Evolution of Web Test Suite

Wei Chen

ACADÉMIE DE BORDEAUX
UNIVERSITÉ DE **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

Présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Evolution of Web Test Suite

par

Wei CHEN

Soutenance prévue pour le 09 September 2021, devant le jury composé de :

Directeur de thèse
Xavier BLANC, Professeur ..... Université de Bordeaux, France

Rapporteurs
David BROMBERG, Professeur ..... Université de Rennes I, France
Tewfik ZIADI, Professeur ..... LIP6 & Université de Pierre et Marie Curie, France

Examinateurs
Laurent RÉVEILLÈRE, Professeur ..... Université de Bordeaux, France
Pascal DESBARATS, Professeur ..... Université de Bordeaux, France

Abstract

Developers rely more and more on so-called End To End (E2E) tests to test the web applications they develop and to check that they have no bug from an end-user point of view. An E2E test simulates the actions performed by the user with his/her browser, and checks that the web application returns the expected outputs. It considers that a web application is a black box, and only knows what are the user actions and what are their expected outputs. However, once some evolutions are performed on a web application, the user actions may change (move the button to another location, add a new button or delete a button). As a result, the E2E test needs to evolve with the evolution of web applications, such as repair the broken test, add the new test, and delete the obsolete test. But it takes a lot of time to evolve E2E tests, especially for large web applications. As such, we do a systematic mapping study to evaluate the existing literature to find gaps in web test suite. We then present an approach, named WebTestSuiteRepair(WTSR), to help the developers who face broken test scripts. In this thesis, WTSR aims at comparing test suite graphs to repair broken actions, hence helps to efficiently repair the E2E tests for web applications automatically. Those approach has been validated through several case studies. We describe some future work to improve our solution, and some research problems that our approaches can target.

Keywords: *Web Application, Test Script, Test Evolution, E2E Test, Test Suite Evolution*

Résumé

Les développeurs s'appuient de plus en plus sur les tests End To End (E2E) pour tester les applications Web qu'ils développent et pour vérifier qu'ils n'ont pas de bogue du point de vue de l'utilisateur final. Un test E2E simule les actions effectuées par l'utilisateur avec son navigateur et vérifie que l'application Web renvoie les sorties attendues. Il considère qu'une application Web est une boîte noire, et ne sait que quelles sont les actions de l'utilisateur et quelles sont leurs sorties attendues. Toutefois, une fois que certaines évolutions sont effectuées sur une application Web, les actions de l'utilisateur peuvent changer (déplacer le bouton vers un autre emplacement, ajouter un nouveau bouton ou supprimer un bouton). En conséquence, le test E2E doit évoluer avec l'évolution des applications Web, telles que la réparation du test cassé, ajouter le nouveau test, et supprimer le test obsolète. Mais il faut beaucoup de temps pour faire évoluer les tests E2E, en particulier pour les grandes applications web. En tant que tel, nous effectuons une étude cartographique systématique pour évaluer la littérature existante afin de trouver des lacunes dans la suite de tests Web. Nous présentons ensuite une approche, nommée WebTestSuiteRepair (WTSR), pour aider les développeurs confrontés à des scripts de test cassés. Dans cette thèse, WTSR vise à comparer les graphiques de la suite de tests pour réparer les actions cassées, contribuant ainsi à réparer efficacement les tests E2E des applications Web automatiquement. Cette approche a été validée par plusieurs études de cas. Nous décrivons certains travaux futurs pour améliorer notre solution et certains problèmes de recherche que nos approches peuvent cibler.

Mots clés : *Application Web, Script de Test, Test Evolution, Test E2E, Test Suite Evolution*

LaBRI — 351, Cours de la Libération — 33400 Talence — France

Contents

CHAPTER

1 Introduction

This chapter introduces the context, problem, motivations, and contributions of this thesis. We describe the fragile problem of test cases maintenance caused by web evolution. This thesis aims to automatically repair test scripts corresponding to the evolution of web applications. As contributions, we first conduct a systematic mapping study of the web test case to investigate a comprehensive understanding of web test cases. We then present an approach to repair broken test scripts by comparing test suite graphs to find a substitute for damaged action. In this chapter, we also describe the structure of this thesis.

