Efficient Test Script Generation and Maintenance for Web Applications

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2023

Hiroyuki KIRINUKI

Abstract

The speed of business has become increasingly important in recent years, leading to a need for quick improvements to services. To achieve this, it is necessary to shorten the software release cycle. Testing is an essential part of the software development process, and regression testing is a critical component of this. Reducing the costs of regression testing is a major challenge in shortening the release cycle. End-to-end testing is a crucial part of regression testing and involves testing applications with a graphical user interface (GUI) such as interactions between web browsers and users.

Test automation is an effective way to reduce the cost of regression testing. Testing tools such as Selenium are commonly used to automate end-to-end testing for web applications. End-to-end test automation involves creating test scripts that describe the steps in the test procedures.

The first challenge related to test scripts is that locators for identifying web elements are fragile because they depend on the structure of web pages and the attributes of the web elements, requiring a significant amount of effort to maintain. The second challenge is the high cost of implementing maintainable test scripts. This dissertation presents three studies that address these challenges and improve end-to-end test automation for web applications.

The first study proposes an approach called COLOR for repairing broken locators in accordance with software updates. COLOR uses various properties from web pages as clues and evaluates their reliability. Our experimental results from four open-source web applications show that COLOR consistently presents the correct locator with an accuracy ranging from 77% to 93% in the first place and is more robust against page layout changes compared to structure-based approaches.

The second study proposes an approach for generating modularized test scripts to improve their maintainability. The technique extracts operations useful for test automation from test logs and generates test cases that cover the features of an application by analyzing its page transitions. The approach was evaluated using test logs from four testers, showing that it can generate more complete methods than an existing approach. Our empirical evaluation also showed that the approach can reduce the time required to im-

plement test scripts by 48% compared to manual implementation. This study contributes to reducing implementation and maintenance efforts.

The third study proposes a technique to identify web elements to be operated on a web page by interpreting natural-language-like test cases. The test cases are written in a domain-specific language that is independent of the metadata of web elements and the structural information of web pages. Natural language processing techniques are used to understand the semantics of web elements, and heuristic search algorithms are used to explore web pages and find promising test procedures. The technique was applied to test cases for two open-source web applications, with the results showing that it was able to successfully identify 94% of web elements to be operated on and all the web elements in 68% of the test cases. This study contributes to the easy implementation and maintenance of test scripts for various users.

List of Publication

Journal

[1-1] Hiroyuki Kirinuki and Haruto Tanno: “Automating End-to-End Web Testing via Manual Testing”, Journal of Information Processing, Vol. 30, pp. 294-306, 2022.

[1-2] Hiroyuki Kirinuki, Haruto Tanno, and Katsuyuki Natsukawa: “Recommending Correct Locator for Broken Test Scripts using Various Clues in Web Application”, Computer Software, Vol. 36, No. 4, pp. 3-17, 2019.

International Conference

[1-3] Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto, “Web Element Identification by Combining NLP and Heuristic Search for Web Testing”, IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 1044-1054, March 2022.

[1-4] Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto, “NLP-assisted Web Element Identification Toward Script-Free Testing”, In Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 639-643, September 2021.

[1-5] Hiroyuki Kirinuki, Haruto Tanno and Katsuyuki Natsukawa, “COLOR: Correct Locator Recommender for Broken Test Scripts using Various Clues in Web Application”, IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 310-320, February 2019.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The speed of business has become increasingly important, and the need for quick service improvement is essential. To achieve this, it is necessary to shorten the software release cycle. Software development involves testing to ensure that the software is working properly before releasing it. In testing, it is necessary to confirm not only that newly added functions work properly, but also that existing functions work as before. Testing to confirm that an existing feature still works as before is called regression testing.

Reducing the costs required for regression testing is a major issue in shortening the release cycle. It has been reported that regression testing accounts for 25% of the total costs required for development in the case of an enhancement or modification project [1]. When aiming to shorten the release cycle, regression testing needs to be performed many times in a short period, which makes the costs of regression testing a significant portion of the total development costs. In today's fast-paced business environment, spending a large amount on regression testing may lead to missed business opportunities. Automation of testing is an effective means of reducing the cost of regression testing, and various research has been conducted on this topic.

1.2 End-to-End Test Automation for Web Application

Regression testing includes end-to-end testing, which is testing throughout the entire system. End-to-end testing for applications with a graphical user interface (GUI) often involves simulating interactions between the application and its users. This type of testing is important because it helps to ensure that different parts of the system are working together correctly.

Test automation tools such as Selenium [2] are commonly used to automate end-to-end tests for web applications. In the case of a web application, the test automation tool is typically used to automate browser operations and validate the results of those operations. To perform end-to-end testing automatically, it is necessary to write test scripts that describe the test procedures.

There are two approaches to automating end-to-end testing: *record & replay* and *programming* [3]. Record & replay tools like Selenium IDE first record the operations performed by testers on a web browser and convert these operations into test scripts. When the test scripts are executed, the tool replays the recorded operations as they were performed. This approach makes it easy to implement test scripts by simply following the test procedures while operating a web browser. However, the generated test scripts are not modularized and have low readability, which becomes an issue when maintaining them over a long period.

In the programming approach, developers implement test scripts in general-purpose programming languages such as Java and Python, using libraries (e.g., Selenium WebDriver) to operate a web browser. Only skilled developers can implement properly modularized and readable test scripts. Implementing such scripts means that developers can maintain them more easily compared to using record and replay tools [4].

In testing web applications, each step of the test script is often represented by a combination of the type of operation to be performed, the input value, and the target of the operation. For example, a test script for a login form might include steps to enter a username and password, click the "Login" button, and verify that the user is redirected to the correct page. One common approach to specifying targets in web application testing is to use locators, which are a way to uniquely identify a web element on a web page. For example, the "id" and "name" attributes of a web element can be used as locators. XPath, which shows the position of web elements within the HTML document object model (DOM), can also be used as a locator. There is also a technique called image-based locator [5], which identifies web elements by matching images.

The implementation of test scripts is known to be costly, with initial implementation time accounting for nearly 90% of the total cost to reach a return on investment [6]. Test scripts include locators for specifying the web elements to be operated on, but these locators frequently change as applications evolve, requiring maintenance of the test scripts [7,8]. If significant efforts are required for this maintenance, the cost of implementing the test scripts may not be recouped. Thus, generating maintainable test scripts and efficiently maintaining them are important challenges.

1.3 Related Work

1.3.1 Test script generation

In the following, we present related studies that have improved upon existing automatic test generation approaches.

Crawling-based techniques for end-to-end test script generation have been proposed to minimize the cost of end-to-end testing [9–11]. These techniques generate test scripts that cover all features of an application through dynamic exploration. GUI ripping is an approach that automatically traverses GUIs and generates their model for regression testing [12, 13]. Some studies also try generating test scripts by extending existing test scripts [14–16] or using reinforcement learning [17]. However, the test scripts generated with these techniques are not complete and require modifications such as adding assertions by the developer, and maintainability is not considered.

In order to generate useful test scripts, it is necessary to follow the test cases written by developers or the use cases of the application. In addition, the maintainability of the generated test scripts should be considered, as developers need to maintain the test scripts as the application evolves. Thus, it is difficult to incorporate existing approaches into continuous development as they are.

1.3.2 Test script maintenance

Some researchers have attempted to overcome the fragility of locators. While structure-based locators are generally considered more fragile than attribute-based locators, developers often use XPath as a locator since elements do not always have an id or name. Leotta et al. proposed the robust XPath algorithms ROBULA (ROBUst Locator Algorithm) [18] and ROBULA+ [19]. These algorithms focused on the expressiveness of XPath. ROBULA starts with a generic XPath returning all nodes (`"//*"`) and refines it using heuristic XPath specialization steps to increase robustness until only the element of interest is selected. ROBULA+ enhances ROBULA with additional heuristics. Yandrapally et al. [20] presented an approach to robustly identify elements using contextual clues without recording anything about the internal representation. Their approach identifies an element relative to other prominent elements on the page (e.g., Click on "LabelA" near "LabelB").

Yeh et al. [5] 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 structure of web pages, and the target elements are easy to understand visually. Stocco et al. [21] proposed a technique called PESTO that converts conventional locators to visual locators, but such visual locators are fragile to changes in user interfaces.

The problem with these approaches is that there is a limit to robustness due to the difficulty in predicting how the software will be updated. Since the eligible locator varies depending on the characteristics of the application, the approach of pre-determining the locator lacks flexibility.

The other approach to dealing with locator changes is repairing locators in accordance with software updates, which is more flexible than the previous approach if the repair accuracy is high. Leotta et al. proposed an approach to repair locators using a multi-locator algorithm [22]. They use five structure-based locators: FirePath absolute, FirePath relative ID-based, Selenium IDE, Montoto, and ROBULA+. The multi-locator algorithm selects the best locator among these five locators based on a voting procedure that assigns different weights to different locators. If this procedure succeeds and the locator with the most votes is able to identify the element, all other broken locators can be automatically repaired. However, this approach has a theoretical limit: if all locators are broken, the multi-locator algorithm cannot correctly identify the element.

Choudhary et al. [23] proposed the WATER (Web Application TEest Repair) approach to repair locators by mainly using the Levenshtein distance of XPath. They assume that two elements are likely to be the same across two releases if the Levenshtein distance between their XPath is small. WATER also takes into account five attributes in case the Levenshtein distance is the same. The problem with this approach is that it is fragile when the page layouts of the application are drastically modified. In such cases, the XPath of the element may be significantly changed and the locator may not be correctly repaired even using WATER. Hammoudi et al. proposed an incremental test repair approach called WATERFALL that applies WATER iteratively across a sequence of fine-grained versions of a web application [24]. Both methods are related to general program repair techniques [25–29], but these methods focus on locators in test scripts.

Yandrapally et al. [30] proposed an approach to modularize test scripts automatically to improve their maintainability of test scripts generated by a record and replay tool. Their approach identifies operations to be modularized by analyzing the test scripts and the document-object model of an application. Their experimental results showed that the number of steps can be reduced by 49–75% by converting parts of the test scripts into a subroutine. However, this approach does not consider how the subroutines follow the actual use cases of the application.

1.3.3 NLP-based approaches

Several researchers have leveraged natural language processing (NLP) techniques in testing or operating web applications. Manipulating web applications in natural language would free us from the problem of implementing test scripts.

Thummalapenta et al. [31] 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 it. Dwarakanath et al. [32] proposed using DSL in test cases to accelerate test automation. However, their technique also requires locators to uniquely identify web elements.

Lin et al. [33] 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. [34] 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, leading to low conversion accuracy. Bajammal et al. utilized NLP techniques for accessibility testing [35].

1.4 Contribution of Dissertation

We believe that the challenges to be addressed regarding end-to-end test scripts that have not been solved by existing studies are as follows:

- Locators are dependent on the structure of web pages and the attributes of web elements, which can decrease the maintainability of test scripts. Additionally, existing locator repair techniques rely on structural information, resulting in low accuracy.
- The implementation of test scripts is costly, and existing test script generation techniques do not take maintainability into account.
- It has not yet been possible to directly execute test cases written in natural language.

We present three studies that we have conducted to address these challenges:

1. Locator recommendation using various clues for test script maintenance
2. Maintainable test script generation via manual testing
3. Web element identification using NLP and heuristic search

The first study uses various information obtained from web pages to perform more accurate locator recommendation. This research contributes to reducing the maintenance cost of test scripts. The second study reduces implementation and maintenance efforts by

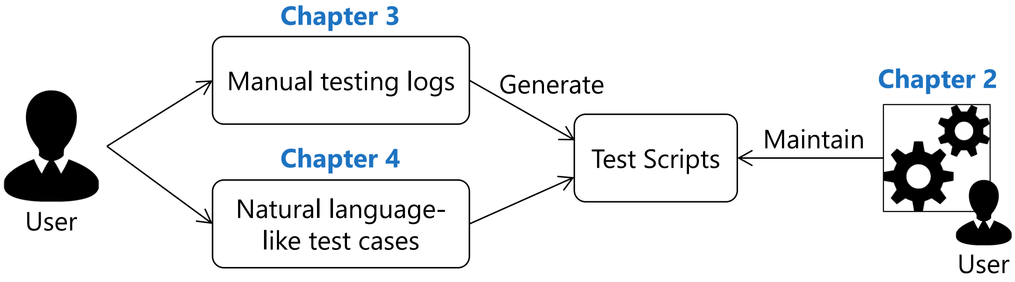


Figure 1.1: Overview of dissertation

automatically generating modularized test scripts based on application use cases extracted from manual testing logs. The third study identifies web elements using NLP and heuristic search to generate test scripts from natural language-like test cases. The third study is a first step towards realizing executable test cases using natural language. In this study, instead of directly executing test cases, we convert natural language-like test cases into test scripts using NLP and heuristic search. If automated testing could be completed with only natural language instructions, the first and second research would not be necessary. However, it is not possible to achieve this with current technology, so we believe that support for generating and maintaining test scripts using conventional locators is also necessary.

The second and third techniques both generate test scripts, but they have different strengths. The second technique is effective at generating test scripts that cover a wide range of possible use cases, whereas the third technique is better suited for generating test scripts for specific scenarios specified by users. One of the advantages of these techniques is that manual testing and test case creation can be done without programming knowledge. Furthermore, they are often performed even without automated testing, so there is no need to prepare for generating test scripts.

Although all the techniques proposed in this dissertation are targeted at web applications, the ideas could potentially be applied to applications with GUI (e.g., mobile applications).

1.5 Overview of Dissertation

This dissertation is composed of the following three studies. Figure 1.1 shows the overview of the dissertation and the three studies.

1.5.1 Locator recommendation using various clues for test script maintenance

As mentioned in the previous sections, test scripts frequently need to be changed as applications are updated. The costs of modifying these test scripts are a major obstacle to test automation due to their fragility. In particular, locators in test scripts are prone to change. Some prior studies attempted to repair broken locators using structural clues, but these approaches usually cannot handle radical changes to page layouts.

In this research, we propose a novel approach called COLOR (correct locator recommender) to support repairing broken locators in accordance with software updates. COLOR uses various properties as clues obtained from screens (i.e., attributes, texts, images, and positions). We examined which properties are reliable for recommending locators by examining changes between two release versions of software, and the reliability is adopted as the weight of a property. Our experimental results obtained from four open-source web applications show that COLOR can present the correct locator in first place with an accuracy of 77% – 93% and is more robust against page layout changes than structure-based approaches.

1.5.2 Maintainable test script generation via manual testing

The cost of implementing and maintaining test scripts is a major obstacle to the introduction of test automation. In addition, many testing activities, such as exploratory testing, user-interface testing, and usability testing, rely heavily on manual efforts. We propose an approach to generate test scripts from manual testing recorded by our tool.

The generated test scripts leverage the page-object pattern, which improves the maintainability of test scripts. To generate page objects, our approach extracts operations as methods useful for test automation from the test logs. Our approach also generates test cases that cover the features of an application by analyzing its page transitions. This enables the generation of test scripts that are close to the actual use cases that were difficult to generate using existing techniques.

We evaluated whether our approach could generate complete test scripts from test logs obtained from four testers. Our experimental results indicate that our approach can generate a greater number of complete methods in page objects than a current page-object generation approach. We also conducted an empirical evaluation of whether our approach can reduce the cost of implementing test scripts for real systems. The result showed that our approach reduces the time required to implement test scripts by about 48% compared to manual implementation.

1.5.3 Web element identification using NLP and heuristic search

It can be challenging to determine and maintain the locators needed by test scripts to identify web elements on web pages. This is because locators depend on the metadata of web elements and the structure of each web page. A potential solution to this problem is to allow natural language test cases to be executed without the need for test scripts.

In this study, we propose a technique for identifying 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 is independent of the metadata and structural information of web elements and web pages. We use natural language processing techniques to understand the semantics of web elements and create heuristic search algorithms to explore web pages and identify promising test procedures.

To evaluate the effectiveness of our technique, we applied it to test cases for two open-source web applications. The results show that our technique was able to successfully identify approximately 94% of the web elements to be operated in the test cases. Furthermore, our approach was able to identify all the web elements that were operated in 68% of the test cases.

1.6 Chapter Organization

Chapter 2 describes locator recommendations for test script maintenance, Chapter 3 addresses maintainable test script generation via manual testing, Chapter 4 provides web element identification using NLP and heuristic search, and in Chapter 5, we summarize the findings of this dissertation and discuss future challenges.

Chapter 2

Locator Recommendation Using Various Clues for Test Script Maintenance

2.1 Introduction

Developers need to develop software quickly and release it in accordance with changes in market conditions. Before releasing it, they perform testing to confirm that the software works properly. This includes not only ensuring that new features work properly, but also confirming that existing features continue to work as expected. This type of test for existing features is called regression testing, which can account for a significant portion of software maintenance costs [36, 37]. Therefore, reducing the costs of regression testing and shortening the release cycle are major challenges in software development. In web application development, test automation tools such as Selenium [2] are commonly used to automate end-to-end tests and make regression testing more efficient.

However, maintaining these test scripts can be time-consuming, as developers often need to modify existing test scripts in response to software updates. Several methods have been proposed to maintain end-to-end test scripts on various platforms [4, 8, 38–40]. For example, Leotta et al. [4] developed test scripts for six open-source web applications to evaluate the costs of automated web testing approaches. As part of their evaluation, they examined the costs of test script maintenance on the new release version of the software. They developed a total of 196 Selenium IDE test scripts on the old version of the software and then had to repair 180 of those scripts on the new version of the software. In other words, about 92% of the test scripts broke between the two releases. Christophe et al. [7] investigated eight OSS repositories that included Selenium test scripts and found that 75% of Selenium test scripts were modified at most three times before the corresponding file was deleted.

The main reason for the fragility of test scripts is known to be the use of locators. Selenium uses locators to find and match the web elements on a page that need to interact with each other. Hammoudi et al. [8] examined the breakages of Selenium IDE test scripts across 453 versions of eight web applications and classified the causes of test breakages. Their experimental results revealed that 73.62% of the breakages were caused by locators. Therefore, in this study, we focus on problems related to broken locators.

Most web testing tools (e.g., Selenium) use attribute-based locators (e.g., id, name, etc.) or structure-based locators (e.g., XPath). A problem with these locators is that they may be changed even by a trivial software update, such as changing a page layout or attributes of web elements. There have been studies that aim to support the maintenance of test scripts for web applications [41–43]. Some of these studies attempted to automatically repair broken test scripts by using structural clues. The core concept of these studies is that web elements located close to each other across two releases are more likely to be the same. However, this concept does not always work well because page layouts can change frequently in some recent web applications. Some testing tools (e.g., Sikuli [5]) use images to find and match web elements. The advantage of image-based testing is that it can be applied to any kind of application and does not require recording anything about the internal representation. Stocco et al. [21] proposed a technique to migrate DOM (document object model)-based testing to image-based testing. Image-based testing is more robust than DOM-based testing in some situations, but it is vulnerable to changes in screen appearance [44].

In this study, we propose a novel approach called COLOR (correct locator recommender) to support the automatic repair of broken locators in accordance with software updates. COLOR uses various properties as clues obtained from screens (i.e., attributes, texts, images, and positions) to overcome the weaknesses of structure-based and image-based approaches. COLOR can recommend correct locators even if page layouts change drastically.

First, we clarify which properties are prone to change between two releases and are reliable for recommending locators. Second, to evaluate the accuracy of the recommendations, we applied COLOR to four open-source web applications that include page layout changes. The experimental results show that COLOR can present the correct locator in the first place with 77% - 93% accuracy and above third place with 82% - 95% accuracy. Furthermore, we prove that COLOR is more robust against page layout changes than existing structure-based approaches by comparing it with a prior method.

