NLP-assisted Web Element Identification Toward Script-free Testing

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

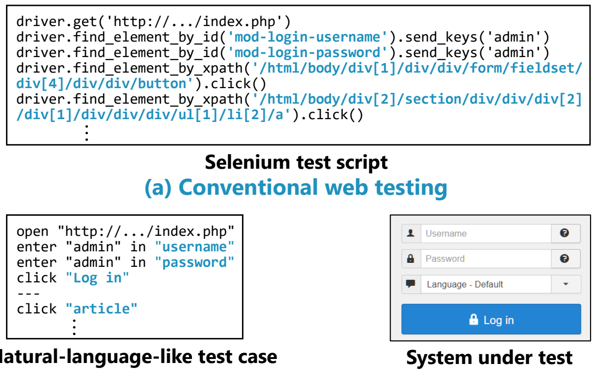
***Abstract*—End-to-end test automation is important in modern web application development. However, existing test automation techniques have challenges in implementing and maintaining test scripts. It is difficult to keep correct locators, which test scripts require to identify web elements on web pages. The reason is that locators depend on the metadata in web elements or the structure of each web page. One efficient way to solve the problem of locators is to make test cases written in natural language executable without test scripts. As the first step of script-free testing, we propose a technique to identify web elements to be operated and to determine test procedures by interpreting test cases. The test cases are written in a domain-specific language without relying on the metadata of web elements or 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 find promising test procedures. To evaluate our proposed technique, we applied it to two open-source web applications. The experimental results show that our technique successfully identified 94% of web elements to be operated in the test cases.**

***Index Terms*—Script-free Testing, Web Testing, Locator**

I. Introduction

In recent years, frequent software updates have become increasingly important in order to respond to rapid changes in market conditions. Developers need to confirm that their software works properly before its release. The cost of regression testing is dominant in software maintenance [1], [2]. Therefore, test automation is an important topic in software development.

Web application developers often use tools that automate end-to-end testing, and they need to implement and maintain test scripts. Dobslaw et al. [3] showed that, compared to manual testing, the initial implementation time was close to 90% of the total cost up until reaching the return on investment. One reason for the high cost of test script implementation is that most end-to-end test automation tools depend on the metadata in web elements or the structure of web pages. For example, Selenium [4], a de facto standard end-to-end test automation tool, requires locators that identify web elements. Some locators depend on metadata such as `id` or `name` attributes described in HTML documents, and other locators depend on structural information such as XPath. Developers often have to understand the detailed implementation of web pages to determine locators. In this way, test script implementation is obstructed by the dependencies on metadata and the structure of each web page.



```
driver.get('http://.../index.php')
driver.find_element_by_id('mod-login-username').send_keys('admin')
driver.find_element_by_id('mod-login-password').send_keys('admin')
driver.find_element_by_xpath('/html/body/div[1]/div/div/form/fieldset/
div[4]/div/div/button').click()
driver.find_element_by_xpath('/html/body/div[2]/section/div/div/div[2]
/div[1]/div/div/div/ul[1]/li[2]/a').click()
        ⋮
```

```
open "http://.../index.php"
enter "admin" in "username"
enter "admin" in "password"
click "Log in"
---
click "article"
       ⋮
```

Fig. 1. Difference between conventional web testing and script-free testing

The dependencies are also an obstacle to maintaining test scripts. Test scripts with locators are known to be fragile. Christophe et al. [5] investigated eight open-source software repositories that have Selenium test scripts. The study showed that 75% of Selenium test scripts changed more than once every nine commits (once every 2.05 days). Hammoudi et al. [6] examined breakages in Selenium IDE test scripts across 453 versions of eight web applications, and classified the causes of test breakages. Their experimental results showed that 73.62% of the breakages were caused by locators.

Another significant problem of end-to-end testing is the cost of writing 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 for all people involved in testing to understand the content of tests. The test cases also require maintenance. When both test cases and test scripts are present, developers need to keep them consistent. Thus, writing and maintaining test cases are costly activities, especially for fast-evolving applications.