Contents

1.1 Context

Since the creation of the World Wide Web in the early 1990s [Berners-Lee et al., 1992], the use of Web applications in our daily lives has greatly increased. A web application is a system usually composed of a database (back-end) and some web pages (front-end). Users can interact with it through a network using a browser. As we all know, web applications have becomed an essential part of daily life [Tonella et al., 2014]. Nowadays millions of web applications are utilized by millions of users every day. [1] For example, Google is one of the most famous web applications. It processes more than 68671 search queries per second on average, equivalent to more than 6 billion searches per day, and 2.1 trillion searches per year globally. [2] This means that web applications are widely used to provide critical services to our society.

The widespread use of web applications puts strict demands on the quality levels that web application developers need to provide [Myers et al., 2011]. To ensure software quality, web testing has been widely used for various artifacts in the industry as a quality assurance technology [Li et al., 2014]. Software testing [Binder, 1996; Fewster and Graham, 1999; Berner et al., 2005] plays a vital role in the production of high-quality software, especially for web application [Ricca and Tonella, 2001; Tonella et al., 2014]. This is because the specific functionality of web-based software makes the assessment of correctness challenging for developers. Therefore, with the widespread use of web applications, the quality and correctness of web applications are highly important.

Web applications have to evolve for satisfying their users' needs. For example, there are 228 different releases of web app Joomla [3] from July 27, 2011, to our searched day on April 18, 2019, which means 28.5 releases every year. Moodle [4] has 325 releases from Aug 19, 2002 to April 18, 2019, which means 19.1 releases every year. Each new release improves the quality of the services proposed by a web application and/or updates its appearance and style. To ensure a high level of quality, the developers should test the new web application release and check that it has no bug from an end-user point of view. Some developers manually interact with the web application to check whether its output is as expected [Stocco et al., 2016]. However, it is error-prone and time-consuming [Stocco et al., 2016; Ricca et al., 2019] to test web applications manually, especially for regression testing [Agrawal et al., 1993; Chen et al., 1994; Gao et al., 2015b]. To that extent, test automation techniques [Tonella et al., 2014] are chosen to enable end-to-end (E2E) functional testing of web applications, which is called automated E2E test [Yoo and Harman, 2012]. This test becomes more and more important [Ricca et al., 2019] with the evolution of web applications. Because developers want to ensure that their latest changes to the source code will not cause errors in existing functionality [Ahmad et al., 2019].

1. https://www.alexa.com/topsites
2. http://www.internetlivestats.com/one-second/
3. https://github.com/joomla/joomla-cms
4. https://github.com/moodle/moodle

An automated E2E test aims to validate a user scenario. It simulates the actions performed by the user with his/her browser (i.e. go to a URL, click on a button, fulfill a form, etc.), and checks that the web application returns the expected outputs [Wong et al., 1997; Rothermel et al., 2001; Nguyen et al., 2014]. An automated E2E test is also called an automated test script to verify the normal functionality of the web application under test (WAUT) [Ricca et al., 2019]. An E2E test considers that a web application is a black box. It only knows what are the user actions and what are their expected outputs. A test should be reusable [Memon and Soffa, 2003], and it should always produce the same assertion when executing a test case on the same version of the web application under test. Several well-known E2E frameworks exist and help developers to design and run E2E tests (i.e. Selenium [5], Segment [6], Puppeteer [7], Cypress [8], Nightwatch [9], Sikuli [Chang et al., 2010], JAutomate [Alegroth et al., 2013], etc.). Nowadays, test execution tools use the DOM-based technique and vision-based technique to interact with web page elements. DOM-based tools (i.e. Selenium, Puppeteer) target and interact with web elements by using Document Object Model (DOM) objects (located in the hierarchy of HTML pages through ID, XPath, or LinkText). Vision-based tools (i.e. Sikuli, JAutomate) use image recognition technology to identify screenshots that represent the visual appearance of web page elements in the browser.

However, the evolutions that are performed on a web application may change its graphical interface (a button can been added, deleted or just moved to another location), which may break the E2E test [Grechanik et al., 2009; Leotta et al., 2015b]. A broken E2E test is an E2E test that cannot be played because one of its user actions cannot be performed. Such action is called a broken action [Leotta et al., 2016b]. Structural changes and logical changes are the main causes of test breakages [Hammoudi et al., 2016b; Leotta et al., 2016a], and more detailed reasons for the damaged test are presented by Hammoudi in the paper [Hammoudi et al., 2016b].