The main contributions of this study are as follows:

- We propose a novel approach called COLOR that uses multiple properties obtained from a screen as clues. COLOR is more robust than structure-based approaches against complex changes, including page layout changes.
- We clarify which properties are reliable for recommending locators by examining 699 locators across two release versions of applications.
- We applied COLOR to broken locators to demonstrate its effectiveness and superiority over structure-based approaches. We also analyzed its results and characteristics.

2.2 Motivating Example

2.2.1 Sample application and test script

We use Joomla![1], an open-source content management system, as an example. We assume that a user logs in as an administrator in Joomla! version 1.5. To do this, the user first clicks on the "Administrator" link on the top page, which takes them to the login page. The

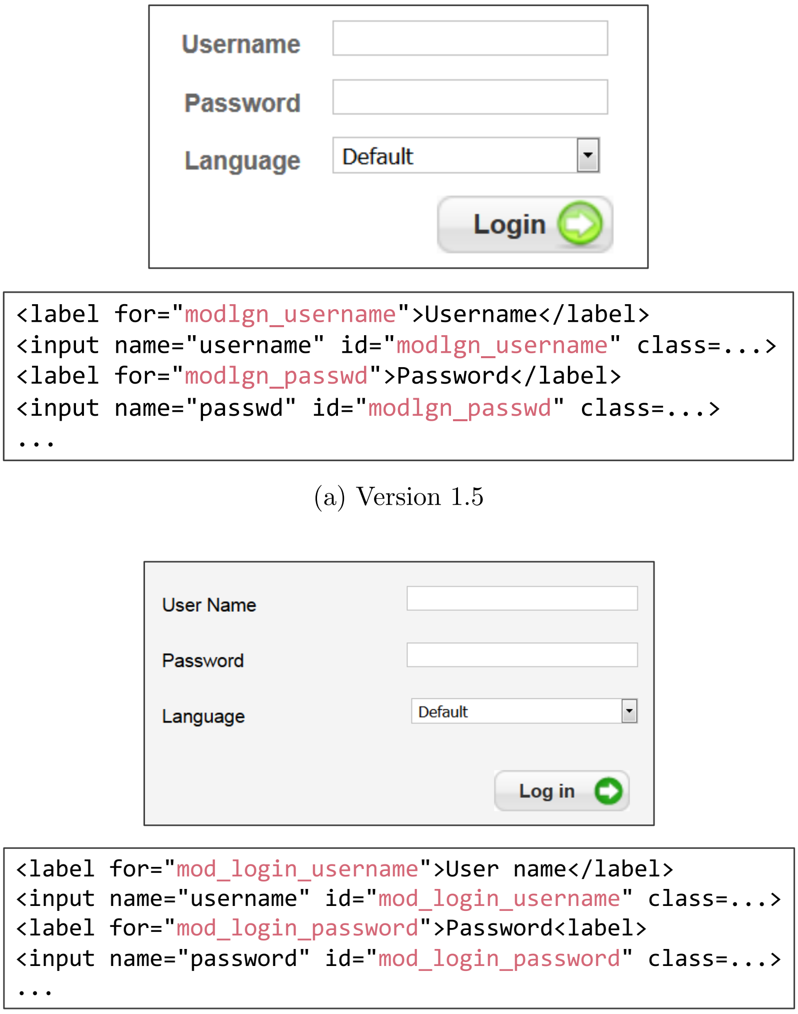
[1] `https://www.joomla.org/`

Table 2.1: Test script for Joomla! version 1.5

Action	Locator	Value
open	`/joomla`	
click	`css=li.item17>a>span`	
type	`id=modlgn_username`	user01
type	`id=modlgn_passwd`	pass01
click	`link=Login`	
assertTitle		Main

Table 2.2: Test script for Joomla! version 2.5

Action	Locator	Value
open	`/joomla`	
click	`link=Site Administrator`	
type	`id=mod_login_username`	user01
type	`id=mod_login_password`	pass01
click	`link=Log in`	
assertTitle		Main



```
<label for="modlgn_username">Username</label>
<input name="username" id="modlgn_username" class=...>
<label for="modlgn_passwd">Password</label>
<input name="passwd" id="modlgn_passwd" class=...>
...
```

(a) Version 1.5

```
<label for="mod_login_username">User name</label>
<input name="username" id="mod_login_username" class=...>
<label for="mod_login_password">Password<label>
<input name="password" id="mod_login_password" class=...>
...
```

(b) Version 2.5

Figure 2.1: Login form of Joomla!

user then enters their "Username" and "Password" on the login page and clicks the "Login" button, as shown in Figure 2.1(a). Table 2.1 shows the implementation of these operations as a Selenium IDE script. Each line in the test script represents one operation and consists of three components: *action*, *locator*, and *value*. The *action* specifies the type of operation, for example, "type" means input from the keyboard. The *locator* is an identifier that specifies the web element for the operation. Id, name, XPath, CSS selector, and so on can be used as a locator in Selenium IDE. For example, `css=li.item17>a>span` in the second line is a CSS selector and refers to the "Administrator" link. `id=modlgn_username` in the third line refers to the "Username" input form, whose id is `modlgn_username` in HTML. The *value* is the input value given by the user or the expected result, and "user01" in the third line is the input given to the "Username" input form. In addition, the postcondition of transition to the main page is given in the last line. “assertTitle” validates that the page title is the same as the value “Main”. If we execute the script in Table 2.1, the operations are executed sequentially from the first line. We can consider the test to have passed if the test execution completes.

2.2.2 Test script repair

We will introduce an example in which a test script requires modification. For the login form in Joomla! version 2.5, as shown in Figure 2.1(b), we cannot execute the test script developed for version 1.5, as shown in Table 2.1. This is because the page layouts and attributes of web elements differ between versions 1.5 and 2.5, and some web elements referred to by the locators in Table 2.1 do not exist in version 2.5. For example, the test script in Table 2.1 can be modified as shown in Table 2.2 for use in version 2.5.

When developers modify a test script manually, they need to first understand its behavior by referring to the locator, comment, and test case specifications. However, test scripts are often unreadable, and documents of test case specifications are often lacking. In such cases, understanding the behavior of test scripts is difficult. For example, the locator in the second line of Table 2.1 is described using a CSS selector, and we cannot understand what operation will be performed just by reading the test script. Test script modification should be automated because such tasks are time-consuming and can lower motivation for test automation.

2.2.3 Causes of test breakages

Existing test scripts are often modified in accordance with software updates, as described in Section 2.2.2. Hammoudi et al. [8] examined the causes of test breakages across 453 versions of eight web applications and roughly classified them into five types: locators, values/actions, page reloading, user session, and JavaScript popup boxes. Locator breakage occurs when the locator does not refer to the web element that the test is supposed to operate on. It is often caused by modifying the layout of pages or the attributes of web elements. To repair the broken test script, developers need to set the correct locator that refers to the web element to be operated on in the test. Hammoudi et al. revealed that locator breakage accounts for 73.62% of all test breakages, so locator repair appears to be important to support. Therefore, we focus on locator breakage in this study.

2.3 Approach

In this study, we propose COLOR, which uses various properties as clues obtained from the screen to gain more robustness against software updates. Existing locator repair ap-

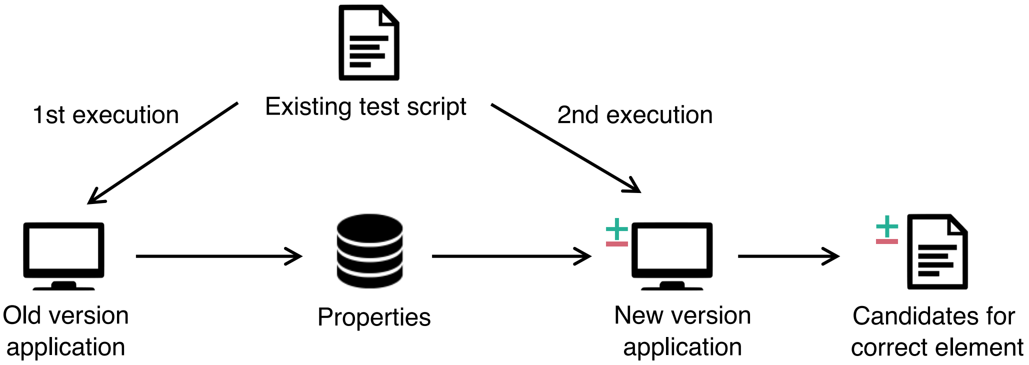


Figure 2.2: Overview of COLOR

proaches mainly focus on structure-based locators. COLOR is more robust than structure-based approaches because it uses more various clues. Therefore, COLOR can handle complex changes, including attribute changes, layout changes, and link text changes.

COLOR uses various properties (i.e., attributes, texts, images, and positions) as multifaceted clues to determine whether two web elements are the same or not. Figure 2.2 shows an overview of COLOR. COLOR requires two successive versions of an application. Now, we postulate that there are executable test scripts for the old version of the software V_k and that the test scripts are not executable for the new version of the software V_{k+1} due to broken locators. First, COLOR executes existing test scripts for V_k and collects 19 kinds of properties from each web element in V_k. These properties are used for recommending locators in the later procedure. Table 2.3 shows the properties as the clues used in this study. The properties are classified into four categories: attribute, position, text, and image. Target web elements are `<input>`, `<button>`, `<a>`, `<img>`, and `<select>` tags in this research, and we adopt their major attributes as properties in the attribute category. Second, COLOR executes existing test scripts on the new version of the software V_{k+1}. The recommendation procedure is triggered by a locator error, and COLOR presents candidates for a correct web element.

2.3.1 Recommendation algorithm

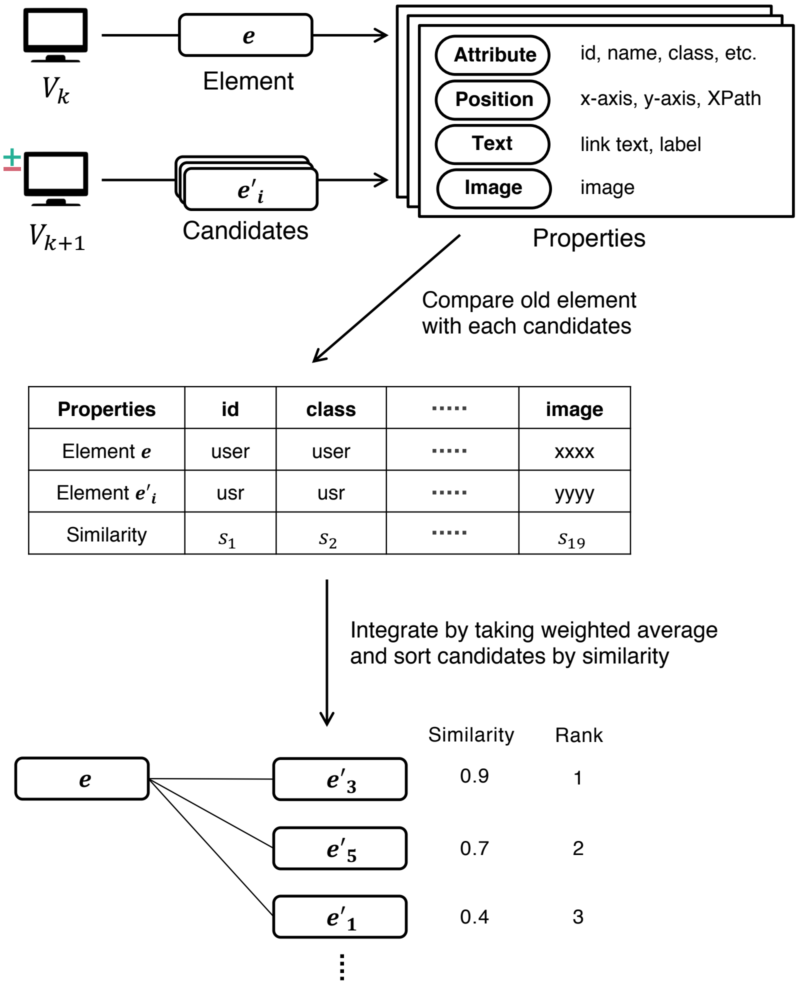
The locator l_e needs to correctly identify the web element to be operated on in the test for V_{k+1}. We define l_e as a locator that refers to the web element e being operated on. If l_e does not uniquely identify a web element in the test execution, the operation cannot be performed. Let E be the set of web elements on the page of V_{k+1}. If there is a web element $e' \in E$ with the same role as e in V_k, it is considered to be the web element that should be operated on in V_{k+1}. We can repair the locator by specifying e' and modifying l_e to $l_{e'}$. The locator is considered repaired if the same operation can be performed on the web elements in the test execution on both V_k and V_{k+1}. In this study, COLOR calculates

Table 2.3: Properties used in this study

Category	Property	Description
Attribute	id	unique id of web elements
	class	class of web elements
	name	name of web elements
	value	initial value of input forms
	type	type of input forms
	tag name	name of HTML tags
	alt	alternative text for `<img>` tags
	src	source URI of images
	href	destination of link anchors
	size	size of input strings
	onclick	event handler of click events
	height	height of web elements
	width	width of web elements
Position	XPath	absolute XPath of web elements
	X-axis	X-axix on the screen
	Y-axis	Y-axis on the screen
Text	link text	link text of `<a>` tags
	label	label linked with input forms
Image	image	image of web elements encoded to Base64

the similarity between e and each web element in V_{k+1} using various clues and finds a web element $e' \in E$ with the same role as e. The higher the similarity, the more likely it is to be a correct locator.

Figure 2.3 shows the outline of the recommendation procedure, and Algorithm 1 shows the details of the recommendation algorithm. First, COLOR calculates the similarities for each property and then combines them into a single similarity. Let P be the set of properties that e and e' have in common, and let $e[p]$ be the value of property $p \in P$ in e. Properties can have various types of values, such as numeric values, character strings, and specific values. Therefore, the method for calculating the similarity for each property needs to be adjusted accordingly. We use the Euclidean distance for numerical values and the Levenshtein distance for character strings as measures of similarity. For specific values, the similarity is set to 1 if the values match and to 0 if they do not. The similarity $s(e[p], e'[p])$ ($0 \leq s(e[p], e'[p]) \leq 1$) between $e[p]$ and $e'[p]$ is calculated as follows:



Properties	id	class	·····	image
Element e	user	user	·····	xxxx
Element e'_i	usr	usr	·····	yyyy
Similarity	s_1	s_2	·····	s_{19}

Figure 2.3: Outline of recommendation algorithm

When p is height, width, X-axis, or Y-axis:

$$s(e[p], e'[p]) = 1 - \frac{|e[p] - e'[p]|}{\text{Max}(p)} \tag{2.1}$$

When p is image, tag name, or size:

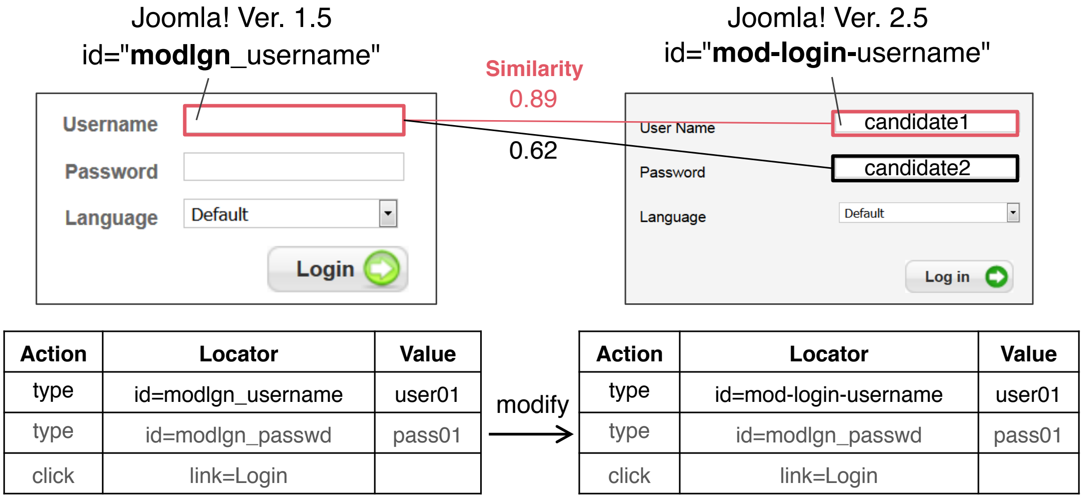
$$s(e[p], e'[p]) = \begin{cases} 1 & (e[p] = e'[p]) \\ 0 & (e[p] \neq e'[p]) \end{cases} \tag{2.2}$$

Otherwise (p takes a character string):

$$s(e[p], e'[p]) = 1 - \frac{\text{Levenshtein}(e[p], e'[p])}{\text{MaxLength}(e[p], e'[p])} \tag{2.3}$$

Algorithm 1: Recommendation algorithm

```
input  : An web element $e$ in $V_k$
output: Candidates $E$ in $V_{k+1}$ sorted by the similarities
for $p \leftarrow P$ do
    $e[p] \leftarrow$ the value of $p$ in $e$
for $e' \leftarrow E$ do
    for $p \leftarrow P$ do
        $e'[p] \leftarrow$ the value of $p$ in $e'$
        $s(e[p], e'[p]) \leftarrow$ calculating similarity between $e[p]$ and $e'[p]$
    $S(e, e') \leftarrow$ taking weighted average from each similarities
sort $E$ by $S(e, e')$
```



Action	Locator	Value
type	id=modlgn_username	user01
type	id=modlgn_passwd	pass01
click	link=Login	

Action	Locator	Value
type	id=mod-login-username	user01
type	id=modlgn_passwd	pass01
click	link=Login	

Figure 2.4: Application example

where Max(p) is the maximum value that $|e_i - e'_i|$ can take (e.g., Max(X-axis) is the width of the screen), Levenshtein($e[p], e'[p]$) is the Levenshtein distance between $e[p]$ and $e'[p]$, and MaxLength($e[p], e'[p]$) is the length of the longer of $e[p]$ and $e'[p]$.

$s(e[p], e'[p])$ equals 1 when $e[p]$ and $e'[p]$ are the same and approaches 0 as the difference between $e[p]$ and $e'[p]$ becomes larger. Next, COLOR integrates the calculated similarity $s(e[p], e'[p])$ for each property, taking into account that different properties may have different contributions to the overall similarity. For example, the contribution of the class property may be considered smaller than that of the id property. This is because different web elements may have the same class but never have the same id. If the ids of two web elements are the same before and after the software update, it is likely that the web elements are the same, but if only their classes are the same, it is not necessarily

Table 2.4: Similarity calculation in Figure 2.4

Property	id	name	type	Y-axis	label	Integrated
"Username" input form	modlgn_ username	username	text	186	Username	-
Candidate1	mod-login-username	username	text	216	User Name	-
Similarity	0.78	1.00	1.00	0.94	0.78	**0.89**
Candidate2	mod-login-password	passwd	password	257	Password	-
Similarity	0.33	0.00	0.00	0.86	0.00	**0.62**

the case. Therefore, after calculating $s(e[p], e'[p])$ for each property, COLOR calculates the integrated similarity $S(e, e')$ $(0 \leq S(e, e') \leq 1)$ by taking a weighted average using the weight w_p for each property. In this study, COLOR determines w_p based on the contribution of each property. We will explain how to set the weight w_p in Section 2.4.2. The similarity $S(e, e')$ is calculated as follows:

$$S(e, e') = \frac{\sum_{p \in P} s(e[p], e'[p]) w_p}{\sum_{p \in P} w_p} \tag{2.4}$$

2.3.2 Application example

Figure 2.4 shows how COLOR can be applied to the example in Section 2.2. In this case, the id of the "Username" input form has been modified between the two releases, so the test script for version 1.5 of the software cannot be executed on version 2.5 of the software. As an example, we will show the procedure for recommending the locator for the "Username" input form in the first line of the test script.

