the same word appears in a web element frequently, the word could be considered to uniquely represent the web element. However, if the same word appears across multiple web elements, the word would not be considered to represent the web elements. Therefore, although tf-idf is usually used to weight words among documents, we apply tf-idf to weight words among elements in this study. The weighting scheme is:

$$\text{tfidf}(w, e, E) = f_{w,e} \times \log \frac{N}{n_w}$$

where w is a word, E is a set of web elements, e ($\in E$) is a web element, $f_{w,e}$ is the frequency of word w in web element e, N is the total number of web elements, and n_w is the number of web elements in which w appears.

[4] https://fasttext.cc/docs/en/english-vectors.html

Let M be the number of text words, and w_i be the i-th unique word. Vector $\boldsymbol{v}_i$ is the resulting vector after applying fastText to w_i. The text vector $\boldsymbol{v}_{\text{text}}$ of a web element e is calculated by the weighted mean of $\boldsymbol{v}_i$ with tf-idf as the weight:

$$\boldsymbol{v}_{\text{text}} = \frac{\sum_{i=1}^{M} (\text{tfidf}(w_i, e, E) \times \boldsymbol{v}_i)}{\sum_{i=1}^{M} \text{tfidf}(w_i, e, E)}$$

The attribute vector $\boldsymbol{v}_{\text{attr}}$ is also calculated in the same way as above.

The method to vectorize target strings is almost the same as that to vectorize web elements. We extract target words from a target string and preprocess the target words in the same way as for web elements. We vectorize each word by using fastText and calculate the mean of vectors of the words without tf-idf. Thus, we obtain the target vector $\boldsymbol{v}_{\text{target}}$ from a target string.

Then, we can calculate the similarity between a target string and a web element by using $\boldsymbol{v}_{\text{target}}$, $\boldsymbol{v}_{\text{text}}$, and $\boldsymbol{v}_{\text{attr}}$. The similarity between a target string t and a web element e is calculated as a weighted mean of the two cosine similarities:

$$\text{similarity}(t, e) = \frac{\alpha \times \text{sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{text}}) + \text{sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{attr}})}{\alpha + 1} \tag{1}$$

where α (≥ 1) is a constant to add weight to the text words, and *sim* is the cosine similarity of two vectors.

B. Heuristic search algorithm

A web element that is most similar to the target string is considered to be operated in the test step. However, we do not determine a test procedure in order from the beginning by using only word-vector-based similarities calculated in Eq. (1). This is because it is uncertain whether the vector representation of the web element correctly represents its semantic.

The uncertainty of using only the NLP-based approach leads to the following problems. The first is that multiple target strings may be determined to be closest to the same web element. For example, suppose that there is a *password* field and a *confirm password* field on the web page. Two test steps have “password” and “confirm password” as target strings respectively in a test case. Suppose also that both strings are determined to be the most similar to the *password* input field. In this case, the two target strings are considered to specify the same web element. However, in general, different target strings should specify different web elements.

The second is that, if a web element identification fails at an early step of the test case, the subsequent test procedure cannot be determined correctly. Our technique requires a browser to render web pages to obtain web elements. Because the web pages may include static or dynamic page transitions, our technique needs to execute each test step each time to execute the expected page transitions. Once a test step executes an incorrect page transition, the subsequent test steps cannot reach the expected web page and will be useless.

To handle such uncertainty associated with the word-vector-based similarity, we create two heuristic search algorithms: page-level search and transition-level search. We use page-level search to handle the first problem and transition-level search to handle the second problem.