In this study, we propose script-free testing, that is, a novel test automation approach distinct from conventional test script implementations using locators like that of Selenium. Figure 1 shows difference between conventional web testing and script-free testing. Conventional web testing requires test scripts depending on the implementation of a system under test. On the other hand, in script-free testing, developers can execute test cases that are close to natural language as automated tests. As the first step of script-free testing, we propose a technique to identify web elements to be operated in test cases. The test cases that we are focusing on are written in a domain-specific language (DSL) without relying on metadata in web elements or the structural information of web pages. Even non-developers can interpret our DSL as if it were natural language, so non-developers can communicate with developers by using the test cases and can also write them without knowledge of programming language. We leverage natural language processing (NLP) techniques to understand the semantics of web elements. We also create heuristic search algorithms to find promising test procedures. To evaluate our proposed technique, we applied it to two open-source web applications. The experimental results show that our technique successfully identified 94% of web elements to be operated in the test cases. The results also show that all the web elements to be operated are identified successfully in 68% of the test cases.

II. Existing Approaches

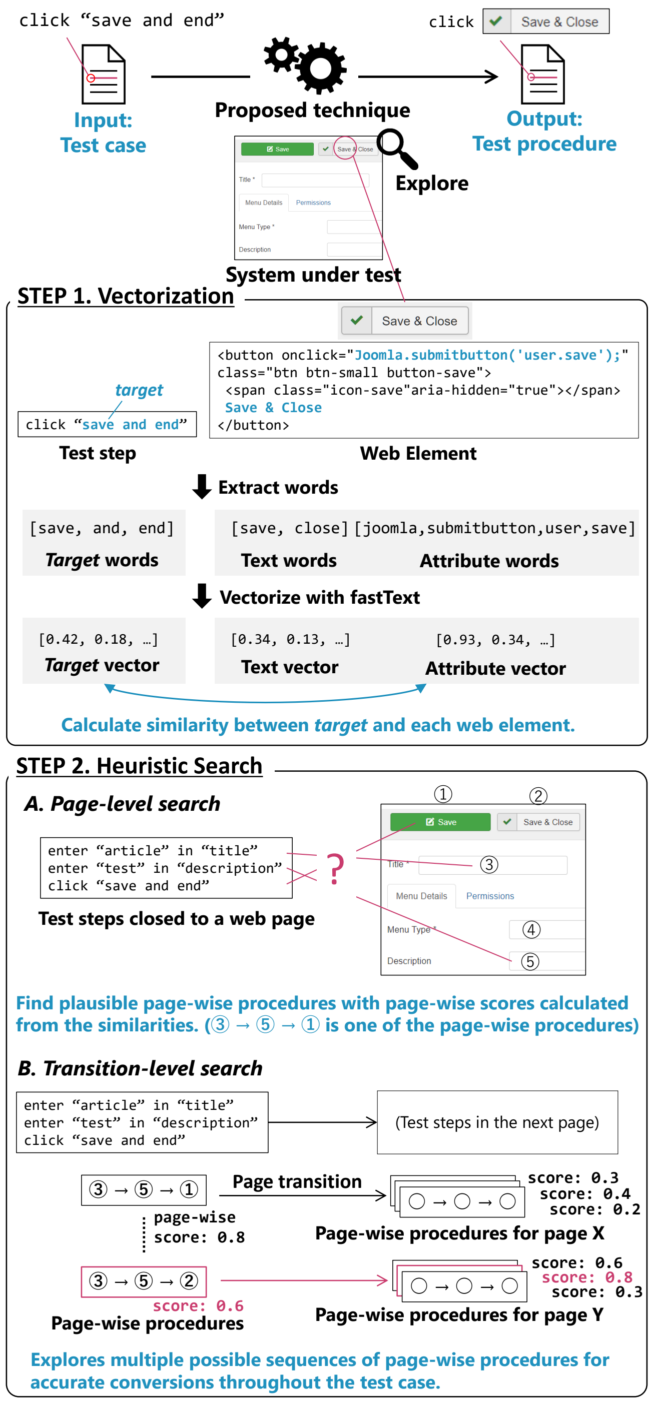
To reduce the cost of test script implementation, many researchers have attempted to automatically generate test scripts for web applications [7]–[9]. The approaches can generate effective test scripts for particular situations. However, it is difficult to generate test scripts that meet the developers' requirements because the techniques generate test scripts according to some criteria, not test cases written by the developers.