First, we calculate the similarity between the "Username" input form in version 1.5 and each web element in version 2.5. We will focus on the two candidates shown in Figure 2.4 in version 2.5. Part of the calculated similarities is shown in Table 2.4. We have included five properties in Table 2.4: id, name, type, Y-axis, and label. The integrated similarity is actually calculated using a total of 19 properties, as shown in Table 2.3.

From the calculation results, we can see that the similarity between *candidate1* and the "Username" input form is the highest. Therefore, we can infer that *candidate1* is the web element most likely to be the "Username" input form in version 2.5. Based on this, we can conclude that the broken locator on the first line of the test script should be modified to `id=mod-login-username` in version 2.5.

Table 2.5: Applications for our experiment

Name	Description	1st Release	2nd Release	# all locators	# broken locators
Joomla!	Content management system	1.5.0	2.5.0	143	94
PHP-Fusion	Content management system	6.0.1	7.0.0	226	47
MantisBT	Bug tracking system	1.1.8	1.2.0	250	40
MRBS	System for booking meeting rooms	1.10.7	1.11.5	80	22

2.4 Evaluation

To evaluate the effectiveness of COLOR, we implemented the recommendation algorithm and formulated the following research questions:

RQ1: *What are the reliable properties for recommending locators?*

COLOR uses various properties as clues obtained from screens, but the properties have different contributions to the recommendation accuracy. Therefore, we will investigate which properties have the greatest impact and propose a method for determining the weight of each property.

RQ2: *Is COLOR more robust against page layout changes than structure-based approaches?*

Structure-based approaches are known to be fragile in the face of page layout changes. We will confirm that structure-based approaches do not always perform well and that using clues other than structural ones can help recommend correct locators. We will use the similarity calculation algorithm from WATER [23] as a structure-based approach for comparison. We have chosen WATER for this comparison because it is similar to COLOR in terms of calculating the similarities between two web elements. Although WATERFALL is a state-of-the-art technique, the core idea of WATERFALL does not conflict with COLOR. This is because WATERFALL applies WATER iteratively across a sequence of fine-grained versions of a web application, and we can apply WATERFALL even if we substitute WATER with COLOR.

RQ3: *Can COLOR accurately recommend correct locators?*

If COLOR recommends many incorrect web elements, it will be difficult to use in practice. We will evaluate COLOR's ability to accurately recommend correct locators. The idea of applying locator repair iteratively across a sequence of fine-grained versions can be incorporated into other techniques, including COLOR.

2.4.1 Experimental setup

Table 2.5 shows the applications used in our experiment. These are open-source web applications implemented in PHP. Table 2.5 includes the names of the open source software (OSS), descriptions, release versions, numbers of all locators surveyed, and numbers of broken locators. These applications have all been used in previous studies [22, 23]. We used two release versions for each application in our experiment. We selected applications and versions that include complex changes such as attribute changes, layout changes, link text changes, and so on.

We will assume that there are test scripts developed for the first release version that cannot be executed on the second release version due to broken locators. The procedure for the experiment is as follows:

(1) We collected the properties shown in Table 2.3 from a total of 699 operable web

elements (i.e., buttons, input forms, and links) on the pages of the main features in each release version.
(2) We manually examined the pairs of web elements that have the same role across two release versions. We define these pairs as the correct set C, and let $\langle e_i, e'_i \rangle \in C$ be each pair of web elements. To reduce bias in the results, we eliminated multiple similar or identical web elements (e.g., buttons in a tabular form, web elements in the header or footer, etc.) from C except for one.
(3) We specified broken locators between two release versions. We used four types of locators (id, link text, name, and absolute XPath) in our experiment because these are commonly used by developers. Although relative XPath and CSS selector are also common, we did not use them because they have multiple expressions and the robustness of such locators depends on their expressions. We prioritized the types of locators in the order of (I) id, (II) link text, (III) name, and (IV) absolute XPath, which is the default order in selenium IDE. A type of locator with a higher priority is used preferentially. For example, if e_i has an id, it is used as a locator for e_i, and if e_i does not have an id but has link text, the link text is used as a locator. If the locator changes between two release versions, we consider it to be broken.
(4) We applied COLOR to broken locators. COLOR shows candidates for the correct web element in order of similarity, and candidates are typically checked to start from the top rank. Therefore, presenting the correct locator at a higher rank is beneficial.

To evaluate the accuracy of the recommendation algorithms, we use mean reciprocal rank (MRR). MRR is a statistical measure commonly used to evaluate the ranking of correct answers in recommendation and search systems. A higher MRR means that the correct answer is more likely to appear at a higher rank. MRR is calculated as the average of the reciprocal ranks of the results for a sample of queries Q:

$$\mathrm{MRR} = \frac{1}{|\mathrm{Q}|} \sum_{\mathrm{i}=1}^{|\mathrm{Q}|} \frac{1}{\mathrm{rank_i}} \tag{2.5}$$

where $\mathrm{rank_i}$ is the rank of the correct answer for the i-th query. In our experiment, the query is e_i and the correct answer is e'_i. If a locator recommendation system raises e'_i to a higher rank, the MRR will be higher, indicating that the system is superior.

2.4.2 Weights of properties

The weights of properties are determined based on their contribution to the accuracy of recommendation. We set weights of properties in three ways: *unweighted*, *heuristic weights*, and *automatically calculated weights*. *Unweighted* means that we consider all properties as making the same contribution to the recommendation, namely, $w_p = 1$.

Table 2.6: Heuristic weights

Property	Joomla!	PHPFusion	MantisBT	MRBS
id	2	–	–	–
class	1	1	–	–
name	4	1	1	1
value	0	1	1	1
type	–	1	1	1
tag name	1	0	0	–
alt	0	1	0	0
src	–	–	–	–
href	0	–	0	3
size	0	–	–	–
onclick	2	2	–	–
height	15	–	–	–
width	1	1	–	–
xpath	1	0	–	1
X–axis	0	1	2	0
Y–axis	1	1	0	0
link text	1	1	0	1
label	0	–	–	–
image	0	–	–	–

The heuristic weights are the weights resulting from the search to maximize the MRR for each OSS in the experiment. The search was conducted by starting from the top of Table 2.3, introducing properties one by one, and increasing the weights by 1 until the MRR reached its maximum.

In Table 2.6, "–" means that the property is not included in the target web elements in the application or its impact is very small and therefore does not affect the result. A weight of 0, on the other hand, can adversely affect the results if the property actually has a positive weight. Table 2.6 shows that the way to set the weight depends on the target, but some tendencies can be observed. To answer RQ1, class, name, type, onclick, and width would generally be effective because they contribute to improving MMR for multiple targets.

The heuristic weights are determined by calculating the MRR for each application, but this method is not practical when actually applying our technique because the correct web elements are not known in advance. Therefore, it is desirable to have weights that can be commonly used for multiple applications.

Table 2.7: Stability, uniqueness, and automatically calculated weights

Property	Stability	Uniqueness	Weight
id	0.86	1.00	0.86
class	0.90	0.40	0.35
name	0.87	0.91	0.80
value	0.70	0.84	0.61
type	0.93	0.47	0.44
tag name	0.96	0.21	0.21
alt	0.57	1.00	0.57
src	0.62	1.00	0.62
href	0.81	0.86	0.62
size	0.83	0.32	0.26
onclick	0.75	0.88	0.66
height	1.00	0.33	0.33
width	0.99	0.66	0.66
XPath	0.72	1.00	0.72
X-axis	0.95	0.64	0.60
Y-axis	0.91	0.71	0.65
link text	0.83	0.91	0.76
label	0.44	0.74	0.31
image	0.12	0.88	0.12

Therefore, we propose an automatic weighting method based on the assumption that properties with more stable and unique values contribute more to identifying similar web elements. If a property is stable and has a unique value, the roles of two web elements with the same value of the property are likely to be the same across two release versions. We define stability(p) as the stability and uniqueness(p) as the uniqueness of a property. Therefore, properties with high stability are less likely to change between releases, and properties with high uniqueness are less likely to have the same value on a page. stability(p), uniqueness(p), and w_p are calculated as follows[2]:

$$\text{stability}(p) = \frac{\sum_{\langle e_i, e_i' \rangle \in C} s(e_i[p], e_i'[p])}{|C|} \tag{2.6}$$

$$\text{uniqueness}(p) = \frac{\text{the number of unique web elements in terms of } p}{|C|} \tag{2.7}$$

$$w_p = \text{stability}(p) \times \text{uniqueness}(p) \tag{2.8}$$

[2] The number of unique web elements in terms of p is summed up from each page.

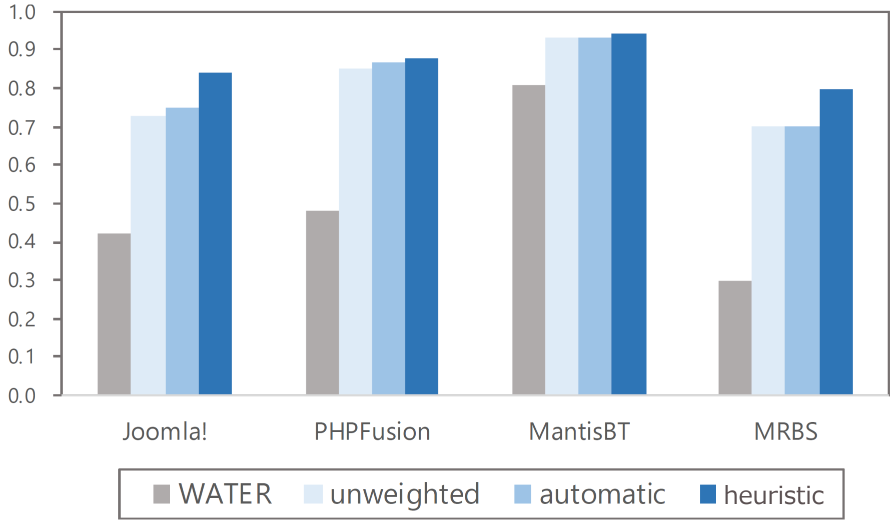


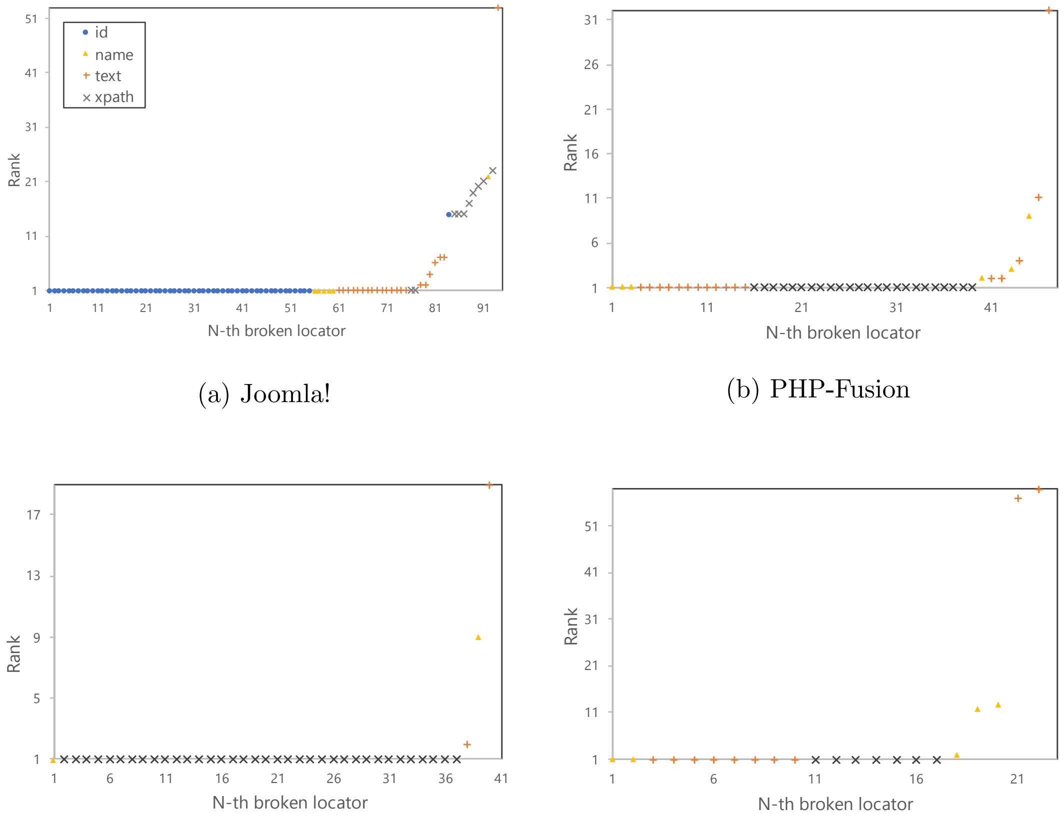
Figure 2.5: MRR calculated by each algorithm

Table 2.7 shows the values of stability(p), uniqueness(p), and w_p for each property. In Table 2.7, the values of weight are not simply the product of stability and uniqueness but are the mean values of w_p calculated by (2.8) for each application. The results are fairly intuitive. For example, id values are completely unique because different web elements cannot have the same id values, whereas class values are not as unique as id because different web elements can have the same class. Therefore, the weight of class values is smaller than that of id.

The weights of id, name, and link text are higher than those of other properties because they have both high stability and uniqueness. These properties are commonly used as locators in test scripts. Value, src, href, and onclick are not typically used as locators, but they also have a high weight. X-axis and Y-axis are not unique in the category of position, but they have higher stability than XPath and are therefore fairly reliable. The image property makes the lowest contribution of all properties. This is because its stability is very low due to page layout changes in the applications.

2.4.3 Results

Figure 2.5 shows the MRR for each application. We calculated the MRR in WATER and three patterns of COLOR: unweighted, heuristic weights, and automatically calculated weights. The results show that COLOR yields higher MRR than WATER, indicating that COLOR recommends correct locators more accurately when there are significant page layout changes. COLOR with automatically calculated weight yields slightly better MRR than COLOR with no weights in the cases of Joomla! and PHP-Fusion but does not improve the MRR in the cases of MantisBT and MRBS. COLOR with heuristic weights



(a) Joomla!

(b) PHP-Fusion

(c) MantisBT

(d) MRBS

Figure 2.6: Rank for each broken locator

improves MRR by 1% – 14% compared with COLOR with automatically calculated weight and by 16% – 167% compared with WATER. To answer RQ2, these results show that COLOR is more robust against page layout changes than structure-based approaches.

Figure 2.6 shows a list of broken locators sorted by rank yielded by COLOR using manually weighted properties. Figure 2.6 contains four types of markers (id, name, text, and XPath) that indicate the types of broken locators. COLOR tends to not present the correct locator at a high rank when the XPath locator is broken in Joomla! or when the text locator is broken in other applications.

As a result, COLOR presents the correct web elements for a broken locator in first place with 77% – 93% accuracy and above third place with 82% – 95% accuracy. To answer RQ3, we consider this accuracy to be sufficient for practical use.

2.5 Discussion

2.5.1 How should we determine the weights?

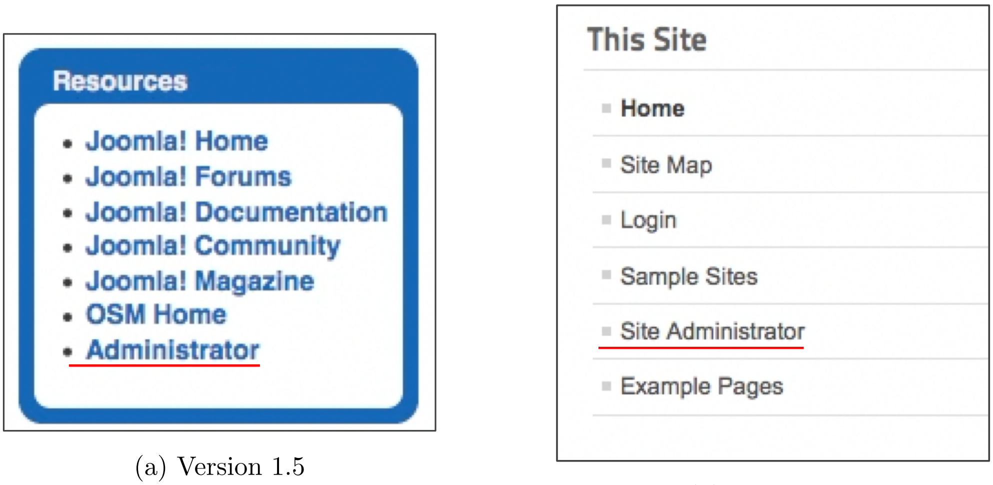
Heuristic weights rely on a history of locator modifications in the target application, making them more difficult to apply compared to automatic weights. If there is no significant difference in accuracy between heuristic and automatic weights, it is preferable to use automatic weights. However, Figure 2.5 shows that automatic weighting was not very effective in this study. Therefore, it is recommended to use heuristic weights when accuracy is a priority and automatic weights when there is no history of locator modifications in the early stages of development.

The problem with automatically calculated weights is that they use all properties, including harmful properties. We believe that the key idea behind automatic weighting is not bad, as properties that have low weights in Table 2.7 (e.g., tag name, size, image) tend to harm the accuracy of recommendation. However, the calculation method (2.8) can be improved. This result indicates that generalization on an application-by-application basis is inadequate, and determining weights on a per-page basis may be necessary.

As can be seen from Table 2.6, the optimal weight varies depending on the target application. This is because the characteristics of the changes made in the target applications are different and difficult to generalize. If the appearance and layout of the application are drastically changed, the image and axis properties may adversely affect accuracy. On the other hand, if they are not changed but the attributes of web elements are changed, the image and axis properties may positively affect accuracy. Therefore, we need to understand the characteristics of changes from past versions and tune the weights accordingly. For example, removing harmful properties from automatic weighting depending on the characteristics of an application may improve the accuracy of recommendation.

2.5.2 Characteristics of COLOR

Figure 2.5 indicates that COLOR is more robust against screen layout changes than WATER. The reason for this improvement is that WATER mainly uses XPath to calculate similarity among web elements, which is not effective when dealing with significant page layout changes. For example, the site design of Joomla! was renewed with the version upgrade from 1 to 2. Figure 2.7 shows the change in the top page menu. The link text to the login page is changed from “Administrator” to “Site Administrator”, and the structure of the top page is changed drastically. In this case, the similarity of XPath is not effective in recognizing that the two link texts are the same because the DOM trees of the screens are likely to have changed. In contrast, information related to the semantics of web elements (i.e. id, name, text, etc.) is less variable than the structure of a screen. In this example,



(a) Version 1.5

(b) Version 2.5

Figure 2.7: Top page menu of Joomla!