Page-level search: The page-level search algorithm contributes to accurately determining a test procedure closed to one web page. To clarify which test steps are closed to one web page, we introduce page separator `"---"` to our DSL. Strictly speaking, page separators ensure that all web elements operated in test steps between two separators are rendered on the web page when the page is loaded. It is helpful to know that the web elements specified by the target strings exist on the same page. The page-level search algorithm finds plausible permutations of web elements corresponding to target strings in test steps closed to a web page. First, the algorithm calculates the similarities of all possible pairs of a target string and a web element on a particular web page. Next, it calculates scores of permutations of web elements corresponding to target strings. Let us call this score a *page-wise score*. Here, more promising permutations have a higher page-wise score. When N test steps are executed on a web page, the page-wise score s_p is calculated as the mean of the sum of similarities between a target string and a web element:

$$s_p = \frac{1}{N} \sum_{i=1}^{N} \text{similarity}(t_i, e_i)$$

where t_i is the i-th target string, and e_i is a web element corresponding to t_i. Page-wise scores are calculated for all possible permutations. We note that the possible permutations are determined by the type of element (input field, button, or drop-down list) and the type of operation (*enter, click or select*). For example, if an operation is *enter*, the candidate web elements operated in the test step are limited to input fields. When a permutation is selected, a procedure to be operated on the web page is determined. We define the operation procedure as a *page-wise procedure*.

Figure 5 shows an example of page-level search. The web page has a *password* field e_1 and a *confirm password* field e_2. Two test steps are given for operating the input fields. The similarities between the target strings and the web elements can be calculated with the algorithm described in Section IV-A. There are two possible permutations here, in which “password” refers to e_1 and “confirm password” refers to e_2, or vice versa. In this example, the page-wise score of the former is 0.8, and that of the latter is 0.7, so the former is more plausible. When the former is selected, the page-wise procedure executes the operations in the order of e_1 and e_2. Without page-level search, both of the target strings would be considered to represent e_1. Thus, the page-level search algorithm contributes to determining a test procedure closed to a web page.

Transition-level search: We obtained multiple page-wise procedures with page-wise scores by applying the page-level search algorithm. However, it is insufficient to perform only the page-level search because the page-wise procedure with the highest page-wise score is not always correct. Transition-

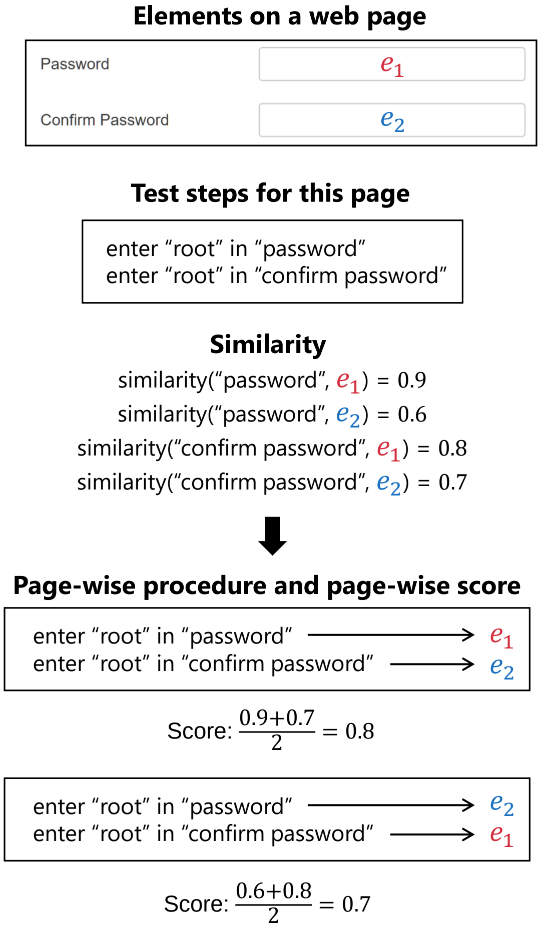

Elements on a web page
Password
e_1
Confirm Password
e_2
Test steps for this page
enter "root" in "password"
enter "root" in "confirm password"
Similarity
similarity("password", e_1) = 0.9
similarity("password", e_2) = 0.6
similarity("confirm password", e_1) = 0.8
similarity("confirm password", e_2) = 0.7
Page-wise procedure and page-wise score
enter "root" in "password" → e_1
enter "root" in "confirm password" → e_2
Score: $\frac{0.9+0.7}{2} = 0.8$
enter "root" in "password" → e_2
enter "root" in "confirm password" → e_1
Score: $\frac{0.6+0.8}{2} = 0.7$