Some researchers have sought to address 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 [10], [11] or by using images of web elements as locators [12], [13]. Leotta et al. proposed a robust XPath algorithm, ROBULA+ [10]. The study showed that ROBULA+ reduced the fragility of Selenium IDE locators by 63%, but these algorithms still depend on metadata or the structural information of web pages.

Several other researchers have leveraged NLP techniques into testing or operating web applications. Lin et al. [14] proposed a technique to identify the topic of input fields for crawling-based test automation techniques, which can be applied to mine behavioral models. However, their technique only considers input fields, and only pre-trained topics are identified. Pasupat et al. [15] 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 given commands in their study are indirect and difficult to interpret with their model. In our technique, we define a DSL that is sufficient to describe test cases and incorporate heuristic search algorithms to find promising test procedures.

III. Our Approach

In order to achieve script-free testing, our first target is to identify web elements to be operated in test cases. The test



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

Fig. 2. An overview of our approach

cases are written in a DSL that is close to natural language. A test case written in the DSL is a sequence of test steps, which are atomic operational units. Figure 2 shows an overview of our approach. The proposed technique interprets a test case and determines a test procedure while exploring the page transitions of a system under test. First, our approach vectorizes web elements and strings specifying the target of the operation by using NLP techniques to understand their semantics. Next, our approach also determines a test procedure by using heuristic search algorithms to consider multiple possibilities of promising test procedures.

Table I shows the specification of our DSL. Since our DSL is close to natural language, it does not require knowledge of programming language to write. Our DSL can currently handle only simple operations such as clicking, inputting, and selecting. We introduce assertions to our DSL only to extend the expressions of test cases. The assertions do not affect the process of the heuristic search algorithms that are explained in Section III-B.

A. Vectorization

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

Word embedding techniques (e.g., Word2Vec) 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 include information that is irrelevant to the semantics. First, we extract the values of attributes and visible texts from a web element separately. Visible texts include inner text and labels associated with a web element by `for` attributes of HTML. The reason for separating attributes and visible text is that we assume visible text represents the semantics of the web element more directly and is more important than attribute values. Here, we ignore some attributes that are mainly used for visual layouts such as `class`, `style` and so on.

STEP 1. in Figure 2 shows an example of vectorizing a target string and a web element in the content management system Joomla! [16]. In this example, 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 an important piece of 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.

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*
---	A separator between pages for the heuristic search algorithm, explained in Section III-B
assert *string* exists	Assert that a specified *string* exists on the current page.
assert title is *string*	Assert that the title of the current page is a specified *string*

Next, we convert these words into vectors representing their semantics. Among the available word-embedding algorithms, we selected fastText [17] because of its ability to handle unknown words by using subword embedding. The fastText model has one million word vectors and is pre-trained on Wikipedia 2017, UMBC WebBase corpus, and statmt.org news dataset. Since web elements often include 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.

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.

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:

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

where $\text{tfidf}(w, e, E)$ is the weight of tf-idf calculated by a word w, a set of web elements E, and a web element e ($\in E$). The attribute vector $\boldsymbol{v}_{\text{attr}}$ is also calculated in the same way as the text vector.

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

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 $\text{similarity}(t, e)$ between a target string t and a web element e is calculated as a weighted mean of the two cosine similarities:

$$\frac{\alpha \times \text{cos\_sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{text}}) + \text{cos\_sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{attr}})}{\alpha + 1} \quad (1)$$

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

B. Heuristic search algorithm

The vectorization step can identify a web element that is most similar to the target string. However, the vector representation of the web element depends on its implementation, so the accuracy of the representation is uncertain. To handle such uncertainty, associated with the word-embedding-based similarity, we create two heuristic search algorithms: page-level search and transition-level search.