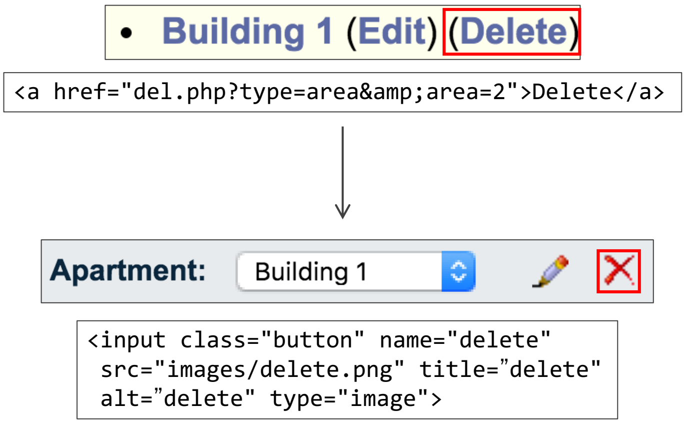
the strings “Administrator” and “Site Administrator” are similar, so COLOR seems to be more effective than WATER.

In Figure 2.6, some web elements are presented at low ranks despite using optimal weights. This is because the weighting cannot improve the MRR due to the characteristics and the theoretical limit of COLOR. We identified two cases where COLOR did not work well:

1. Many similar web elements on a web page.
2. Changes to a different HTML tag.

The first case occurs when there are many similar web elements on a page (e.g. web elements in a table, radio buttons with many options, etc.). In such cases, there are many web elements that have similar values for their properties, making it difficult for COLOR to differentiate between them. Such web elements often do not have a descriptive name or id, so XPath is often chosen as a locator. This is why COLOR tends not to present correct web elements for XPath locators of Joomla! in Figure 2.6.

The latter case occurs when one HTML tag is changed to another. For example, in MRBS, the “delete” link was changed to a “delete” button between two release versions, as shown in Figure 2.8. COLOR calculates similarities by using properties that two web elements have in common. However, the “delete” link and the “delete” button have no attributes in common, which means that COLOR may present incorrect web elements at higher ranks because there are few factors to distinguish between the two without structural clues, and it cannot handle page layout changes.



```
<a href="del.php?type=area&amp;area=2">Delete</a>
```

```
<input class="button" name="delete"
 src="images/delete.png" title="delete"
 alt="delete" type="image">
```

Figure 2.8: Example of changes to different HTML tag

To solve these problems, we need to analyze the semantic equivalence between two web elements. One possible solution is to collect clues from web elements outside of the focus. For instance, we could obtain a character string that explains a web element. COLOR currently refers to `<label>` elements that have a `for` attribute linked with `<input>` elements. However, `<label>` elements often do not have a `for` attribute in actual applications. We believe that most of these problems can be solved if COLOR can obtain a character string that explains a web element by using structural clues (e.g., labels are often on the left side of the input form).

Another solution is to extract semantics from the HTML expression of the web element. Although the “delete” link and the “delete” button in the previous example both include the string “delete” in their HTML, COLOR cannot use it as a clue because of the different properties. We can solve this problem by using natural language processing techniques to extract semantics from web elements.

A limitation of COLOR comes from the selection of target web elements. In this research, we chose `<input>`, `<button>`, `<a>`, `<img>`, and `<select>` tags as operable web elements. However, some applications use the `<div>` tag as a button, which means that COLOR cannot recognize them as operable web elements. We excluded these from the target web elements in this study, so it would be too costly to regard all `<div>` tags as operable web elements. We think we can handle this problem by analyzing the event linked to the web element.

2.5.3 Execution time

The execution time of the COLOR algorithm depends on the number of target web elements on the page. If the page has many input forms, links, buttons, and so on, the execution time will become longer because the number of candidates increases. The application of COLOR takes approximately 40 ms per operable web element on a web page. Most pages only contain several dozen web elements, so we consider COLOR to have no practical problems in terms of execution time.

In some cases, it may be difficult to apply COLOR to pages with a large number of web elements, such as portal sites. In such cases, we can compare web elements that have the same tag in order to reduce the execution time.

2.6 Threats to Validity

This study has some threats to external validity. First, the number of applications in our experiment is small, which may bias the results. We chose applications that include page layout changes to show that COLOR can handle them. In the future, we want to confirm whether the results will change when we use different applications. The same is true for the release versions we chose, because the experimental results may be different if different types of changes are made between the first and second releases. We plan to obtain more general results by conducting experiments using three or more versions for each application. However, there are sometimes drastic changes even between successive versions of software, and we want to show that COLOR is effective in such cases.

We also need to discuss the selection of properties. We chose attributes that appear frequently in the target applications. Other applications may include attributes that COLOR does not use, and these attributes may be important for recommendations. In the four target applications, we were not able to find any examples to show that these properties are inadequate. We need to apply COLOR to more applications to confirm the adequacy of the target properties.

Furthermore, the weights of the properties were determined based on the four applications in our study. When we apply COLOR to other applications, the results may change. However, there were no applications that resulted in low MRR due to weighting in our experiment. Therefore, these weights seem to be versatile to some extent. In the future, we want to consider ways to calculate optimal weights for each application and increase the number of target applications to obtain more general results.

2.7 Conclusion

We propose a novel approach called COLOR to support repairing broken locators by using various clues obtained from a screen, in order to reduce the costs of regression testing. COLOR uses various properties (e.g., attributes, positions, texts, and images) as clues and is more robust than structure-based approaches against complex changes including page layout changes. We first identified which properties are reliable for recommending locators by examining 699 locators across two release versions of applications, and found that the class, name, type, onclick, and width properties are effective for multiple targets, and several properties are effective for some targets but harmful for others. We applied COLOR to broken locators and showed that (1) COLOR can present the correct locator in first place with 77% - 93% accuracy, and (2) COLOR is more effective against complex changes including page layout changes than WATER, a conventional structure-based approach.

We plan to improve COLOR to enable it to recommend locators that were not recommended in our experiment. Additionally, we want to increase the number of applications for experiments and confirm that COLOR is generally applicable. We also aim to assess the extent to which our proposed technique can actually reduce the amount of human effort required. The potential reduction in effort may vary depending on how the proposed technique is utilized (e.g., automatically fixing broken test scripts and having the user review them later, or running tests and interactively fixing each failed step as it occurs, etc.).

Chapter 3

Maintainable Test Script Generation via Manual Testing

3.1 Introduction

Software testing is a critical process for evaluating and improving the quality of software, and developers often spend significant time and effort on it [45, 46]. Many researchers are working to make software testing more efficient and effective at detecting bugs [47]. Automated testing can greatly improve the efficiency of software testing, especially for tests that are repeated frequently, such as regression testing. With the increasing need to shorten software release cycles in order to respond quickly to market changes, test automation has become essential for software development.

When testing software with graphical user interfaces (GUIs), such as web applications, it is also important to test the GUI from the user's perspective. This type of testing is called end-to-end testing, which is the focus of this study. To automate end-to-end testing, tools that automate web-browser operation, such as Selenium [2], are often used. Automating end-to-end testing requires implementing test scripts, and these scripts may need to be modified as the application under test is modified. Christophe et al. [48] studied the change history of the source code including Selenium test scripts, for eight open-source web applications, and how modifications to the applications affected these scripts. They found that 75% of Selenium test scripts were changed at least once every nine commits (once every 2.05 days). This shows that Selenium test scripts are updated frequently as the application evolves, so the maintainability of these scripts is important.

Record & replay and programming are two main approaches to automating end-to-end testing. Leotta et al. [3] conducted a comparative study of these two approaches and found that the programming approach took 32-112% longer to implement test scripts, but required 16-51% less time to modify them. Overall, the programming approach was found to be less costly in most cases when more than three modifications were needed.

In their experiment, the participants who used the programming approach implemented their test scripts using the page-object pattern. The page-object pattern is a design pattern for end-to-end test automation that improves the maintainability of test scripts by separating test cases from page-specific code [49, 50]. The results of the study suggest that the programming approach with the page-object pattern is well-suited for software that is released frequently in short periods of time. However, using the programming approach to create highly maintainable test scripts requires skilled developers. This difficulty in implementing test scripts can make it challenging to introduce automated end-to-end testing into software development. To address this issue, Stocco et al. [51] proposed an approach for automatically generating page objects. This approach involves crawling an application to generate page objects, but it can be difficult to apply to large-scale applications and generate complete page objects.

We propose an approach for automatically generating test scripts using the page-object pattern from manual testing logs. The key advantage of our approach is that it allows test scripts to be generated simply by performing manual tests, without the need for users to be aware of test automation. While automated testing is becoming more common in industry, not all testing can be automated, due to the high implementation cost and the difficulty of automating certain types of tests, such as user-interface testing and usability testing.

Manual testing approaches can be broadly classified as scripted testing or exploratory testing [52–54]. Scripted testing involves designing tests in advance and then executing them according to the test design. Developers often document the test design and may create test-procedure manuals with detailed instructions for each test. In contrast, exploratory testing involves conducting test execution, test design, and learning simultaneously, without pre-planning the tests. Exploratory testing cannot currently be replaced by automated testing because it relies on the expertise of the tester.

Our proposed approach allows test scripts to be generated with minimal preparation by recording manual testing, which is an essential part of software development. Our approach addresses the problems of manually implementing test scripts and the limitations of other approaches for generating page objects. In our approach, the actions of the tester are treated as the use of a feature on a web page, and these actions are converted into methods of the page object. This makes it possible to automatically generate test scripts from manual testing logs.

The test scripts generated by our approach not only include page objects, but also test cases that cover the features of an application by analyzing the page transitions of that application. In general, a test case is a specification of a test and includes a set of operations that are executed on an application to determine if it meets the software requirements. Although test code may be written in natural language, for the purposes of this study, test code is defined as implemented in source code. Our approach lowers the barrier to introducing automated end-to-end testing to software development.

To evaluate the effectiveness of our approach, we asked four testers to conduct manual testing on an open-source web application and evaluated whether our approach could generate useful test scripts. A test script in this context refers to a set of page objects and test cases. The results of the experiment showed that our approach was able to generate a greater number of complete methods in page objects than a current approach for page-object generation. We also conducted an empirical evaluation of whether our approach could reduce the cost of implementing test scripts for real systems. The results showed that our approach reduced the time required to implement test scripts by about 48% compared to manual implementation.

The contributions of this study are as follows:

- We propose a technique for generating automated test scripts with page objects from manual testing activities, which is an essential part of software testing.

- We evaluated the completeness of the generated test scripts and showed that our approach can generate a greater number of complete page objects than an existing page-object generation technique.
- Our empirical evaluation showed that our approach can significantly reduce the costs of test script implementation compared to other practical approaches used in real-world software development.

3.2 Page Object

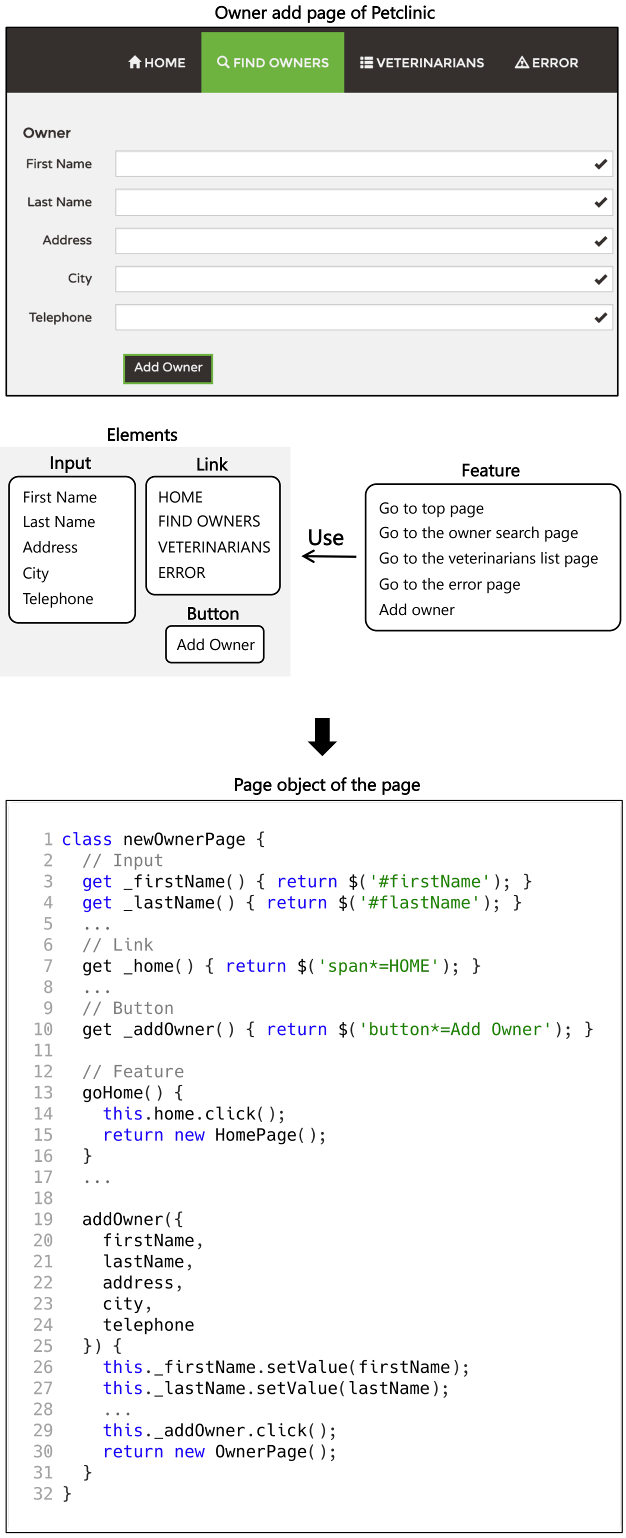
A page object is an object-oriented representation of a web page, with each web page represented as an object. In this study, we define a page object as a class that contains accessors and methods. An accessor is used to obtain a reference to a web element using a locator. The body of a method is a sequence of operations, such as clicking on web elements, entering values into input fields, or selecting items from drop-down lists. The method returns the page object of the destination page. Web elements that are operated on in the methods are specified using an accessor.

Figure 3.1 shows an example of the owner add page in the PetClinic open-source web application [55] and its corresponding page object. The owner add page consists of five input fields, four links, and one button. It also has a feature that allows it to transition to another page or add an owner. The page object in Figure 3.1 is implemented in JavaScript using WebdriverIO [56], a test-automation framework for web or mobile applications that makes the test code more concise than plain Selenium WebDriver.

The `_firstName` accessor indicates an input field for the first name. `$('#firstName')` captures the web element with the id of "firstName" in the HTML of the web page. The information used to uniquely identify a web element on a web page, such as `#firstName`, is called a locator. Users can use id, name, text, XPath, etc. as a locator. The defined accessors are only called from methods within the page objects.

The `addOwner()` method in Figure 3.1 takes the values to be entered in each input field as arguments. This method inputs the values in each input field and then clicks the add owner button. When writing test cases to carry out an operation on the owner addition page, we use the methods defined in the page object. The return value of the method is generally the page object of the destination page after the page transition. This allows us to write test cases using a method chain.

The page-object pattern allows test cases and page-specific code to be separated by modularizing operations and locators. This means that changes to test cases can be minimized by only modifying accessors or methods in the page object when web pages or features under test are modified. This makes it easier to maintain test cases as the application evolves.



```
class newOwnerPage {
  // Input
  get _firstName() { return $('#firstName'); }
  get _lastName() { return $('#flastName'); }
  ...
  // Link
  get _home() { return $('span*=HOME'); }
  ...
  // Button
  get _addOwner() { return $('button*=Add Owner'); }

  // Feature
  goHome() {
    this.home.click();
    return new HomePage();
  }
  ...

  addOwner({
    firstName,
    lastName,
    address,
    city,
    telephone
  }) {
    this._firstName.setValue(firstName);
    this._lastName.setValue(lastName);
    ...
    this._addOwner.click();
    return new OwnerPage();
  }
}
```

Figure 3.1: Owner add page of PetClinic and its page object

3.3 Related Work

Stocco et al. [51] proposed a technique called APOGEN for automatically generating page objects. APOGEN generates page objects by crawling web applications under test and automatically extracting web elements from the pages. It then clusters the web pages based on their similarity and integrates the pages belonging to the same cluster into a single page object. This is because having multiple similar page objects can reduce the modularity of the test script, which can undermine the strength of the page object.

Although APOGEN can help reduce the cost of implementing page objects, it has two problems with regard to page-object generation. The first problem is that it requires some preparation to use. When an application requires specific input values for page transitions, it is necessary to teach the crawler the locators of input fields and the input values in advance. In addition, if APOGEN does not propose the correct cluster, users need to fix the cluster manually. In contrast, our approach requires almost no preparation because it uses logs of manual testing, which are essential for software development. Chen et al. [57] improved the accuracy of page clustering for page-object generation by considering CSS styles and the attributes of web elements, but the problems caused by crawling have not been resolved.

The second problem with APOGEN is the completeness of the page objects it generates. If the crawling cannot cover certain web pages, APOGEN will not be able to generate page objects for those pages. APOGEN generates methods that operate web elements enclosed in `<form>` tags as a feature of the web page, but this technique may not be applicable on some web pages. This is because using features of web pages is not always the same as operating web elements enclosed in `<form>` tags. In addition, because APOGEN converts all possible page transitions into methods, it may generate too many methods that are not used in actual tests if the application has many links or buttons. In contrast, our approach can accurately extract features from pages regardless of their structure by using tester operations, so it is likely to generate useful methods for automated testing. Furthermore, APOGEN only generates page objects, but our approach can also generate test cases that use these page objects.

3.4 Approach

Figure 3.2 shows an overview of our approach, which takes manual testing logs as input and outputs test cases and page objects. The generated test cases use the page objects and call methods declared in them. In order to record manual testing activity, we developed a tool [58] that collects test logs that consist of operation data, such as information about tags and attributes of operated web elements, types of operations (e.g. click or input),

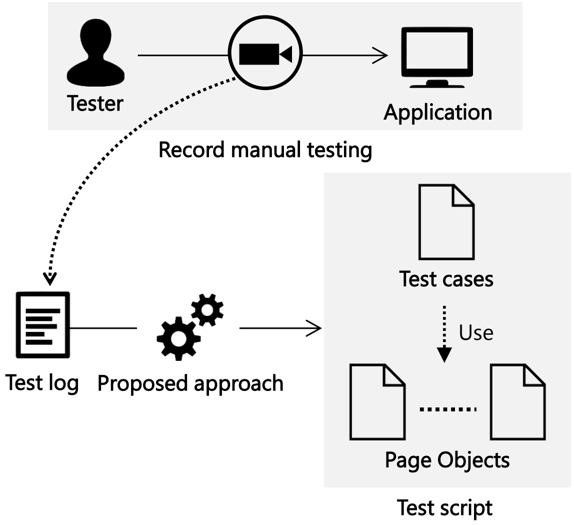


Figure 3.2: Overview of proposed approach

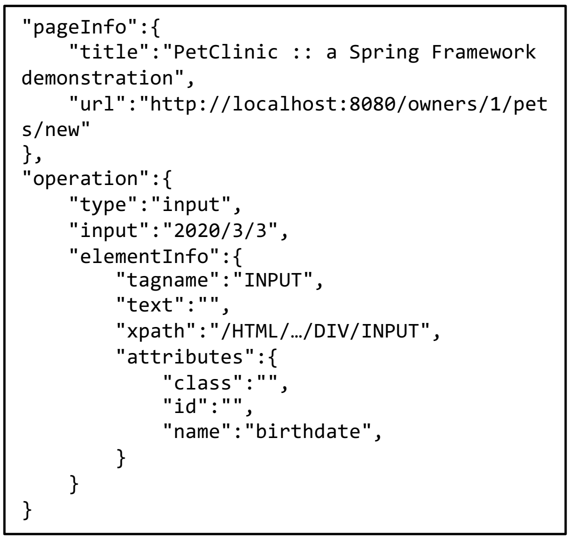
and page titles/URLs where the operations are performed. Testers can record this data without being aware of the tool's existence during the test. Figure 3.3 shows an example of an operation data when a pet's birthday is entered on the pet add page of PetClinic.