Fig. 5. An example of page-level search

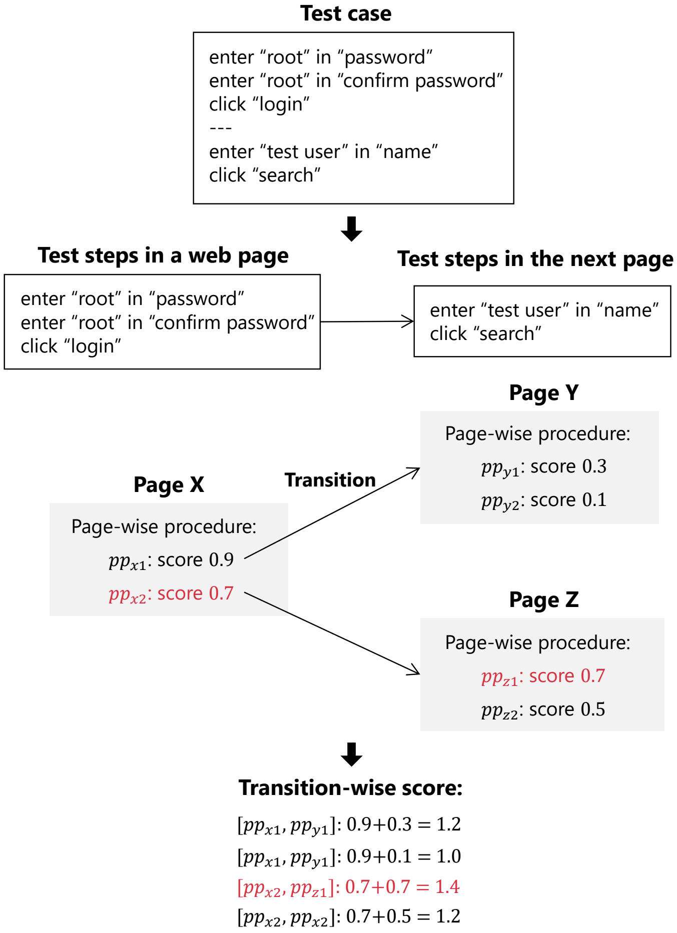

Test case
enter "root" in "password"
enter "root" in "confirm password"
click "login"
---
enter "test user" in "name"
click "search"
Test steps in a web page
enter "root" in "password"
enter "root" in "confirm password"
click "login"
Test steps in the next page
enter "test user" in "name"
click "search"
Page Y
Page-wise procedure:
pp_{y1}: score 0.3
pp_{y2}: score 0.1
Page X
Transition
Page-wise procedure:
pp_{x1}: score 0.9
pp_{x2}: score 0.7
Page Z
Page-wise procedure:
pp_{z1}: score 0.7
pp_{z2}: score 0.5
Transition-wise score:
$[pp_{x1}, pp_{y1}]$: 0.9+0.3 = 1.2
$[pp_{x1}, pp_{y1}]$: 0.9+0.1 = 1.0
$[pp_{x2}, pp_{z1}]$: 0.7+0.7 = 1.4
$[pp_{x2}, pp_{x2}]$: 0.7+0.5 = 1.2

Fig. 6. An example of transition-level search

level search explores multiple possible sequences of page-wise procedures. It contributes to determining a promising test procedure throughout the test case.

Figure 6 shows an example of transition-level search. We assume that a test case has five test steps, excluding the page separator. The first three test steps are executed on page X, and then the last two are executed on any of the pages following page X. In this case, there are two page-wise procedures, pp_{x1} and pp_{x2}, on page X, and they are the most promising procedures on page X. pp_{x1} makes a page transition from X to Y, and pp_{x2} makes a page transition from X to Z. We also assume that the two most promising page-wise procedures are obtained on page Y or Z after pp_{x1} or pp_{x2} is executed. It should be noted that pp_{x1} is likely to be wrong even though it is the most promising on page X. This is because both pp_{y1} and pp_{y2} have low page-wise scores. This means that page Y is not likely to have web elements specified by target strings "name" and "search". On the other hand, page Z seems to have web elements specified by the target strings because of its high page-wise score. Therefore, although it does not have the highest page-wise score, it is more promising to execute pp_{x2} on page X than pp_{x1}.