1) Page-level search: The page-level search algorithm contributes to determining a part of 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. It is necessary to know that the web elements specified by the target string exists on the same page. First, the algorithm calculates the similarities of all possible pairs of a target string and a web element on a particular web page. We consider permutations of web elements corresponding to target strings. When a permutation is selected, a procedure to be operated on the web page is determined. We define such a part of a test procedure closed to one web page as a *page-wise procedure*.

Next, the algorithm calculates scores of all possible page-wise procedures and then sorts by the score. Let us call this score a *page-wise score*. When N test steps are performed on a web page, the page-wise score s_p is calculated as the mean of the sum of similarities between each target string and each web element: $\frac{1}{N}\sum_{i=1}^{N} \text{similarity}(t_i, e_i)$ where t_i is the i-th target string, and e_i is a web element corresponding to t_i. The more promising page-wise procedure has higher page-wise scores.

2) Transition-level search: We obtained multiple page-wise procedures with page-wise scores by applying the page-level search algorithm. However, the page-wise procedure with the highest page-wise score is not always correct. In the example of STEP 2-B. in Figure 2, page-wise procedure ③ $\rightarrow$ ⑤ $\rightarrow$ ① has a higher page-wise score than page-wise procedure ③ $\rightarrow$ ⑤ $\rightarrow$ ②. However, considering the next page after the page transition, all the page-wise scores with the next page are low. Therefore, we can assume that the page-wise procedure ③ $\rightarrow$ ⑤ $\rightarrow$ ② is more plausible in this page.

Transition-level search explores multiple possible sequences of page-wise procedures. It contributes to determining a test procedure throughout the test case. We determine the most promising procedure 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 takes too much time to explore all of 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 algorithm has two parameters: a search width W_s and a beam width W_b. The algorithm searches 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 $W_s \times N$. The algorithm then 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 calculated by $\sum_{i=1}^{M} s_{p_i}$ where s_{p_i} is the page-wise score of the i-th page-wise procedure. The sequence of page-wise procedures with the highest transition-wise score is assumed to be the most promising test procedure for the test case. Now, we obtain the test procedure corresponding to the test case.

IV. Experiment

We applied the proposed technique to test cases written in our DSL to evaluate the effectiveness of our technique. We implemented our technique as a tool that determines the most promising test procedure and that generates a test script written in Python with Selenium to execute the test procedure. The generated test scripts contain locators to identify web elements corresponding to target strings in the test cases. We manually checked the locators to determine if the web elements are identified as intended in the test cases. Note that the fact that the locators are correct does not necessarily mean that the generated test scripts can be executed correctly. This is because our tool does not currently consider a waiting time for rendering web elements. Our tool, test cases, and the generated test scripts are publicly available: https://zenodo.org/record/4973219.

A. Experimental Setup

The target applications in our experiment were Joomla! and MantisBT [18], 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 tool. We chose 21 representative 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 [19]: article management, user management, and menu management. Because the user manual of MantisBT does not have an organized categorization like Joomla!, we assumed the following three features as constituting the main features of MantisBT: issue management, user management, and others (management of projects, tags, custom fields, and global profiles).

We wrote 47 test cases to verify the chosen use cases. The test cases have 453 test steps in total. However, the way to write test cases depends on the user. The target string in particular is dominant for the accuracy of our technique. Therefore, we limit texts used as target strings to rendered strings on the web browser and its type (button, checkbox, etc.). 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 W_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 treat text vectors and attribute vectors separately to identify web elements accurately. To see if 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, both when distinguishing and when not distinguishing between the vectors. When distinguishing between the vectors, $\alpha = 3$ is set as the weight text vector in Formula (1).

TABLE II
THE NUMBER OF SUCCESSFUL CONVERSIONS

Search/Beam width Distinguish text/attribute	$W_s = W_b = 5$ Yes		$W_s = W_b = 3$ Yes		$W_s = W_b = 1$ Yes		$W_s = W_b = 5$ 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%)

B. Results and Discussion

Table II shows the number of successful identifications. *Test step* in the table means the number of test steps correctly identifying web elements to be operated. *Test case* in the table means the number of test cases in which all test steps in the test case identify web elements correctly. In other words, even if one of the test steps failed to identify web elements in the test case, the test case was counted as a failure.