Our approach consists of two phases: page-object generation and test-case generation. In the page-object-generation phase, our approach generates page objects using data of operated web elements and operation procedures. In the test-case-generation phase, our approach selects test cases that cover all page transitions by analyzing page transitions obtained from test logs and constructs test cases that leverage the generated page objects.

3.4.1 Page-object generation

The proposed approach generates page objects for all web pages visited during a test. However, some web applications (e.g., single-page applications) may not have obvious page transitions, so we clarify the definition of a page. In this study, testers have the option to choose either a title match or URL match as the definition of page equality. They can also use regular expressions to define pages with matching titles or URLs as the same.

The generated page objects contain accessors that access web elements that have been operated at least once during the test. These accessors return web elements specified by locators using a function of WebdriverIO. To improve the robustness against application modifications, we use locators in the following order of priority: (i) id, (ii) name, (iii) text, and (iv) absolute XPath. This is because XPath locators are known to change more frequently than other locators. Text locators are only used for web elements that can contain text (e.g., `<a>` and `<button>`). They identify web elements by whether the link



```
"pageInfo":{
    "title":"PetClinic :: a Spring Framework
demonstration",
    "url":"http://localhost:8080/owners/1/pet
s/new"
},
"operation":{
    "type":"input",
    "input":"2020/3/3",
    "elementInfo":{
        "tagname":"INPUT",
        "text":"",
        "xpath":"/HTML/…/DIV/INPUT",
        "attributes":{
            "class":"",
            "id":"",
            "name":"birthdate",
        }
    }
}
```

Figure 3.3: Datum of operation in test log

text and inner text match the given string.

In the page-object pattern, a method contains a sequence of operations performed on a web page and represents a feature provided by the page. An operation o is defined as follows:

$$o = \langle t, i, e, p \rangle$$

where t is the type of operation (e.g. input or click), i is the input value, e is the operated web element, and p is the web page where the operation is performed. We consider a sequence of operations performed from the time a tester arrives on a certain page until the time they leave as the use of a certain feature of that page. We call such operations an *operation sequence*.

Algorithm 2 describes the detailed algorithm for generating methods for page objects. We first need to obtain the set of pages that testers visited by analyzing the test log (lines 1–4). Suppose we have pages p_1,..., p_n. We then extract the operation sequences performed on each page as the candidates for the methods (lines 5–16). Let S_{p_i} be the operation sequences for p_i. By scanning the test log, we can retrieve the operation sequences performed on p_i.

If we converted all operation sequences into methods, many duplicate methods would be generated. Therefore, we prevent the generation of duplicate methods by rejecting operation sequences that are included in other operation sequences (lines 17–27). This process aims to generate only versatile methods. For example, we can replace the absence of an operation on an input field with the operation of entering an empty string into the input field.

Let us define operation sequence s_1 and E_{s_1} as the set of web elements operated in

Algorithm 2: Method generation for page objects

```
Input: a test log
Output: methods for each page object
page set $P \leftarrow \emptyset$;
foreach operation in the test log do
    add the page where the operation is executed on $P$;
/* Now we have pages $p_1, p_2, ..., p_n$ */
foreach page $p_i$ in $P$ do
    let $S_{p_i}$ is the operation sequences for $p_i$;
    $S_{p_i} \leftarrow \emptyset$;
operation sequence $s \leftarrow \emptyset$;
foreach operation in the test log do
    add the operation to $s$;
    if there is a page transition, and the previous operation is executed on $p_i$ then
        add $s$ to $S_{p_i}$ for the page;
        $s \leftarrow \emptyset$;
foreach page $p_i$ in $P$ do
    let the sequences adopted as methods for the page object of $p_i$ be $M_{p_i}$;
    $M_{p_i} \leftarrow \emptyset$;
    sort $S_{p_i}$ in descending order by length;
    foreach operation sequence $s$ in $S_{p_i}$ do
        if $s$ is not included in any other $s' \in M_{p_i}$ then
            add $s$ to $M_{p_i}$;
    convert $M_{p_i}$ to methods;
```

s_1, and define s_2 and E_{s_2} in the same manner. We assume that "operation sequence s_2 includes s_1" means that the destination of s_1 and s_2 are the same, and E_{s_2} includes E_{s_1}. For example, suppose a web page has web elements $e_1, ..., e_4$, and we set $E_{s_1} = \{e_1, e_2, e_4\}$ and $E_{s_2} = \{e_1, e_2, e_3, e_4\}$, where E_{s_2} includes E_{s_1}. Also, suppose that the destinations of s_1 and s_2 are the same. In this case, s_1 is not adopted as a method because s_2 includes s_1. We only take into account the operated web elements, regardless of the input value, to determine the inclusion. Let M_{p_i} be the operation sequences adopted as methods in the page object for p_i.

Finally, the algorithm converts each operation sequence in M_{p_i} into JavaScript code that uses the APIs of WebdriverIO. Since the algorithm converts operation sequences into methods in this manner, the roles of the generated methods are unlikely to overlap.

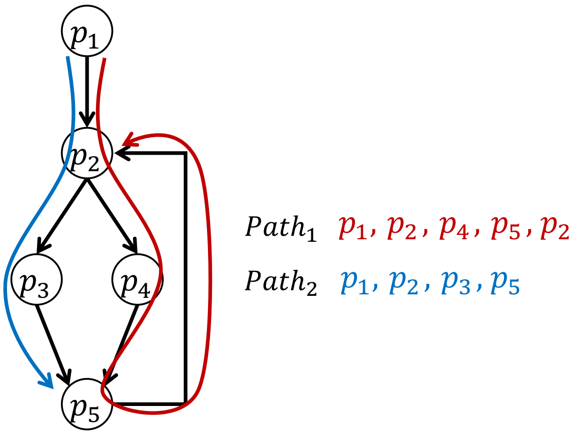

p_1
p_2
p_3
p_4
p_5
$Path_1$ p_1, p_2, p_4, p_5, p_2
$Path_2$ p_1, p_2, p_3, p_5

Figure 3.4: Example of path selection for test-case generation

We now present how to determine the identifiers of classes, accessors, and methods in the page objects. Class names are determined by the title or URL used to define the page. When we use regular expressions to define web pages, each web page can have a user-defined alias. Accessor names are determined by the id, name, or text of the web element. Method names are "go<class name of the destination>" when the method clicks a link at the end of it; otherwise, it is "do<accessor name called lastly>". If the generated identifier name conflicts with other identifiers, the algorithm adds a serial number to the end of the identifier name.

3.4.2 Test-case generation

In addition to page objects, the proposed approach also generates test cases using the page objects. Our approach first determines the paths of page transitions to be checked in each test case (a path is represented as a sequence of pages). A test case is constructed by combining methods defined in the page objects and is executed along one of the determined paths. We note that our test case generation algorithm does not take into account the states of the target application, so the generated tests may not always be executable. This limitation is discussed in Section 3.4.3.

The following presents the algorithm for determining the paths of page transitions. Our approach selects paths that satisfy the following rules:

1. The path covers all page transitions checked during manual testing.

2. If the same pages are visited twice in one path, subsequent pages will not be visited.

3. Page transitions executed in other paths are not executed as often.

Figure 3.4 shows an example of path selection. The web pages $p_1, ..., p_5$ and page

Algorithm 3: Test-case generation

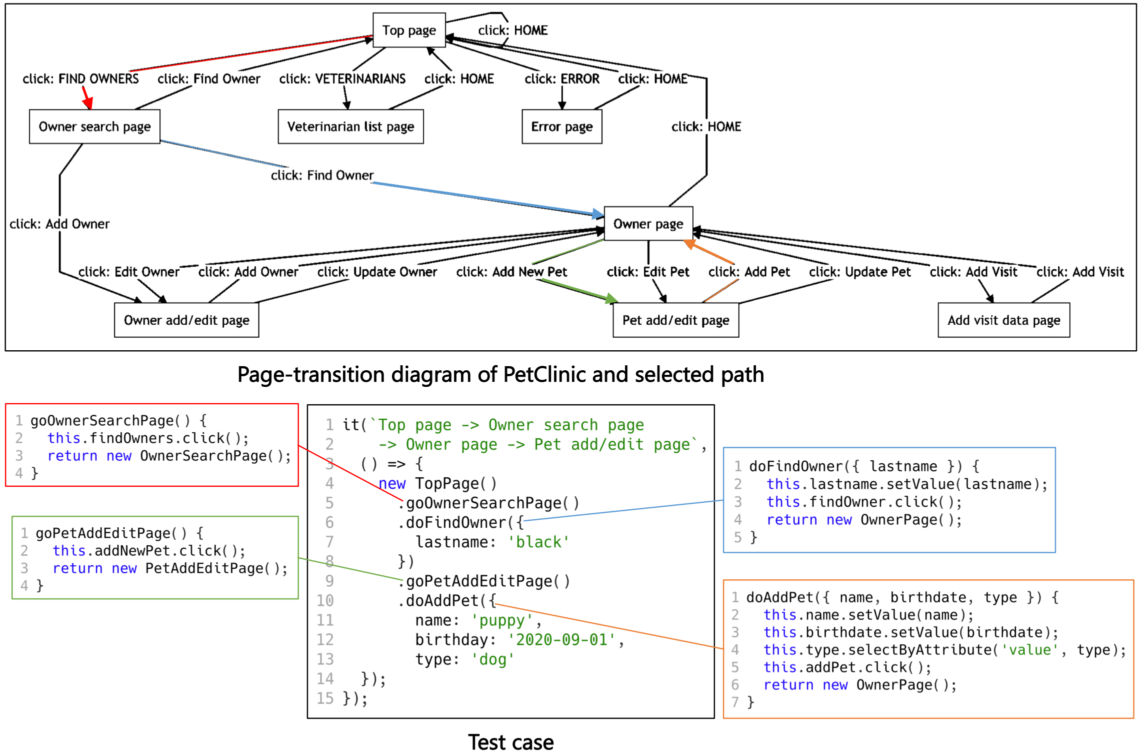
```
Input: A test log and page objects
Output: Test cases
path_list ← $\emptyset$;
path$_c$ ← $\emptyset$;
add the start page to path$_c$;
Construct page transition diagram from the test log;
Function breadthFirstSearch():
    queue (of path) ← $\emptyset$;
    Enqueue path$_c$ to queue;
    while queue is not empty do
        path$_c$ ← Dequeue from queue;
        if all destinations from the last page of path are included in path$_c$ then
            Cut off the redundant page transitions at the end of path$_c$;
            if path$_c$ includes undiscovered page transitions then
                Add path$_c$ to path_list;
        foreach $p_a$ ← adjacent page of the last page of path$_c$ do
            if $p_a$ is not included in path$_c$ then
                path′ ← path$_c$ with $p_a$ appended;
                Enqueue path′ to queue;
breadthFirstSearch();
foreach path in the path_list do
    Convert path to a test case that consists of chained methods;
```

transitions among them are shown. In this case, the rules determine the two paths:

$$\text{Path}_1 = [p_1, p_2, p_4, p_5, p_2],\ \text{Path}_2 = [p_1, p_2, p_3, p_5].$$

Here, Path_1 and Path_2 obviously satisfy the first rule. Next, p_2 is the last page of Path_1, and subsequent pages are not visited. We can see that Path_1 follows the second rule. p_5 is the last page of Path_2, and the page transition from p_5 to p_2 is not executed. Since Path_1 has already passed through the page transition from p_5 to p_2, the third rule rejects the page transition.

On the other hand, both Path_1 and Path_2 pass through the page transition from p_1 to p_2. The page transition is not rejected by the third rule because it is necessary to cover all page transitions with two paths.



Page-transition diagram of PetClinic and selected path

```
goOwnerSearchPage() {
  this.findOwners.click();
  return new OwnerSearchPage();
}
```

```
goPetAddEditPage() {
  this.addNewPet.click();
  return new PetAddEditPage();
}
```

```
it(`Top page -> Owner search page
    -> Owner page -> Pet add/edit page`,
  () => {
    new TopPage()
      .goOwnerSearchPage()
      .doFindOwner({
        lastname: 'black'
      })
      .goPetAddEditPage()
      .doAddPet({
        name: 'puppy',
        birthday: '2020-09-01',
        type: 'dog'
  });
});
```

```
doFindOwner({ lastname }) {
  this.lastname.setValue(lastname);
  this.findOwner.click();
  return new OwnerPage();
}
```

```
doAddPet({ name, birthdate, type }) {
  this.name.setValue(name);
  this.birthdate.setValue(birthdate);
  this.type.selectByAttribute('value', type);
  this.addPet.click();
  return new OwnerPage();
}
```

Test case

Figure 3.5: Example of generated test case and methods that it calls

Algorithm 3 describes the algorithm for test case generation. First, our approach analyzes the test logs to generate a page-transition diagram of visited pages in the test (line 4). Next, we obtain a list of paths satisfying the above three rules by carrying out a breadth-first search on the page-transition diagram (lines 5–21). Depth-first search is also a well-known graph-traversal algorithm, but in this case, a breadth-first search is superior. This is because a breadth-first search determines the shorter paths first, contributing to generating concise test cases. Let the start page of all paths be the start page of the manual testing. When no more page transitions are possible due to the second rule, we cut off the redundant page transitions at the end of the path because of the third rule. A test case consists of chained methods defined in the page objects and executes the page transitions following the path (lines 16–18).

For example, suppose there are three pages p_1, p_2, and p_3. When a path $[p_1, p_2, p_3]$ is converted to a test case, the test case first calls a method defined in the page object of p_1 to go to p_2. The return value of the first method is the page object of p_2, so the test case then calls a method defined in the page object of p_2 to go to p_3. In this manner, our approach generates test cases that satisfy the rules by converting each path into a test case.

Note that if multiple methods execute the same page transition in one page object, the first generated method is used in the test case. Our approach can also generate arguments and input values for the method because the test logs contain input values when each page transition is carried out by testers.

Figure 3.5 shows an example of a generated test case and the methods called from the test case. The page-transition diagram and test case is a part of the output in our experiment using PetClinic described in Section 3.5.

Suppose we obtain a path that transitions in the following order: the top page, owner search page, owner page, and pet add/edit page. In this case, our approach generates a test case that consists of four methods executing the page transitions. Each method is declared in a different page object. The test cases start from the page object of the top page, and the page object has the `goOwnerSearchPage()` method. Next, `goOwnerSearchPage()` returns the page object of the owner search page, and the page object calls the `doFindOwner()` method. By repeating the same steps, the test case is built. If methods require arguments, the proposed approach extracts a set of input values that caused the required page transition from the test log.

3.4.3 Limitation

Our approach in this study may generate test cases that require the application to be in a certain state in order to be executed. To execute such test cases, we would need to insert a process to initialize the database or modify the input values in the generated test scripts. This state dependency issue is a common problem in many crawling-based test generation techniques and record & replay tools, but it is outside the scope of this study. There are existing studies [59, 60] on dependency-aware test generation that could potentially be used to address this problem. If these techniques are not adopted, users will have to manually modify the generated test scripts to solve the state dependency issue. However, using page objects in our test scripts makes them easier to modify, which partially mitigates this problem. One of the goals of this research is to generate test scripts that are easy to modify, as it is often difficult to create perfect test scripts for users.

Our approach has some limitations in terms of the types of applications and situations it can be applied to. Currently, our approach does not generate any assertions, so users will need to insert their own assertions to verify that the test scripts are being executed correctly. However, it is technically possible to verify that the currently opened web page matches the expected one, as we record the titles and URLs of the web pages where operations are performed. Additionally, the generated test scripts use the page-object pattern, which makes them highly maintainable and easy to add more detailed assertions to.

Our recording tool is currently only able to record click and input operations, so it is not possible to perform other types of operations (such as drag, mouse hover, etc.) in the generated test scripts. Furthermore, our approach cannot generate methods that include operations that were not performed during manual testing. However, we believe that by keeping logs of not only planned manual testing but also some behavior verification and smoke testing, we can compensate for the lack of test logs.

3.5 Evaluation

We conducted experiments to evaluate the effectiveness of our approach in generating test scripts. Specifically, we sought to answer the following research questions:

RQ1. Can our approach generate a greater number of complete methods in page objects than the current APOGEN approach?

RQ2. Can the test cases generated with our approach be used without modification and cover the features of the application?

RQ3. Does our approach reduce the initial cost of test script implementation compared to practical approaches used in industry?

3.5.1 Experimental setup

Four testers conducted manual testing on Spring PetClinic version 2.2.0, which is an open-source web application. Spring PetClinic is a sample application of the Spring Framework

Table 3.1: Web pages and features of PetClinic

Web page	Feature
Top page	Nothing
Owner search page	Search owners by a last name
Owner search result page	Show the list of owners hit by a search
Owner add/edit page	Input owner data and add or update the owner
Owner page	Add or edit pet data of the owner and add visit data for the pet
Pet add/edit page	Enter pet data and add or update the pet
Visit data add page	Enter visit data for the pet and add them
Veterinarians list page	Nothing
Error page	Nothing

and has more than 6k lines of Java code. Table 3.1 lists all the web pages and features of PetClinic. We used the database prepared by PetClinic as the initial state.

All the web pages in PetClinic have a header with links to go to the top page, owner search page, veterinarians list page, and error page. We consider the owner add page and the owner edit page to be the same page because these two pages are generated from the same template file in the Spring Framework and have a similar structure. We also consider the pet add/edit page to be a single page for the same reason. Additionally, each owner has a separate owner page in PetClinic, but we also treat these pages as a single page. Since these page objects would need to be modified in the same way frequently when the template is modified, we believe it would be more practical to separate these page objects for experimentation.

In this experiment, we used two different manual testing approaches to evaluate whether our approach does not depend on the specific manner of manual testing. Two testers performed scripted testing, while the other two performed exploratory testing. All four testers had more than three years of testing experience, and the two conducting exploratory testing had experience with this type of testing.

Before conducting the experiment, we had the testers familiarize themselves with the specifications of PetClinic by operating it. We also assumed that PetClinic had been adequately unit tested on both the server-side and client-side. Next, we instructed the testers on how to conduct end-to-end testing on PetClinic to check its functionality and usability. Even though PetClinic is a stable application, we asked the testers to test it with the aim of finding bugs.

The two testers who performed scripted testing designed and documented the content of the tests in advance as test scenarios. A test scenario is a sequence of steps to test a

Table 3.2: Summary of tests by testers

Tester	Approach	# of test scenario	# of operations
A	Scripted testing	9	135
B	Scripted testing	20	258
C	Exploratory testing	–	378
D	Exploratory testing	–	505

specific use case of the target application. The other two testers performed exploratory testing for up to 30 minutes to find bugs using their knowledge and experience.

All the operations performed during the tests were recorded using our tool, which is described in Section 3.4. Let the test logs obtained from testers A–D be test logs A–D, respectively. Table 3.2 summarizes the tests conducted by the testers. The cases of exploratory testing did not have documented test scenarios because the two testers did not design the tests in advance. The number of operations is equal to the number of click events and change events that occurred when the testers interacted with web elements.