We determine the most promising procedure throughout the test case by considering the transition-wise scores. The transition-wise score is calculated as the sum of the page-wise scores up to the current web page. Generally, there are many possible page-wise procedures and page transitions, so it would take too much time to explore all the possible sequences within the page-wise procedures. Hence, we adopt the beam search algorithm, which explores a graph by expanding the most promising node in a limited set. The beam search has two parameters: a search width and a beam width. When the search width is W_s, we search the top W_s page-wise procedures at each step. Therefore, if the beam search considers N states at the current step, the number of states at the next step will be $W_s \times N$. When the beam width is W_b, the beam search prunes the states, leaving the W_b states with the highest transition-wise scores. Let M be the number of page-wise procedures executed up until the current state. The transition-wise score s_t is:

$$s_t = \sum_{i=1}^{M} s_{p_i}$$

where s_{p_i} is the page-wise score of the i-th page-wise procedures. Note that transition-level search is performed while dynamically exploring the application that is being tested. Since

TABLE II
A SUMMARY OF THE TARGET APPLICATIONS AND TEST CASES

Application	Version	Description	Feature category	# of test cases	# of total test steps
Joomla!	3.9	Content management system	Article management	10	90
			User managmement	4	44
			Menu management	7	64
MantisBT	2.24.1	Bug tracker	Issue management	8	80
			User management	6	54
			Others	12	121
Total				47	453

the state changes of the application during the exploration may affect the result of the transition-level search algorithm, it is desirable to initialize the state of the application each time a new page transition is attempted. To summarize this section, the transition-level search algorithm determines the procedure with the highest transition-wise score throughout the test case. The sequence of page-wise procedures with the highest transition-wise score is assumed to be the most promising for the test case.

V. Evaluation

We applied the proposed technique to test cases written in our DSL to evaluate the accuracy of our technique. The target applications in our experiment were Joomla! and MantisBT, which are non-trivial and popular open-source web applications. We chose these applications because they are feature-rich, have dynamic user interfaces, and are widely used in practice. We first prepared test cases manually for the two applications as inputs for our technique. To investigate the effectiveness of our technique, we addressed the following research questions:

RQ1. How accurately can our approach identify web elements and determine test procedures?

RQ2. Did the vectorization and the heuristic search contribute to determining test procedures?

RQ3. Can we apply our approach to testing in actual development?

A. Experimental setup

There are a large number of features in Joomla! and MantisBT, so we did not prepare test cases that cover all of the features. We therefore chose the key use cases of the applications by referring to their user manuals and then wrote test cases to cover them. As a result, we chose 21 use cases of Joomla! and 26 use cases of MantisBT. The use cases of Joomla! belong to the following three categories, described in the user manual for administrators [25]: article management, user management, and menu management. Because the user manual of MantisBT does not have an organized categorization like Joomla!, we assumed that there are three features that constitute the main functionality of MantisBT: issue management, user management, and others (management of projects, tags, custom fields, and global profiles). We then chose the use cases to cover them. We note that we exclude some use cases here requiring operations that our technique does not support. Table II shows a summary of the application and the test cases for our experiment.

We wrote 47 test cases to verify the chosen use cases. However, the way of writing test cases seems to depend on the person. In particular, the target string is dominant for the accuracy of our technique. Hence, we set the following rules for writing test cases. First, if the manual describes a specific procedure for the use case, we follow the manual as much as possible. When a use case has multiple ways in which it can be achieved, we chose one of them randomly. Second, it is not necessary to fill in all of the input fields in the test cases. In addition to the required input fields, we fill in one or more of the optional input fields. Furthermore, when we operate the same web pages in multiple test cases, we try to fill in different optional input fields from the input fields operated in the other test cases. Third, we limit the text used as target strings to one or a combination of the following:

- The text of nearby labels that are obviously related to a target element (e.g., a label right next to an input field)
- The text displayed in tooltips of a target element
- For buttons, the text displayed on the button
- For input fields, the default text specified by placeholder attributes
- For checkboxes and radio buttons, the text “checkbox” and “radio button”

The idea of following these rules is to reduce bias when creating test cases.