We will explain the results when text vectors and attribute vectors were distinguished between. The experimental results show that, when $W_s = W_b = 5$, about 94% of test steps and about 68% of test cases are successful in identifying web elements. When $W_s = W_b = 3$, the accuracy was slightly lower than in the case where $W_s = W_b = 5$. This result means that the correct page-wise procedure is proposed 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. We can see that when $W_s = W_b = 1$ (without transition-level search), the accuracy was rather low compared to the other cases. By comparing the case $W_s = W_b = 1$ and 3, we can see that the transition-level search contributes significantly to the accuracy of our technique.

Next, we will explain the results when text vectors and attribute vectors were not distinguished between. When $W_s = W_b = 5$, comparing the case where the two vectors were distinguished between and the case where they were not, distinguishing between the vectors increased the accuracy. The result indicates that distinguishing the vectors is effective for our approach and that text words represent the semantics of elements more directly than attribute words. Therefore, the approach weighting text vectors contributed to the accuracy of the proposed technique.

V. Conclusion and Future Work

In this study, we proposed script-free testing, that is, a novel test automation approach distinct from conventional test script implementation using locators. If test cases written in natural language can be executed as automated tests, it will not only make testing more efficient but will also allow more people to participate in software development. We also proposed an approach to identify web elements to be operated in test cases that are close to natural language, as the first step of script-free testing. Our approach combines NLP techniques and heuristic search algorithms to determine a promising test procedure throughout the test case. The experimental results showed that the proposed technique can identify web elements accurately and that our NLP-based approach and heuristic search algorithms contribute to the accuracy.

In the future, we are going to increase operations of our DSL to improve the expressiveness of test case descriptions and apply our approach to real-world software development. To demonstrate the practical effectiveness of our approach, we are also going to evaluate our approach by having experts use it and compare it with existing test script implementation techniques on robustness, readability, cost-effectiveness, and so on.

References

[1] G. Rothermel and M. Harrold, “A safe, efficient regression test selection technique,” *ACM Trans. Software Eng. Method.*, vol. 6, no. 2, 1997.
[2] H. K. N. Leung and L. White, “Insights into regression testing (software testing),” in *ICSM*, 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 *Proc. ISSRE*, 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 *Proc. ICSME*, 2014, pp. 141–150.
[6] M. Hammoudi, G. Rothermel, and P. Tonella, “Why do record/replay tests of web applications break?” in *Proc. ICST*, 2016, pp. 180–190.
[7] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, “Guided test generation for web applications,” in *Proc. ICSE*, 2013, pp. 162–171.
[8] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, “Diversity-based web test generation,” in *Proc. ESEC/FSE*, 2019, pp. 142–153.
[9] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, “Leveraging existing tests in automated test generation for web applications,” in *Proc. ASE*, 2014, pp. 67–78.
[10] 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.
[11] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, “Robust test automation using contextual clues,” in *Proc. ISSTA*, 2014, pp. 304–314.
[12] T. Yeh, T. Chang, and R. C. Miller, “Sikuli: Using gui screenshots for search and automation,” in *Proc. UIST*, 2009, pp. 183–192.
[13] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, “Pesto: A tool for migrating dom-based to visual web tests,” in *Proc. SCAM*, 2014, pp. 65–70.
[14] J. Lin, F. Wang, and P. Chu, “Using semantic similarity in crawling-based web application testing,” in *Proc. ICST*, 2017, pp. 138–148.
[15] P. Pasupat, T. Jiang, E. Liu, K. Guu, and P. Liang, “Mapping natural language commands to web elements,” in *Proc. EMNLP*, 2018, pp. 4970–4976.
[16] (2021) Joomla! [Online]. Available: https://www.joomla.org/
[17] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching word vectors with subword information,” *TACL*, vol. 5, pp. 135–146, 2017.
[18] (2021) MantisBT. [Online]. Available: https://www.mantisbt.org/
[19] (2021) Joomla! administrator’s manual. [Online]. Available: https://docs.joomla.org/Portal:Administrators