We applied our approach to the test logs and generated four sets of test scripts. We also merged the four test logs to create a single large test log that was equivalent to four tests conducted consecutively. We applied our approach to the merged test log in the same way. The generated test scripts from our approach and APOGEN are publicly available[1].

3.5.2 Page-object generation

We compared our approach with APOGEN to evaluate whether the page-object-generation phase of our approach was able to generate complete methods. We first examined the page objects generated with our approach and APOGEN. PetClinic has nine pages, as shown in Table 3.1. Note that we define owner pages, owner add/edit page, and pet add/edit page each as one page using regular expressions on the URLs. This is because these pages are generated from the same template, as explained in Section 3.5.1. Therefore, our approach generated nine page objects from each test log.

Next, we applied APOGEN to PetClinic to generate page objects. We provided information to APOGEN's crawler to reach as many pages as possible and manually classified the reached web pages into the pages shown in Table 3.1. However, we were unable to generate page objects for the error page and owner search page due to limitations of APOGEN. This result is the same as in the evaluation of an existing paper [51]. As a result, we obtained seven page objects with APOGEN, excluding the two pages that could not be reached.

[1] https://zenodo.org/record/5655786

Table 3.3: Classification of methods in page objects generated with our approach and APOGEN

Source	Complete	Redundant	To modify	Unnecessary	Header	Total
APOGEN	5	0	6	0	18	31
Test log A	6 (9)	0	3	0	6 (7)	13 (17)
Test log B	8 (12)	0	7	1	8 (12)	24 (32)
Test log C	6 (10)	0	4	6	12 (16)	28 (36)
Test log D	9 (13)	1	6	3	10 (13)	29 (36)
Merged log	12 (16)	0	3	11	18 (23)	44 (53)

We then classified the methods in the page objects according to the following criteria:

Complete The methods have no parts that need to be modified in terms of arguments, operations, and return values.

Redundant The methods function correctly but contain unnecessary operations that do not affect their functionality.

To modify The methods require modification of arguments, operations, or return values in order to use them.

Unnecessary The second or subsequent methods of multiple methods that check the same page transitions.

Header The methods click links on the header to go to another page.

Although methods classified as *header* are all *complete* methods, we decided to distinguish *header* from the others. This is because they are unlikely to be used in actual test cases despite their large number.

Table 3.3 shows the results of the classification. The table shows both the case where we counted only the methods in the seven page objects generated by APOGEN, and the case where the two page objects that APOGEN could not generate were included. The values in parentheses are counted by including the number of methods in the page objects that could not be generated by APOGEN.

Our approach generated a greater number of complete methods than APOGEN, even when we excluded the page objects of the web pages that APOGEN could not reach. It also generated one *redundant* method `goOwnerSearchPage()` for test log D. This is because tester D performed the operation sequence to click a link to go to the owner search page after filling an input field on the owner add/edit page. The operation sequence was converted to a method, but the operation of filling an input field was not necessary to go to the owner search page.

The *to modify* methods were generated by both APOGEN and our approach, but our approach tended to generate fewer of them. There were no correct page objects as return values in four *to modify* methods generated by APOGEN, probably because APOGEN does not take into account the case where different page transitions are performed depending on the input values when using the same feature. The other two *to modify* methods by APOGEN lack operations to enter values when updating pet or owner information. On the other hand, our approach was able to generate these methods that APOGEN could not generate correctly. APOGEN converts operations on web elements enclosed in `<form>` tags into a method, but PetClinic did not have such sets of web elements. Hence, there was no sequence of operations that our approach could recognize but APOGEN could not.

Most of the *to modify* methods generated by our approach have an insufficient number of arguments and cannot enter values into some input fields. This is because the testers did not fill in all input fields on some pages during the tests. If the testers had conducted a test that attempted to register an owner with a blank name, the generated method would not have included the operation to fill in the name input field due to the method-generation algorithm of our approach. However, there are other possible inputs that could cause registration to fail, such as not giving an address and inputting incorrect characters. The methods should always have arguments for all inputs to register an owner because missing arguments reduce versatility. If there are no missing arguments and users want to register an owner with a blank name, they can achieve this by providing an empty string as an argument.

Our approach generated *unnecessary* methods that separately click different owners on the owner search result page in most cases. Since these methods have the same destination, the second and subsequent methods are classified as *unnecessary.* The *unnecessary* methods were only generated by our approach. However, if APOGEN reached the owner search result page, it would generate many methods to click each owner and generate more *unnecessary* methods than our approach.

Our approach generated fewer *header* methods, even though it generated more page objects. Most of the methods classified as *header* are not important and would not be used because developers usually just need to make sure the links are valid.

Our approach to generating page objects from the merged log results in the most *complete* methods and the fewest methods *to modify.* This is because each log fills in missing operations, and our approach converts operation sequences that include other small operation sequences into methods. As a result, even if a log does not include operations on all input fields, our approach can generate a *complete* method if all input fields are operated on in the other logs.

Table 3.4: Classification and average length of generated test cases

Source	Complete	Data-dependent	To modify	Total	Avg. length
Test log A	7	0	1	8	4.25
Test log B	11	0	1	12	4.00
Test log C	10	4	0	14	4.58
Test log D	11	0	2	13	4.50
Merged log	19	0	1	20	4.46

However, generating page objects from the merged log can also lead to the generation of many *unnecessary* and *header* methods. This is because the merged log includes many operations for clicking various elements on the owner search result page and clicking links in the header on each page. This problem may not occur in other applications, as it is largely due to the specific design of PetClinic.

To summarize the evaluation of page-object generation and answer RQ1, our approach to page-object generation is more likely to generate a greater number of complete methods compared to APOGEN, regardless of the manual testing approach or testers. However, generating page objects from a merged log can compensate for the incompleteness of each log, but it also increases the number of extra methods that may not be used.

3.5.3 Test-case generation

We next evaluated the ability of our approach to generate complete test cases during the test-case-generation phase. The choice of which test cases to automate depends on the project, but in this experiment, we evaluated whether our approach could generate test cases that check the normal scenarios for each feature without requiring modifications. This is because such test cases are versatile and can be useful in any project. Additionally, it is easy to implement test cases for exceptional scenarios (e.g., cases where owner registration fails) by reusing the generated test cases and page objects. We classified the test cases generated from test logs A–D according to the following criteria:

Complete A test case can be executed without modifying the order of method calls, argument values, and the database state.

Data-dependent A test case can be executed by changing the database state from the initial state or changing the value given by the method argument.

To modify A test case can be executed by replacing some called methods with other methods that have the same transition destination as before.

Table 3.4 shows the results of the classification of generated test cases and the average length of the test cases. The length of a test case refers to the length of paths of page transitions checked in a test case. It also equals the number of called methods in a test case, as a method call invokes a page transition.

The reason why four test cases of test log C were classified as *data-dependent* is that tester C added a pet to an owner registered during manual testing. Since our approach does not take into account the state of the application, as explained in Section 3.4.3, it generated test cases that add a pet to an owner who does not exist in the initial state of the database. The testers other than C only tested the edit feature for users and pets registered by default, so this problem did not occur when using logs other than C. We found that whether or not our technique generates *data-dependent* test scripts depends on the way of testing. These *data-dependent* test cases can be turned into *complete* test cases by replacing the method call in the test case with the method to click an initially existing owner, or by changing the initial state of the database.

Some test cases were classified as *to modify* because some web pages with different features were defined as one page. For example, adding and editing pets are different operations, but we define the pet add/edit page as a single page because the templates of the pages are the same. The method for clicking the "Add Pet" button after filling in input fields and the method for clicking the "Update Pet" button after that are declared as different methods in the page object. However, our approach did not distinguish these methods when constructing test cases because both methods go to the owner page from the pet add/edit page. As a result, our approach may generate test cases that call the method to update a pet when the method to add a pet should be called. In this case, the test case becomes *complete* if we replace the method call to update a pet with the call to add a pet.

Table 3.5 shows which features were checked by the test cases generated from test logs A–D and the merged log (labeled "M"). The features of PetClinic were extracted from the test-case specifications written by testers A and B. In the table, a "✓" indicates that the generated test cases checked the feature, "×" indicates that the generated test cases did not check the feature even though the manual test did, and "–" indicates that the generated test cases could not check the feature because the manual test did not check it. Due to the limitation of our approach, it is not able to generate test cases for features that were not checked by the manual tests. We assume that one test case can confirm multiple features. For example, we have a test case that adds a pet to an owner found in the owner search after moving from the top page to the owner search page. In this case, we determine that the test case confirms features (1, 6, 12) in Table 3.5. Note that Table 3.5 shows the results when the *data-dependent* and *to modify* test cases were correctly modified and became complete.

Table 3.5: Features confirmed from generated test cases

Web page	#	Feature	A	B	C	D	M
Owner search page	1	If one hit is made in the owner search, the owner page will be displayed.	✓	✓	✓	✓	✓
	2	If two or more hits are made in the owner search, they will be displayed on the owner search result page.	–	✓	×	×	✓
	3	If nothing is entered in the owner search, all owners will be displayed on the owner search result page.	✓	×	✓	✓	×
	4	Go to the owner add page.	✓	✓	✓	✓	✓
Owner search result page	5	Move to the owner page by clicking an owner name.	–	✓	✓	✓	✓
Owner page	6	Go to the pet add page.	✓	✓	✓	✓	✓
	7	Go to the pet edit page.	×	×	×	×	×
	8	Go to the owner edit page.	✓	✓	✓	✓	✓
	9	Go to visit data add page.	✓	✓	✓	✓	✓
Owner add/edit page	10	Add an owner by filling in input fields.	✓	✓	✓	✓	✓
	11	Edit an owner by filling in input fields.	×	×	×	×	×
Pet add/edit page	12	Add a pet by filling in input fields.	✓	✓	✓	✓	✓
	13	Edit a pet by filling in input fields.	×	×	–	×	×
Visit data add page	14	Add visit data by filling in input fields.	✓	✓	✓	✓	✓
Header	15	Go to the top page, owner search page, veterinarians list page, or error page.	✓	✓	✓	✓	✓

Some features were not checked by the generated test cases even though the manual tests checked them. In most cases, this was because our approach generates test cases based on the coverage of page transitions. For example, in a certain test case, if a transition from the owner page to the pet add/edit page was performed by clicking the “Add New Pet” button, the page transition was checked. However, when the “Edit Pet” button was clicked from the owner page, the transition to the pet add/edit page was also performed. Since this page transition had already been checked, our approach did not generate a test case to check the feature for editing pets. As a result, for the pairs of features (2, 3), (6, 7), (10, 11) and (12, 13) in Table 3.5, only one feature of each pair was checked. However, we believe that we can easily create test cases to check the other feature of each pair by slightly modifying the generated test cases.

Finally, we discuss the smallness and simplicity of the generated test cases. Table 3.5 shows that the average length of the test cases was at most 4.58. This indicates that each test case is concise and that users can easily understand them. An interesting point is that test log D had about twice as many operations as test log B, and testers B and D adopted different manual testing approaches, yet the numbers of test cases were almost the same. Since the generated test cases depend on the page-transition diagram obtained from the manual tests, our approach has the advantage of generating similar test cases no matter how the manual tests were conducted, as long as the page-transition diagrams are similar.

In this experiment, although the test cases generated from test log A were the smallest, they covered most of the features checked in the other test cases. Therefore, we can say that there is redundancy in the test cases generated from the other test logs. This is because the more links on the header are clicked, the more complex the page-transition diagram becomes. Our approach uses the page-transition diagram to make the test cases cover the page transitions executed in the tests. However, every page of PetClick has a header, and if testers go to another page by clicking the links on the header, the page transitions are regarded as different. Hence, we found that our approach may generate a redundant set of test cases if applications have mesh-like page transitions that are interconnected.

To answer RQ2, our approach generated complete test cases in most situations. The generated test cases covered most of the features of the application. However, our approach may generate incomplete or redundant test cases when multiple pages with different features are treated as the same one, or when the application has interconnected page transitions.

3.5.4 Empirical evaluation

We evaluated whether our approach is efficient for implementing test scripts using page objects at a lower cost than existing approaches. For comparison, we chose to implement

Table 3.6: Summary of the test scenarios for empirical evaluation

	System	# of test procedures	# of involved pages	Description
Scenario 1	A	3	4	Check data query feature
Scenario 2	A	2	14	Check data update feature
Scenario 3	A	2	14	Check data lifecycle
Scenario 4	B	2	6	Check standard operation procedures

Table 3.7: The time to implement test scripts (minute)

	Our approach				SeleniumIDE				Manual			
	Rec[1]	PO[2]	TC[3]	Total	Rec	PO	TC	Total	Rec	PO	TC	Total
Scenario 1	1	7	1	9	1	16	1	18	0	21	1	22
Scenario 2	3	26	2	31	3	41	2	46	0	66	2	68
Scenario 3	3	60	2	65	3	55	2	60	0	94	2	96
Scenario 4	1	5	1	7	1	20	1	22	0	28	1	29
Total	8	98	6	112	8	132	6	146	0	209	6	215

[1] Time to record manual tests

[2] Time to create or modify page objects

[3] Time to create or modify test cases

test scripts manually and with SeleniumIDE, which are commonly used in real-world software development. The target systems are an internet banking system (System A) and a campaign information management system (System B), which were developed in a real project of a partner company. We prepared three test scenarios for System A and one test scenario for System B. Table 3.6 shows a summary of each scenario. Each test scenario has multiple predetermined test procedures.

We asked one developer from the partner company to carry out the tasks of implementing test scripts to automate the predetermined test procedures. The developer was familiar with our approach, SeleniumIDE, and how to implement test scripts with page objects. They also had a detailed understanding of the target systems. The condition for task completion was that the developer implemented test scripts and confirmed that they could automate the predetermined test procedures. The test procedures were also complied with when recording the tests with SeleniumIDE and our recording tool. The test script implementation tasks were carried out in the following order:

(i) Manual implementation: The developer implemented test scripts with page objects written in JavaScript.

(ii) SeleniumIDE: The developer recorded tests, exported them as test scripts written in JavaScript, and then rewrote them into test scripts with page objects.

(iii) Our approach: The developer recorded tests, generated test scripts using our approach, and modified them to automate the predetermined test procedures.

The developer was not allowed to use test scripts implemented in the previous tasks in the later tasks. Carrying out the previous tasks was likely to make the later tasks easier, which may not result in a fair outcome. We will discuss this issue in Section 3.6.

Table 3.7 shows how many minutes it took to finish each task. The results show that the proposed approach reduced the time for implementing the test scripts by 48% compared to manual implementation and by 23% compared to using SeleniumIDE. Most of the task time was spent on creating or modifying page objects. The time spent on recording the operations and creating or modifying the test cases was relatively small. The reason why it took less time to create or modify the test cases is that they can be written easily as a combination of methods in page objects, and the number of test cases is small. When using our approach, the largest amount of time (42.3%) was spent on modifying the source code to fix the methods in the page objects. The time spent on correcting locator errors (34.7%) followed this. Other modifications included adding commands to wait for web pages to load and removing unnecessary test steps.

The reason why the page objects generated by our approach required modifications was due to the complexity of System A. Depending on its internal state, the page transitions may change even if the same operation is carried out. In addition, System A has web pages that change drastically and dynamically using JavaScript. Our approach currently cannot handle such internal states of applications and drastic screen changes. If page objects are not correctly associated with each page, the generated page objects require significant modifications. However, despite the need for modifications to the generated test scripts, the results show that using our approach is more efficient than implementing from scratch. Our approach could potentially solve such problems by making it possible to define screens more flexibly, for example, by defining pages using strings rendered on web pages. Alternatively, using more advanced screen recognition techniques proposed in research such as [61, 62] could potentially improve the performance of our approach.

The main reason why the developer needed to fix the locators was that the locators generated by our approach did not uniquely identify the web elements in a web page in some cases. Since our approach does not collect any information other than the web elements operated during the manual testing, the generated locators may not be unique. This problem can be solved by considering all web elements in a web page to generate locators during manual testing. We believe that these improvements will further reduce the time required to implement test scripts using our approach.

To answer RQ3, our approach has the potential to reduce the cost of test script implementation in real-world software development. Additionally, improving the algorithm of our approach could potentially reduce costs even further.

3.6 Threats to Validity

The external validity of our study refers to the generalizability of our findings. First, we only used PetClinic as the target to evaluate the proposed technique. Different results from those in this study may be obtained if we apply the proposed approach to other applications. In this study, we chose PetClinic since it was used in an existing paper [51] to compare our approach with APOGEN. Next, the results of our experiment depended on the content of the testers' manual testing approach. Our experiments showed that our approach can generate a greater number of complete methods and test cases for a variety of testing approaches. However, when other testers conduct manual tests, we may not obtain similar results. In addition, the proposed technique may not work well if manual testing is not performed sufficiently. If the proposed technique is applied to an application more complex than PetClinic, testers may miss features to be tested. Even if the manual testing is sufficient, the generated test scripts may not be able to be executed due to the state-dependency problem when some operations in the manual testing depend on past ones.

In the empirical evaluation, only one developer carried out the test script implementation tasks. We may obtain different results from this evaluation if we have more developers carry out the same tasks. In addition, doing the previous tasks may make the later tasks easier, so it is possible that the time to carry out tasks using our approach is shorter than it should be. However, we believe that the effect of the previous tasks on the evaluation is small because the tasks assigned to the developer are simple compared to usual test script implementation. In usual test script implementation, developers often implement test scripts through trial and error. In our experiment, on the other hand, the test procedures were predetermined and the developer understood the details of the systems under test. Moreover, the developer did not spend time on properly naming identifiers and refactoring, other than converting predetermined test cases into test scripts. This would have made it clearer how to implement test scripts in many parts.

A threat to the internal validity of our study is that we defined the web pages and features of PetClinic ourselves and classified the generated page objects and test cases. The definition of the features was based on the test specifications written by testers A and B, so we believe that the definition is objective to some extent. To address this threat, We will make the output of our approach publicly available so that other researchers can verify the results.

3.7 Conclusion

In this study, we proposed a novel approach to generate test scripts using the page-object pattern from manual testing logs. Through experiments, we showed that our approach is able to solve the problems with existing approaches and generate a greater number of complete methods in page objects. Our approach also generates test cases that leverage the generated page objects and cover most of the features of the application under test. The generated test cases and page objects are reusable, and users can easily add new test cases to them.

In addition, our empirical evaluation demonstrated the potential for reducing the cost of test script implementation in real-world software development. Our results showed that our approach can reduce the time required for implementing test scripts by up to 48% compared to manual implementation and by up to 23% compared to using SeleniumIDE. This indicates that our approach is effective at reducing the cost of implementing and maintaining test scripts by generating useful test scripts through the conducting of only manual testing, which is essential in software development.

For future work, we aim to generate more complete page objects and test cases by converting operation sequences into methods more precisely using the testers' knowledge contained in the test logs effectively. We also plan to make our recording tool publicly available so that everyone can record their testing activities. By making the test logs publicly available, researchers will be able to mine the data and use it for a variety of purposes beyond test automation.

Chapter 4

Web Element Identification using NLP and Heuristic Search

4.1 Introduction

In recent years, the timely updating of software has become increasingly important in order to respond to rapid changes in market conditions. Developers need to verify that their software works properly before release. The cost of regression testing can be overwhelming in software maintenance [36, 37]. Test automation is therefore an important technique for reducing this 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, as shown in a study by Dobslaw et al. [6]. The authors investigated the return on investment (ROI) of end-to-end test automation frameworks and found that, compared to manual testing, the initial implementation time accounted for nearly 90% of the total cost 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 rely on the metadata of web elements and the structure of web pages.