We applied the proposed technique with three different parameters. In this experiment, we set the same values for the search widths W_s and beam widths S_b and tried three different values: $W_s = W_b = 1$, 3, or 5. $W_s = W_b = 1$ means that the transition-level search was not performed. In other words, the page-wise procedure with the highest page-wise score was adopted on each web page. In this study, we attempted to treat text vectors and attribute vectors separately for better web-element embeddings. To confirm whether this approach worked well, we also examined the case where elements are represented by a single vector without distinguishing between text vectors and attribute vectors at the vectorization step. This means that all words in a web element are treated equally. In this case, we set W_s and W_b to 5, whether the vectors are distinguished or not. When distinguishing between the vectors,

TABLE III
HOW TEST STEPS ARE CONVERTED INTO PYTHON CODE

Operation	Python code
open *url*	`driver.get(`*url*`)`
enter *value* in *target*	`driver.find_element_by_`*type*`(`*locator*`).send_keys(`*value*`)`
select *value* from *target*	`Select(driver.find_element_by_`*type*`(`*locator*`).select_by_visible_text(`*value*`)`
`---` (page separator)	(This is not reflected in test scripts.)

TABLE IV
THE NUMBER OF SUCCESSFUL IDENTIFICATIONS

Search/Beam width	$W_s = W_b = 5$		$W_s = W_b = 3$		$W_s = W_b = 1$		$W_s = W_b = 5$	
Distinguish text/attribute	Yes		Yes		Yes		No	
	Test step	Test case	Test step	Test case	Test step	Test case	Test step	Test case
Joomla!	179 (90.4%)	13 (61.9%)	179 (90.4%)	13 (61.9%)	163 (82.3%)	9 (42.9%)	162 (81.8%)	9 (42.9%)
MantisBT	247 (96.9%)	19 (73.1%)	245 (96.1%)	18 (69.2%)	231 (90.6%)	15 (57.7%)	240 (94.1%)	18 (69.2%)
Total	426 (94.0%)	32 (68.1%)	424 (93.6%)	31 (66.0%)	394 (86.0%)	24 (51.1%)	402 (88.7%)	27 (57.5%)

$\alpha = 3$ is set as the weight text vector in Eq. (1).

We output a test procedure as a test script written in Python to confirm the test procedure determined by the proposed technique. We can determine locators for the test script according to the sequence of the page-wise procedures. This is because, if a web element operated at a certain test step is determined, we can obtain a locator from the implementation of the web element. Table III shows how each test step is converted into Python code. As shown in the table, a single test step is converted into a single line of python code. *type* in the Python code is `id`, `name` or `xpath` according to the locator type, and *locator* is a locator string obtained by the web element. *open* are directly converted into Python code because these operations do not include a target string. We do not ensure that the generated test scripts are always executable. This is because we do not consider an appropriate waiting time for rendering pages and the states of the system under test. We note that we are focusing on whether the proposed technique can identify correct web elements and determine a test procedure in this evaluation.

B. Results

We manually checked the test scripts to determine whether the proposed technique correctly identified web elements. Table IV shows the number of successful identifications. *Test step* in the table means the number of test steps that correctly identified web elements. Some test steps were duplicated because the test cases often included the same test steps. For example, the log-in steps were included at the beginning of all of the test cases. However, we counted the duplicate steps as being distinct, even if the test steps looked the same. This is because the XPaths of web elements may change depending on the state of web pages even though the web elements themselves may look the same. Furthermore, because of the uncertainty of our approach, the same test steps may be interpreted as different test procedures depending on the context. *Test case* in the table means the number of test cases in which all test steps in the test case identified web elements correctly. In other words, even if one of the test steps failed to identify the correct web element, it was counted as a failure.

RQ1: How accurately can our approach identify web elements and determine test procedures?