For example, Selenium [2], 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 documents, while 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 in order to determine the appropriate locators. In this way, the implementation of test scripts can be obstructed by the reliance on metadata and the structure of each web page.

The dependence on metadata and the structure of web pages is also an obstacle to maintaining test scripts. Test scripts that use locators are known to be fragile, as shown by previous research [7, 8]. From these studies, it is clear that using locators can increase the cost of maintaining test scripts and hinder efficient regression testing.

Another major challenge with end-to-end testing is the cost of creating and maintaining test cases. Note that this study defines a test case as a specification of test procedures and expected results, and a test script as an automated program to verify the specification. Writing test cases is important because not all tests can be automated, and not everyone involved in testing may understand test scripts written in a programming language. Test cases also require maintenance, and if both test cases and test scripts exist, developers need to keep them consistent. As a result, it can be costly to create and maintain test cases, especially for fast-evolving applications.

One efficient way to address the challenges discussed above is to make test cases executable without the need for test scripts. This would relieve developers from the problems of implementing test scripts and maintaining consistency between test cases and test scripts. Our goal is to make it possible to execute test cases written in natural languages without the need for conventional test script implementation using locators. To achieve this goal, it is first necessary to be able to identify web elements from test case descriptions without relying on the implementation of the application. In this study, we propose a technique for identifying web elements to be operated on web pages by interpreting test cases. The test cases we focus on are written in a domain-specific language (DSL) without relying on metadata of web elements or the structural information of web pages. We use natural language processing (NLP) techniques to understand the semantics of web elements and test cases and create heuristic search algorithms to find promising test procedures from the 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 approximately 94% of the 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 procedures are publicly available[1].

The contributions of this study are as follows:

- We propose a novel technique for identifying web elements by interpreting test cases that are written in a domain-specific language that is close to natural language.
- We propose an algorithm that combines natural language processing (NLP) and heuristic search to find promising test procedures.
- Our experiments demonstrate the potential for semantic-based identification of web elements and reuse of test cases across multiple contexts and applications.

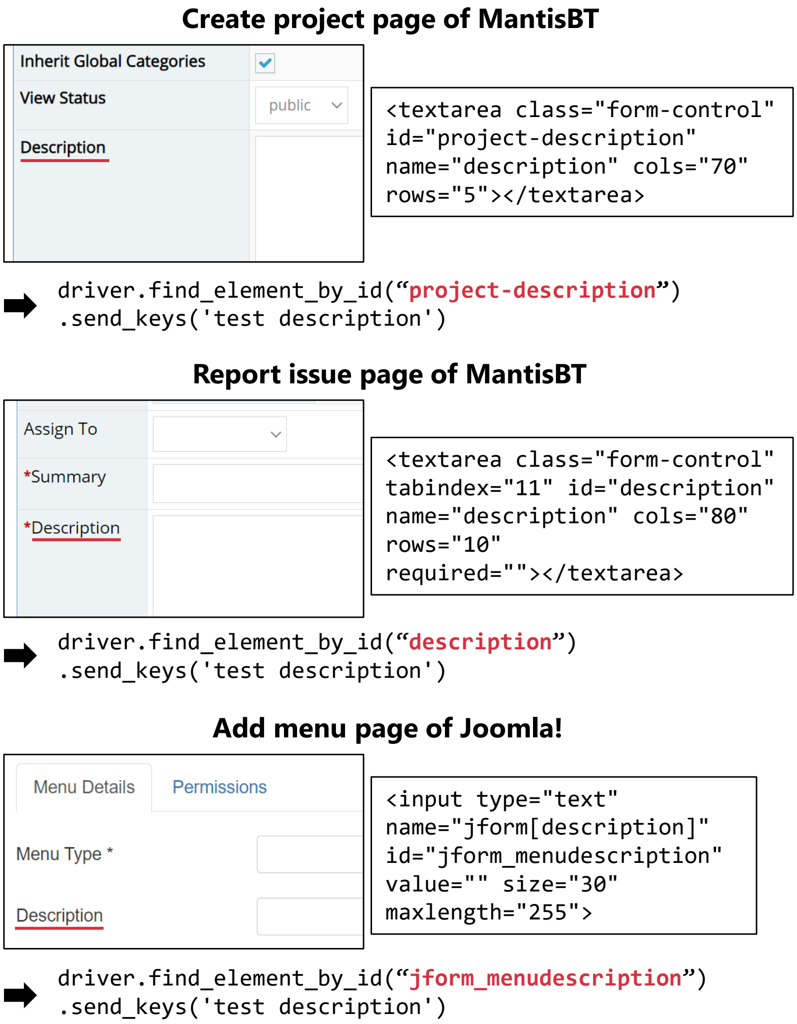
4.2 Motivating Example

In this section, we provide some examples of the problems that can arise when implementing test scripts using locators, as discussed in Section 4.1. Figure 4.1 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 serve similar purposes, it is necessary to use different locators when scripting them because they are implemented differently. These differences in test script implementation can make it diffi-

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

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

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



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!

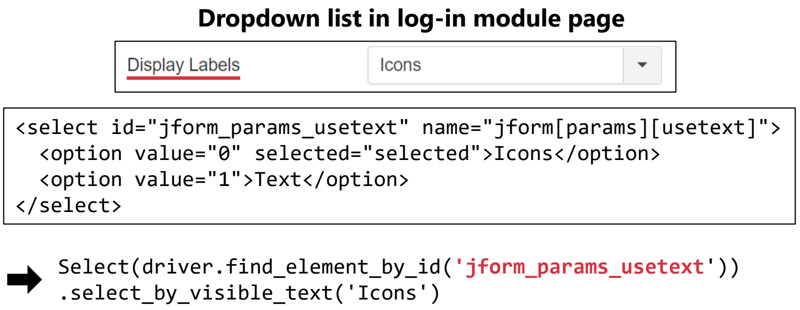
```
<input type="text"
name="jform[description]"
id="jform_menudescription"
value="" size="30"
maxlength="255">
```

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

Figure 4.1: *Description* input fields and Python snippets to enter the value “test description”

cult to reuse parts of test scripts across different contexts. If all these web elements could be represented by the word “description”, it would enable the reuse of parts of test scripts.

Figure 4.2 shows a drop-down list in the log-in module page of Joomla! and a Python snippet using Selenium to select the value “Icons”. Despite the drop-down list being labeled as “Display Label”, the id and name attributes of the web element do not seem to be related to it. This can make it difficult for developers to understand what the snippet means when they read it. If a web element does not have an id or name attribute, the same problem can arise because XPath or CSS selectors would have to be used as a locator. In this case, we suggest using the string “display labels” to identify this drop-down list.



```
<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')
```

Figure 4.2: A drop-down list in the log-in module page of Joomla! and a Python snippet to select the value “Icons”

4.3 Approach

The proposed technique interprets test cases written in a DSL that is 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 an operation that identifies a web element that might be represented by “username” and enters “admin” into it.

Figure 4.3 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 in order to understand their semantics. We also use a heuristic search algorithm to consider multiple test procedures and find the most promising one.

Table 4.1 shows the specification of our DSL. Our DSL can currently handle only simple operations such as clicking, inputting, and selecting. *open* command opens a specified URL in a browser and is generally called at the beginning of the test case. *click*, *enter*, and *select* commands operate a certain web element. These operations contain a *target* to specify the web element to be operated. Let us call this string that specifies the web element *target string*. The target string can be any user-specified string, regardless of the implementation of a web page. *enter* and *select* commands also include a *value* to be entered into the input field or selected from a drop-down list.

4.3.1 Vectorization

In order to determine the appropriate test procedure, we need to identify the web element that corresponds to the target string specified in the test case. To do this, we measure the similarity between the web elements and target strings.

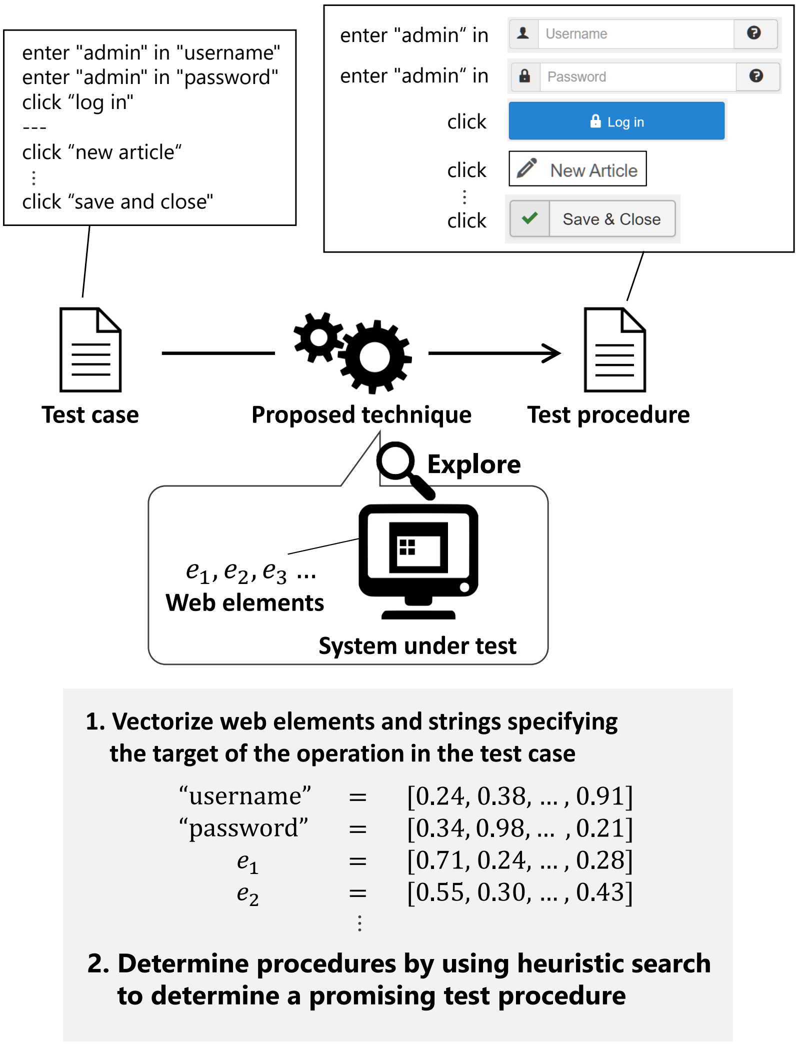

enter "admin" in "username"
enter "admin" in "password"
click "log in"
---
click "new article"
⋮
click "save and close"
enter "admin" in
enter "admin" in
click
click
⋮
click
Username
Password
Log in
New Article
Save & Close
Test case
Proposed technique
Test procedure
Explore
e_1, e_2, e_3 …
Web elements
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

Figure 4.3: An overview of our approach

One approach we use is to vectorize both web elements and target strings in order to represent their semantics. Word embedding techniques, such as Word2Vec, fastText, and GloVe, are often used to represent the semantics of a word or sentence as a vector.

However, web elements often contain information that is irrelevant to their semantics, so we devise a specific approach to represent the semantics of web elements. First, we separately extract the values of attributes and visible texts from a web element. Visible texts include the inner text of the element, as well as any labels associated with the element through the use of the `for` attribute. The `for` attribute specifies which web element a label is bound to, allowing us to identify the label that represents the element.

We separate attributes and visible texts because we believe that visible texts more directly represent the semantics of the web element, and are therefore more important

Table 4.1: 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 4.3.2

than attribute values. In this process, we ignore some attributes that are primarily used for visual layouts, such as the `class` and `style` attributes.

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 4.4 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:

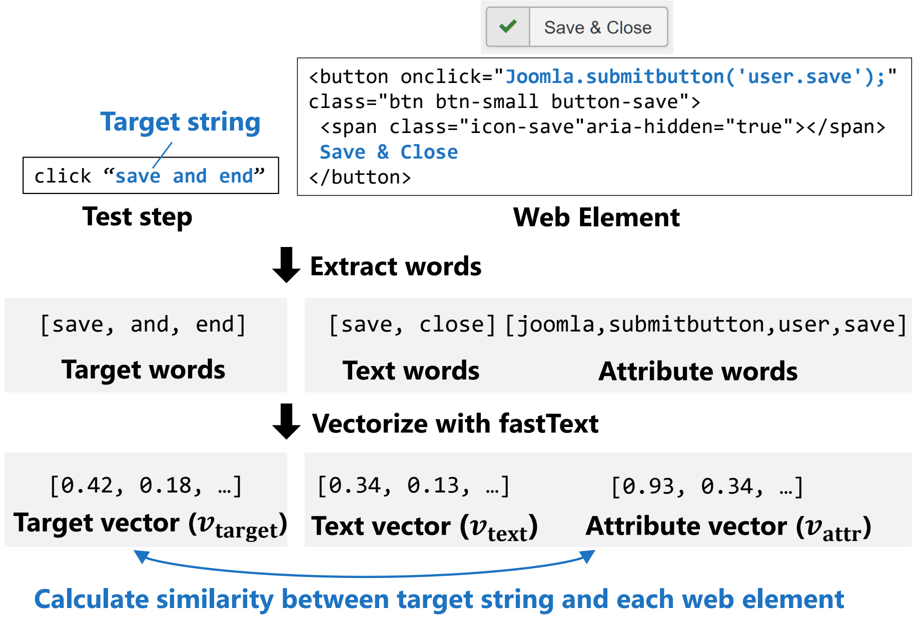
$$[\mathrm{save}, \mathrm{close}]$$

and the attribute words:

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

Next, we convert these words into vectors representing their semantics. Among the available word-embedding algorithms, we selected fastText [63] 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 for this task.

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



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

Figure 4.4: An example of web element vectorization

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 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 as follows:

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

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.

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:

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

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

The method for vectorizing target strings is almost the same as that for vectorizing 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:

$$\text{similarity}(t, e) = \frac{\alpha \times \text{sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{text}}) + \text{sim}(\boldsymbol{v}_{\text{target}}, \boldsymbol{v}_{\text{attr}})}{\alpha + 1} \tag{4.1}$$

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

4.3.2 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. (4.1). This is because it is uncertain whether the vector representation of the web element correctly represents its semantics.

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 problem 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 in order to obtain web elements. However, because web pages may include static or dynamic page transitions, our technique needs to execute each test step each time to properly execute the expected page transitions. If a test step executes an incorrect page transition, the subsequent test steps will not be able to reach the expected web page and will therefore be ineffective.

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

Page-level search

The page-level search algorithm helps to accurately determine a test procedure that is relevant to a single web page. To clarify which test steps are relevant to a single web page, we introduce the page separator `"---"` in our DSL. The page separator ensures that all web elements that are operated in the test steps between two separators are rendered on the web page when the page is loaded. This is useful because it allows us to confirm 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 that correspond to the target strings in test steps relevant to a web page. First, the algorithm calculates the similarities between all possible pairs of a target string and a web element on a particular web page. Next, it calculates scores for permutations of web elements that correspond to the target strings. We call this score a *page-wise score.* 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 that can be operated in the test step are limited to input fields. When a permutation is selected, the operation procedure for the web page is determined. We call this a *page-wise procedure.*

Figure 4.5 shows an example of page-level search. In this example, the web page has a *password* field e_1 and a *confirm password* field e_2. Two test steps are given for operating on the input fields. The similarities between the target strings and the web elements can be calculated using the algorithm described in Section 4.3.1. There are two possible permutations in this example: “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. The page-level search algorithm helps to correctly determine the test procedure that is relevant to a web page.

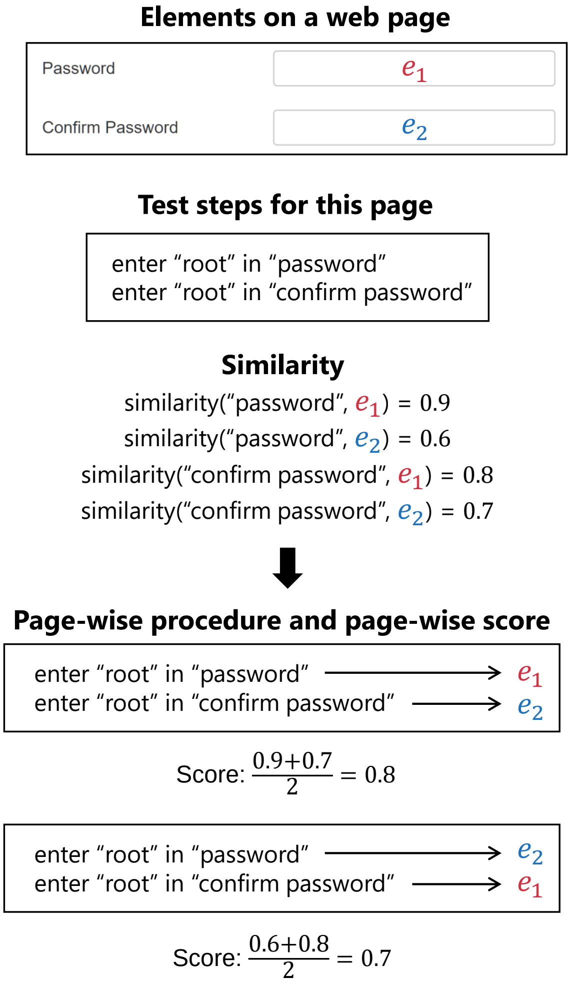

Elements on a web page
Password
e_1
Confirm Password
e_2
Test steps for this page
enter "root" in "password"
enter "root" in "confirm password"
Similarity
similarity("password", e_1) = 0.9
similarity("password", e_2) = 0.6
similarity("confirm password", e_1) = 0.8
similarity("confirm password", e_2)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

Figure 4.5: An example of page-level search

Transition-level search

We obtained multiple page-wise procedures with page-wise scores by applying the page-level search algorithm. However, it is not enough to only perform the page-level search because the page-wise procedure with the highest page-wise score is not always correct. The transition-level search algorithm explores multiple possible sequences of page-wise procedures. It helps to determine a promising test procedure throughout the entire test case.