First, we explain the results when text vectors and attribute vectors were distinguished between. Table IV shows that, when $W_s = W_b = 5$, about 94% of test steps are successful in identifying web elements. Our approach also succeeded in identifying all the web elements that were operated in 68% of the test cases. No improvement in accuracy was observed for more search width or beam width. To answer RQ1, our technique can correctly identify web elements in up to about 94% of the test steps and identify all web elements in 68% of the test cases.

RQ2: Did the vectorization and the heuristic search contribute to determining test procedures?

When $W_s = W_b = 3$, the accuracy was slightly lower than in the case where $W_s = W_b = 5$. We can see that when $W_s = W_b = 1$ (without transition-level search), the accuracy was rather low compared to the other cases. This result means that the correct page-wise procedure is suggested in the top three by the page-level search in most cases. Therefore, we can say that page-level search worked well in our approach. By comparing between the cases $W_s = W_b = 1$ and 3, we can see that the transition-level search contributes significantly to the accuracy of our technique. Thus, we can say that the heuristic search algorithms compensate for the uncertainty of the NLP-based approach.

Next, we explain the results when text vectors and attribute vectors were not distinguished between. Furthermore, by comparing the case in which the text vector and attribute vector are distinguished and the case where they are not, we can see that distinguishing the vectors is effective for our approach. The result also indicates that text words represent the semantics of elements more directly than attribute words. Therefore, the approach weighting text vectors contributed to the accuracy

TABLE V

Average machine time (in seconds) required per test case.

	$W_s = W_b = 5$	$W_s = W_b = 3$	$W_s = W_b = 1$
Joomla!	107	66	25
MantisBT	94	56	21
Average	101	61	23

of the proposed technique. To answer RQ2, the vectorization approach and the heuristic search algorithms contribute to determining correct test procedures.

RQ3: Can we apply our approach to testing in actual development?

Table V shows the average execution time (in seconds) of our technique required per test case. We can see that the time is approximately proportional to the search width and beam width. The loading time of the fastText model, about 200 seconds, is not included here. We consider that this time to be negligible when the number of test cases to be processed at a time is long because our technique can process multiple test cases simultaneously after the model was loaded once. Most of the execution time of our technique is due to the dynamic exploration of the application by the transition-level search. However, we can make the exploration executed in parallel, in which case the execution time is not proportional to the number of test cases. We therefore assume that the time required to handle a large number of test cases is reasonable.

We obtained some results that illustrate the strengths of the NLP-based approach in real-world development. First, our technique was able to identify different web elements by the same test step depending on the context. In our experiments, the test step "`enter "test description" in "description"`" were able to correctly identify all three web elements in the situation of Figure 1. It showed the possibility of reusing the same test steps in different test cases and applications. Our technique was also able to identify web elements that did not seem to be directly related to target strings. The web element in Figure 2 did not include the words *display* and *labels* in the HTML document of the web element. Nevertheless, our technique was able to correctly identify this web element with the test step "`select "Icon" from "display labels"`", even though this web page contained 48 drop-down lists as candidates for the operation. This result indicates that the NLP-based approach works well in capturing the abstract semantics of web elements.

Figure 7 shows the relationship between the number of test steps in each test case and the number of executable test scripts generated from the rule described in Table III. The result is for the case when $W_s = W_b = 5$. There were between six and thirteen test steps in all of the test cases. In the figure, *All* means the total number of test cases. *Plausible* means the number of test cases in which all web elements were correctly identified. *Executable* means the number of generated test scripts that were executable from start to finish. This result shows that 56% of the plausible test cases were converted to executable test scripts. The main reason for unexecutable test scripts is that some web elements, especially in Joomla!, were not able to be operated by Selenium despite the locator having been correct. These failures depend on the implementation of the web page, not on locator errors. To operate these web elements in the test executions, for example, it is necessary to include a command to wait for a page load or to execute JavaScript directly through Selenium. Since it is uncertain whether our technique can execute the correct test procedure, we need to find a way to deal with the uncertainty, such as combining our approach with existing locator-based techniques. In addition, our current DSL does not consider assertions, which are essential for automated testing.

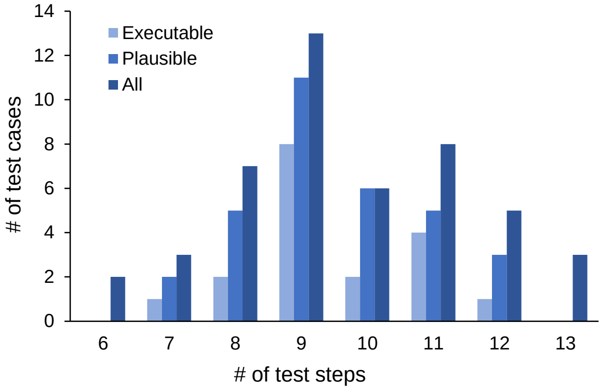


Fig. 7. The relationship between the number of test steps and that of plausible or executable test scripts

To answer RQ3, we believe that there is no practical problem with the execution time. By analyzing individual cases, we showed the potential of reusing the same test step for various test cases and applications. In addition, users without knowledge of programming may be able to write test cases since our technique does not require considering the detailed implementation of the system under test. However, it is necessary to improve the expressiveness of test cases in order to use them for real-world development.

VI. Discussion

A. What are the cases where our approach does not work?

In this study, we did not find relationships between the number of test steps and the success rate of determining correct test procedures. Intuitively, as the number of test steps increases, the probability of correct test procedures would be expected to decrease, but this was not the case in this experiment. This is probably because whether the web element is difficult to identify is a more significant factor than the number of steps. We therefore need to focus on individual failures for more detailed analyses. Suppose that identifying web elements fails at an early step of the test case, and an unexpected page transition is executed. In this case, the identification of subsequent web elements will also fail. Nevertheless, the result in Table IV shows that the accuracy of web element identifications is high, and the accuracy of identifying all web elements in a test case is low. This indicates that web element identifications often fail in the latter part of each test case. In this experiment, we found that web element identifications often failed, especially on the last web pages checked in the test case. This is because our technique does not benefit from the transition-level search on the last page. This means that, on the last page, the transition-level search cannot use the information of the next pages to choose the page-wise procedures. Therefore, our approach is prone to failing identifications of web elements at the end of the test case. We can say that this is a weakness of our heuristic search algorithms.

We found two patterns of web pages in which the NLP-based approach did not work well. The first was when there were multiple elements with the exact same label on the page. In particular, in our experiments, if the web elements have the same label, our technique cannot distinguish them by the rules for describing test cases. For example, the user management page of Joomla! has two "Users" links on a web page. Within the test case description rules, there is no other way to write other than "`click "Users"`" when we want to click on these elements. If there are meaningful words in the attribute text of the web elements, our technique may distinguish them by adding the words to the target string. Alternatively, by extending our approach so that positional information can be added to target strings, our technique may be able to handle the problem of the same label.

The second pattern where the proposed technique does not work is in the presence of an excessive number of elements on a web page. Our technique selects a web element to be operated from the web elements rendered on the browser. A large number of elements increases the likelihood of failing identification of a web element because there are more candidates for the operation. Note that here there may be many invisible elements in the HTML document despite only some of the web elements being visible on the screen. For example, some pages in Joomla! have such invisible elements. The web page to add menu items in Joomla! has five tab menus, but their contents are embedded in a single HTML document when the page is loaded. In addition, when the "Select" button is clicked on the page, a pop-up menu appears, which is also embedded in the HTML document. From the above, the actual number of web elements on the page is much larger than the number of visible elements. One of the solutions for this problem is to leverage heuristics into our technique, e.g., elements operated consecutively tend to be close to each other in terms of their position on the screen.

B. Limitations

Our approach limits the target applications and possible operations. Since the proposed technique uses Selenium internally, the technique can only operate web elements that Selenium can identify. For example, contents created by *canvas* feature or Flash cannot be operated. Current our DSL also cannot handle operations other than *click*, *enter*, and *select* (e.g., drag and mouse hover). These operations can be addressed by extending the proposed technique.