Figure 4.6 shows an example of transition-level search. In this example, 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 one 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

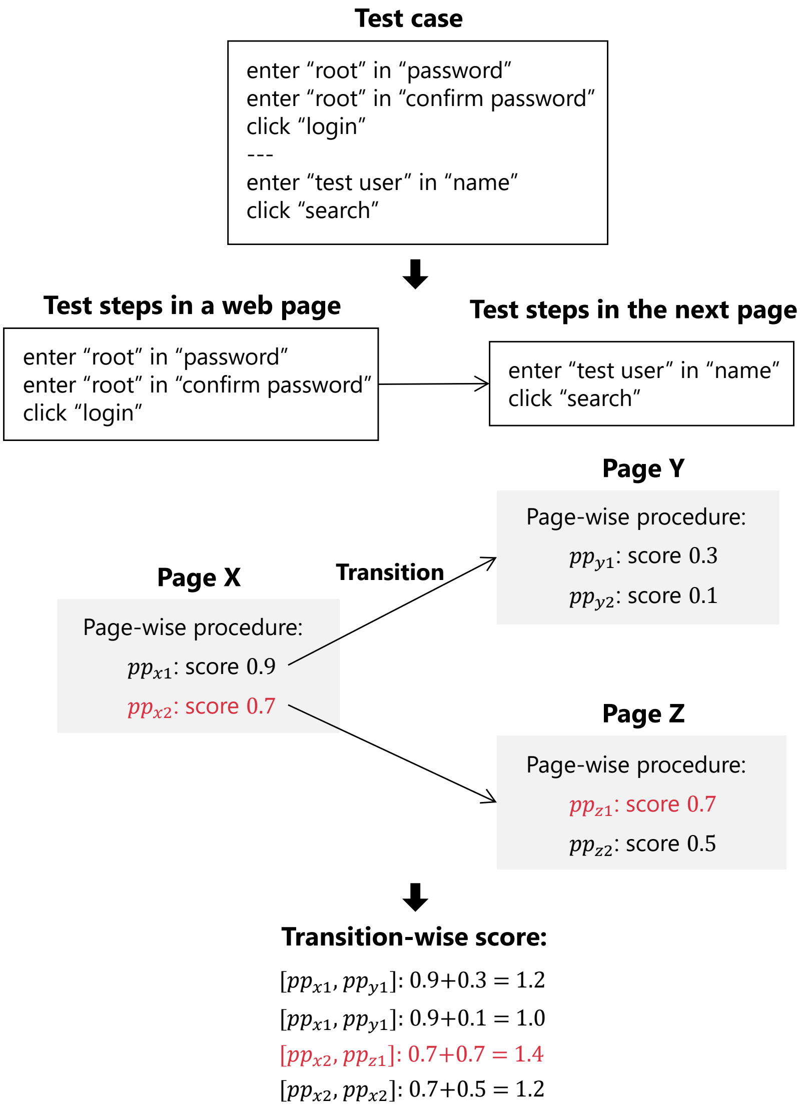

Test case
enter “root” in “password”
enter “root” in “confirm password”
click “login”
---
enter “test user” in “name”
click “search”
Test steps in a web page
enter “root” in “password”
enter “root” in “confirm password”
click “login”
Test steps in the next page
enter “test user” in “name”
click “search”
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

Figure 4.6: An example of transition-level search

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 incorrect even though it has the highest page-wise score on page X. This is because both pp_{y1} and pp_{y2} have low page-wise scores, which means that page Y is not likely to have the web elements specified by the target strings “name” and “search”.

On the other hand, page Z seems to have the web elements specified by the target strings because of its high page-wise score. Therefore, even though it does not have the highest page-wise score on page X, it is more promising to execute pp_{x2} on page X than pp_{x1}.

Table 4.2: 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

Table 4.3: 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.)

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. Because there are many possible page-wise procedures and page transitions, it would take too much time to explore all the possible sequences within the page-wise procedures. Therefore, we use 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, the beam search considers 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 procedure. It is important to note that the transition-level search is performed while dynamically exploring the application that is being tested. Because the state changes of the application during the exploration can 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 considered to be the most promising for the test case.

4.4 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 have rich features, 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?

Table 4.4: 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%)

4.4.1 Experimental setup

There are a large number of features in Joomla! and MantisBT, so we did not prepare test cases that cover all of them. 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, as described in the user manual for administrators [64]: 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 main features in MantisBT: issue management, user management, and others (management of projects, tags, custom fields, and global profiles). We then chose the use cases to cover these features. We note that we excluded some use cases that require operations that our technique does not support.

Table 4.2 shows a summary of the applications and the test cases used in our experiment. We wrote 47 test cases to verify the chosen use cases. However, the way of writing test cases can vary depending on the person. In particular, the accuracy of our technique depends heavily on the target strings used. Therefore, we set the following rules for writing test cases:

1. If the manual describes a specific procedure for the use case, we follow the manual as closely as possible. If a use case has multiple ways it can be achieved, we choose one of them randomly.

2. It is not necessary to fill in all input fields in the test cases. In addition to the required input fields, we fill in one or more optional input fields. When we operate on 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.

3. 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 the target element (e.g., a label right next to an input field)
   - The text displayed in tooltips of the 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 behind these rules is to reduce bias when creating test cases.

We applied the proposed technique with three different sets of 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, and 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, we set $\alpha = 3$ as the weight of the text vector in Eq. (4.1).

To confirm the accuracy of the test procedure determined by the proposed technique, we output the test procedure as a test script written in Python. We can determine the 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 4.3 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. The *type* in the Python code is `id`, `name`, or `xpath`, depending on the locator type, and *locator* is a locator string obtained from the web element. *open* operations 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 are focusing on whether the proposed technique can accurately identify web elements and determine a test procedure in this evaluation.

4.4.2 Results

We manually checked the test scripts to determine the accuracy of the proposed technique in identifying web elements. Table 4.4 shows the number of successful identifications. The *test step* in the table refers to 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 the web pages, even if 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. The *test case* in the table refers to 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

Table 4.5: 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

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. Table 4.4 shows that when $W_s = W_b = 5$, approximately 94% of test steps were successful in identifying web elements. Our approach also succeeded in identifying all web elements in 68% of the test cases. No improvements in accuracy were observed for more search width or beam widths. To answer RQ1, therefore, our technique can correctly identify web elements in up to approximately 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 much lower compared to the other cases. This result indicates 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. Comparing the cases $W_s = W_b = 1$ and 3, we can see that the transition-level search significantly contributes to the accuracy of our technique. Thus, 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. 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 suggests that text words represent the semantics of elements more directly than attribute words. Therefore, the approach weighting text vectors contributes to the accuracy of the proposed technique. To answer RQ2, the vectorization approach and the heuristic search algorithms both contribute to determining correct test procedures.

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

Table 4.5 shows the average execution time (in seconds) of our technique 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, approximately 200 seconds, is not included here. We consider this time to be negligible when the number of test cases to be processed at a time is large because our technique can process multiple test cases simultaneously after the model is loaded. 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. Therefore, we 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"`" was able to correctly identify all three web elements in the situation shown in Figure 4.1. This shows 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 the target strings. The web element in Figure 4.2 did not include the words *display* and *labels* in the HTML document of the web element. However, 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 is effective in capturing the abstract semantics of web elements.

Figure 4.7 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 4.3. 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* represents the total number of test cases. *Plausible* represents the number of test cases in which all web elements were correctly identified. *Executable* represents 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!, could not be operated by Selenium despite the locator being correct. These failures depend on the implementation of the web page, not on locator errors. To operate these web elements in the test executions, it may be 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

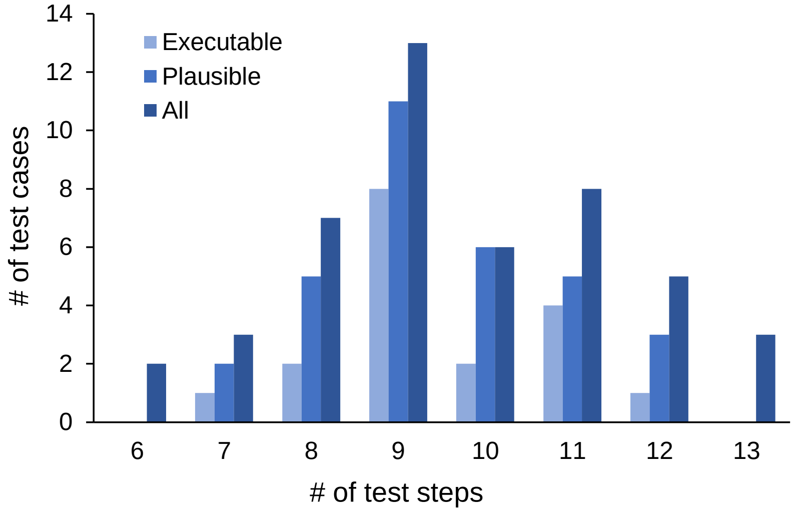


Figure 4.7: The relationship between the number of test steps and that of plausible or executable test scripts

way to deal with this 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.

To answer RQ3, we believe that the execution time is not a practical issue. By analyzing individual cases, we demonstrated the potential of reusing the same test step for various test cases and applications. Additionally, users without programming knowledge may be able to write test cases since our technique does not require knowledge of 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.

4.5 Discussion

4.5.1 What are the cases where our approach does not work?

In this study, we did not find a relationship 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 likely 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. For example, if identifying web elements fails at an early step of the test case and an unexpected page transition is executed, the identification of subsequent web elements will also fail. However, the result in Table 4.4 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. 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. This is a weakness of our heuristic search algorithms.

We found two patterns of web pages where 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 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 be able to distinguish them by adding the words to the target string. Alternatively, by extending our approach to allow for positional information to 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 well 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 to identify a web element because there are more candidates for the operation. Note that 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. This means that the actual number of web elements on the page is much larger than the number of visible elements. One solution for this problem is to incorporate heuristics into our technique, e.g., elements operated consecutively tend to be close to each other in terms of their position on the screen.

4.5.2 Limitations

Our approach has limitations in the target applications and possible operations. Since the proposed technique uses Selenium internally, it can only operate web elements that Selenium can identify. For example, contents created using the *canvas* feature or Flash cannot be operated. Currently, 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.

Furthermore, 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, users need to know when web elements will appear on the web page in order to write appropriate test cases for such applications. 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.

4.5.3 Threats to validity

The following presents two factors that undermine the external validity of our study. The first is the scale of the experiment. We 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. More accurate results could be obtained 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 accuracy of the NLP-based approach depends heavily on the way target strings are written. In this study, we attempted to set rules for writing test cases. These 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 some bias was inevitable, but we tried to reduce it by following the rules. In addition, the created test cases are publicly available, so it is possible to verify the validity of these test cases.

4.6 Conclusion and Future Work

In this study, we proposed an approach to identify web elements from test cases written in a form similar 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 increase the expressiveness of test case descriptions for practical use. Additionally, we want to evaluate our approach on a larger number of applications and test cases in order to demonstrate its effectiveness more generally.

Chapter 5

Conclusion

5.1 Summary

In this dissertation, we conducted three studies to improve the efficiency of implementing and maintaining end-to-end test scripts for Web applications.

The first study proposes COLOR, an approach for repairing broken locators in response to software updates. It uses clues from web pages to evaluate their reliability, and our experimental results show that it has high accuracy and is robust against page layout changes. This study contributes to reducing the cost of test script maintenance.

The second study proposes an approach for generating modularized test scripts to improve their maintainability. It extracts useful operations from test logs and generates test cases that cover the features of an application by analyzing page transitions. The approach was evaluated using test logs from four testers, showing that it can generate more complete methods than an existing approach. Our empirical evaluation also showed that it can reduce the time required to implement test scripts by 48% compared to manual implementation. This study contributes to reducing implementation and maintenance efforts.

The third study proposes a technique for identifying web elements to be operated on a web page by interpreting natural-language-like test cases. The test cases are written in a domain-specific language that is independent of the metadata of web elements and the structural information of web pages. Natural language processing techniques are used to understand the semantics of web elements, and heuristic search algorithms are used to explore web pages and find promising test procedures. The technique was applied to test cases for two open-source web applications, with the results showing that it was able to successfully identify 94% of web elements to be operated and all the web elements in 68% of the test cases. This study contributes to the easy implementation and maintenance of test scripts for various users.

5.2 Future Work

These studies have provided valuable results and knowledge that will serve as the foundation for future work and help identify potential issues and areas for improvement.

The technique proposed in Chapter 4 involved converting natural language-like test cases into locator-based test scripts. Ideally, however, the process should be carried out entirely in natural language, without the need for test scripts. In order to achieve script-free testing, the following two techniques must be developed.

The first technique necessary for end-to-end testing using natural language is the ability to interpret natural language and execute it directly as a test, without the need to convert it into a test script. While it is currently difficult to achieve end-to-end testing using natural language due to the limitations of natural language processing and machine learning technologies, we believe it is feasible if test cases can be interpreted based on an understanding of the structure of the web page and the domain of the web application.

The second technique is the ability to record operations and output them as test cases written in natural languages, similar to conventional record & replay techniques. Even if it is possible to interpret natural language and execute tests, writing test cases can still be labor-intensive. However, this problem can be solved if test cases can be automatically generated by interpreting the content of the operations.

Acknowledgements

First and foremost, I have to thank my research supervisor, Professor Shinji Kusumoto, for his assistance and dedicated involvement in every step of my research activity. I am grateful to him for attending our research discussions and providing valuable insights.

I would also like to express my sincere gratitude to Professor Yoshiki Higo for his guidance in my research while I was in graduate school and for his willingness to accept me into the doctoral program. Even though I was conducting my research remotely, he always took care of me and was always there to support me.

I would also like to express my sincere gratitude to Assistant Professor Shinsuke Matsumoto for his precise advice on my research and papers. He provided insightful comments during our discussions, and his detailed feedback on the content of my papers was invaluable.

I would like to thank Professor Kozo Okano at Shinshu University, Professor Hiroshi Igaki at Osaka Institute of Technology, and Keisuke Hotta at Fujitsu Ltd. for their helpful advice while I was a graduate student at Osaka University.

I would also like to thank the office workers in our laboratory, Tomoko Kamiya, Kaori Fujino, and Misako Hashimoto for their encouragement and support in various situations.

I would like to express my sincere gratitude to Yui Sasaki from The Japan Research Institute, Ltd., who was also a working professional pursuing a doctoral degree like myself. She provided valuable guidance on a variety of topics, including research scheduling and paper writing.

I would like to express my deepest gratitude to Hoshino Takashi, Katsuyuki Natsukawa, Kazuo Morimura, Tatsuya Muramoto, Haruto Tanno, and my colleagues at NTT Software Innovation Center for encouraging and supporting me to enter the doctoral program.

Finally, I would like to thank my family and all of the staff in the Department of Computer Science at Osaka University for their constant support and encouragement.

References

[1] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. In *Proceedings of the 11th International Conference on Product-Focused Software Process Improvement*, PROFES'10, pages 3–16. Springer-Verlag, 2010.

[2] Selenium. http://www.seleniumhq.org/.

[3] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *20th Working Conference on Reverse Engineering*, pages 272–281, October 2013.

[4] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering*, pages 272–281, 2013.

[5] 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*, pages 183–192. ACM, 2009.

[6] 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*, pages 271–282, 2019.

[7] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and maintenance of automated functional tests for web applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 141–150, 2014.

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

[9] M. Iyama, H. Kirinuki, H. Tanno, and T. Kurabayashi. Automatically Generating Test Scripts for GUI Testing. In *IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 146–150, April 2018.

[10] 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*, pages 67–78, September 2014.

[11] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller. WebMate: Web Application Test Generation in the Real World. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 413–418, March 2014.

[12] A Memon. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *In Proceedings of The 10th Working Conference on Reverse Engineering*, pages 260–269, 2003.

[13] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261, September 2012.

[14] 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*, pages 162–171, 2013.

[15] 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*, pages 142–153, 2019.

[16] 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*, pages 67–78, 2014.

[17] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. *Automatic Web Testing Using Curiosity-Driven Reinforcement Learning*, page 423435. 2021.

[18] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust xpath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 449–454, 2014.

[19] 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*, 28(3):177–204, 2016.

[20] 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*, pages 304–314. ACM, 2014.

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

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

[23] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29. ACM, 2011.

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

[25] M. Monperrus. Automatic software repair: A bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, 2018.

[26] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering*, pages 772–781, 2013.

[28] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[29] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[30] R. Yandrapally, G. Sridhara, and S. Sinha. Automated Modularization of GUI Test Cases. In *Proceedings of the 37th International Conference on Software Engineering*, pages 44–54, 2015.

[31] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering*, pages 881–891. IEEE Press, 2012.

[32] Anurag Dwarakanath, Dipin Era, Aditya Priyadarshi, Neville Dubash, and Sanjay Podder. Accelerating test automation through a domain specific language. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 460–467, 2017.

[33] 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*, pages 138–148, 2017.

[34] 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*, pages 4970–4976, 2018.

[35] Mohammad Bajammal and Ali Mesbah. Semantic Web Accessibility Testing via Hierarchical Visual Analysis. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 1610–1621, 2021.

[36] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[37] H. K. N. Leung and L. White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance*, pages 60–69, 1989.

[38] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li. Atom: Automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation*, pages 161–171, 2017.

[39] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, pages 245–254. IEEE Computer Society, 2010.

[40] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418, 2009.

[41] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Why creating web page objects manually if it can be done automatically? In *Proceedings of the 10th International Workshop on Automation of Software Test*, pages 70–74. IEEE Press, 2015.

[42] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Apogen: automatic page object generator for web testing. *Software Quality Journal*, 25(3):1007–1039, 2017.

[43] R. Yandrapally, G. Sridhara, and S. Sinha. Automated modularization of GUI test cases. In *Proceedings of the 37th International Conference on Software Engineering*, pages 44–54. IEEE Press, 2015.

[44] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. *Visual vs. DOM-Based Web Locators: An Empirical Study*, pages 322–340. Springer International Publishing, 2014.

[45] B. Yang, H. Hu, and L. Jia. A Study of Uncertainty in Software Cost and Its Impact on Optimal Software Release Time. *IEEE Transactions on Software Engineering*, 34(6):813–825, November 2008.

[46] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering*, pages 85–103, May 2007.

[47] A. Orso and G. Rothermel. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, pages 117–132, 2014.

[48] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and Maintenance of Automated Functional Tests for Web Applications. In *IEEE International Conference on Software Maintenance and Evolution*, pages 141–150, September 2014.

[49] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. In *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 108–113, March 2013.

[50] F Ricca and A Stocco. Web test automation: Insights from the grey literature. In *Conference: 47th International Conference on Current Trends in Theory and Practice of Computer Science*, 10 2020.

[51] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Clustering-Aided Page Object Generation for Web Testing. In *Web Engineering*, pages 132–151. Springer, Cham, June 2016.

[52] A. N. Ghazi, K. Petersen, E. Bjarnason, and P. Runeson. Levels of exploration in exploratory testing: From freestyle to fully scripted. *IEEE Access*, 6:26416–26423, 2018.

[53] S. M. A. Shah, U. S. Alvi, C. Gencel, and K. Petersen. *Comparing a Hybrid Testing Process with Scripted and Exploratory Testing: An Experimental Study with Practitioners*, page 187202. 2014.

[54] J. Itkonen, M. V. Mantyla, and C. Lassenius. Defect detection efficiency: Test case based vs. exploratory testing. In *First International Symposium on Empirical Software Engineering and Measurement*, pages 61–70, 2007.

[55] PetClinic. `https://github.com/spring-projects/spring-petclinic`, Accessed on Aug 2, 2021.

[56] WebdriverIO. `https://webdriver.io/`, Accessed on Aug 2, 2021.

[57] Y. Chen, Z. Li, R. Zhao, and J. Guo. Research on page object generation approach for web application testing. In *The 31st International Conference on Software Engineering and Knowledge Engineering*, pages 43–48, 07 2019.

[58] H. Kirinuki, T. Kurabayashi, H. Tanno, and I. Kumagawa. Poster: SONAR Testing Novel Testing Approach Based on Operation Recording and Visualization. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 410–413, October 2020.

[59] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella. Dependency-Aware Web Test Generation. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 175–185, October 2020.

[60] M Biagiola, A Stocco, A Mesbah, F Ricca, and P Tonella. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 154–164. Association for Computing Machinery, August 2019.

[61] D. Roest, A. Mesbah, and A. v Deursen. Regression Testing Ajax Applications: Coping with Dynamism. In *Verification and Validation 2010 Third International Conference on Software Testing*, pages 127–136, April 2010.

[62] R Yandrapally, A Stocco, and A Mesbah. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 186–197. Association for Computing Machinery, June 2020.

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

[64] Joomla! Administrator's Manual. `https://docs.joomla.org/Portal:Administrators`, 2020.