In addition, it is difficult to apply our approach to applications with ambiguous page transitions such as single-page applications. Our approach assumes that page separators are included in the test case properly. Therefore, to write appropriate test cases for such applications, users need to know when web elements will appear on the web page. Eliminating the page separator from the DSL and not performing the page-level search can solve this difficulty, but it will reduce the accuracy of our technique.

C. Threats to validity

The following presents two factors that undermine the external validity. The first is the scale of the experiment. We have only experimented with two applications, Joomla! and MantisBT, and the number of test cases is limited. Experiments on applications in other domains may yield different results from our experiment. We would obtain more accurate results by applying the proposed technique to a larger number of test cases. We tried to make the results more reliable by referring to the official manuals and selecting use cases from multiple functional categories to create test cases.

The second is the way of writing the test case. The way to write target strings is highly dependent on the accuracy of the NLP-based approach. In this study, we attempted to set rules for writing test cases. The rules are based on the assumption that test cases are written while observing the web page of the application under test. We wrote the test cases ourselves, so a certain amount of bias was inevitable, but we tried to reduce the bias by following the rules. In addition, since the created test cases are publicly available, it is possible to verify the validity of these test cases.

VII. Conclusion and Future Work

In this study, we proposed an approach to identify web elements from test cases that are close to natural language and determine a test procedure. Our approach uses an algorithm that combines NLP and heuristic search to obtain promising test procedures. To evaluate the proposed technique, we took test cases written in our DSL as input and applied our technique to two open-source web applications. The experimental results showed that our NLP-based approach and heuristic search contribute to determining correct test procedures.

As future work, we aim to extend the expressiveness of test case descriptions for practical use. In addition, we want to increase the number of applications and test cases for the evaluation and to demonstrate the effectiveness of our approach more generally.

References

[1] G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, 1997.

[2] H. K. N. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings of Conference on Software Maintenance*, 1989, pp. 60–69.

[3] F. Dobslaw, R. Feldt, D. Michaelsson, P. Haar, F. de Oliveira Neto, and R. Torkar, "Estimating return on investment for gui test automation frameworks," in *IEEE 30th International Symposium on Software Reliability Engineering*, 2019, pp. 271–282.

[4] (2004) Selenium. [Online]. Available: https://www.selenium.dev/

[5] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 141–150.

[6] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *IEEE International Conference on Software Testing, Verification and Validation*, 2016, pp. 180–190.

[7] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *35th International Conference on Software Engineering*, 2013, pp. 162–171.

[8] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 142–153.

[9] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 67–78.

[10] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, *Automatic Web Testing Using Curiosity-Driven Reinforcement Learning*, 2021, p. 423435.

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

[12] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 304–314.

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

[14] T. Yeh, T. Chang, and R. C. Miller, "Sikuli: Using gui screenshots for search and automation," in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, 2009, pp. 183–192.

[15] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Pesto: A tool for migrating dom-based to visual web tests," in *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 65–70.

[16] Z. Long, G. Wu, X. Chen, W. Chen, and J. Wei, "Webrr: Self-replay enhanced robust record/replay for web application testing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 14981508.

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

[18] H. Kirinuki, H. Tanno, and K. Natsukawa, "Color: Correct locator recommender for broken test scripts using various clues in web application," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019, pp. 310–320.

[19] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 881–891.

[20] A. Dwarakanath, D. Era, A. Priyadarshi, N. Dubash, and S. Podder, "Accelerating Test Automation through a Domain Specific Language," in *IEEE International Conference on Software Testing, Verification and Validation*, 2017, pp. 460–467.

[21] J. Lin, F. Wang, and P. Chu, "Using semantic similarity in crawling-based web application testing," in *IEEE International Conference on Software Testing, Verification and Validation*, 2017, pp. 138–148.

[22] P. Pasupat, T. Jiang, E. Liu, K. Guu, and P. Liang, "Mapping natural language commands to web elements," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 4970–4976.

[23] M. Bajammal and A. Mesbah, "Semantic Web Accessibility Testing via Hierarchical Visual Analysis," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021, pp. 1610–1621.

[24] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[25] (2020) Joomla! administrator's manual. [Online]. Available: https://docs.joomla.org/Portal:Administrators