To tackle the breakage problem of the web test script, WATER [Choudhary et al., 2011] and VISTA [Stocco et al., 2018] try to repair broken scripts directly, and some techniques [Christophe et al., 2014; Leotta et al., 2014b; Yandrapally et al., 2014; Leotta et al., 2015b, 2016b] try to increase the robustness of web test script. WATER [Choudhary et al., 2011] is a technique to automatically suggest repairs for web test scripts. Waterfall [Hammoudi et al., 2016a] improves the algorithm of Water and repairs the breakages that occurred in the intermediate minor release between two major releases of a web application. However, it is still challenging for these DOM-based technologies because the breakages do not always occur at the same location where the test execution breaks [Hammoudi et al., 2016b]. Moreover, in many cases, there is more than one breakage in a web test suite. VISTA [Stocco

5. https://selenium.dev/
6. https://open.segment.com
7. https://pptr.dev/
8. https://www.cypress.io/
9. https://nightwatchjs.org/

et al., 2018] is an automated test repair technique, which uses visual analysis to repair web test breakages automatically. However, in some situations, although the appearance of web UI is the same, the selector in DOM will change during the web evolution process, which is difficult for vision-based technologies.

The article [Leotta et al., 2015b] proposes multi-locator to prevent test script breaking by selecting the best locator among a candidate set of locators. ROBULA [Leotta et al., 2014b] generates robust XPath-based locators to partially prevent reducing the aging of web test cases. Leotta et al. [Leotta et al., 2015b] propose a voting algorithm to select the most robust DOM element locator from multi-locators to increase the robustness of the locators for web test cases. Bajaj et al. [Bajaj et al., 2015] improve this algorithm by generating locators from positive and negative examples of DOM elements for multiple DOM elements. Their preventive methods improve the robustness of test suites, but there are still some breakages that require the use of techniques to repair broken tests.

1.2 Problem Statement

After clarifying the research context, we try to introduce the problems in more detail in this section. More specifically, we discuss the main reasons for breaking web tests, then present motivations and our goals.

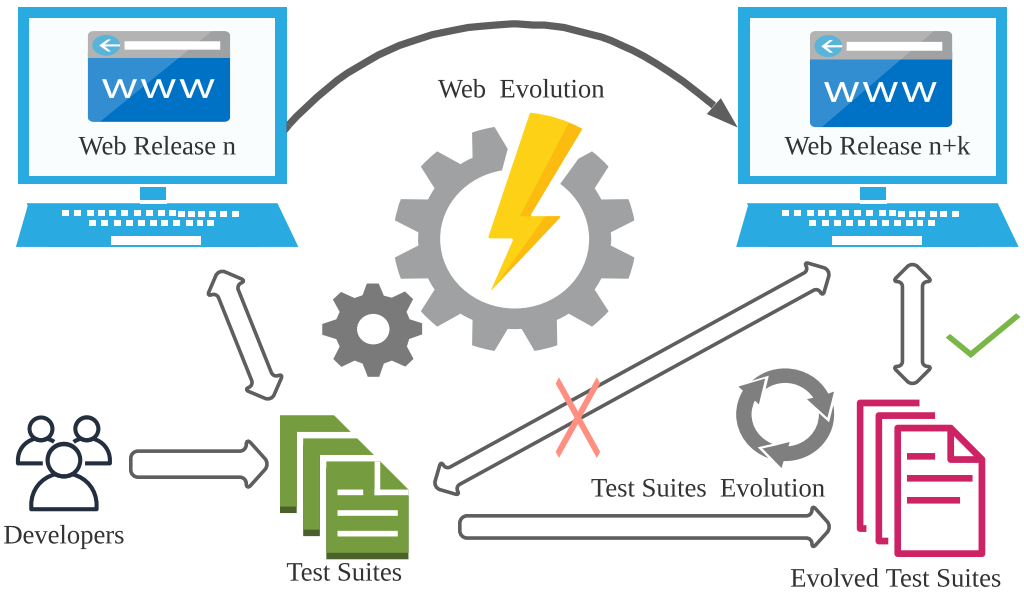


Figure 1.1 – The Problem of Web Evolution

Figure 1.1 indicates the problem caused by web evolution. As it shows, developers create a test suite consists of several test cases. These test cases can work well on web release n. However, a new release n+k is committed which is regarded as the evolution of the web application. Some test cases are interrupted due to changes in the web application (such as add, move, or delete elements of the web application). This inevitable web evolution makes it difficult to repair test cases, so test cases must be constantly evolving to match the new web release n + k.

In [Hammoudi et al., 2016b], Hammoudi et al. show that locators of web elements are the main reason for test breakage. It is expensive to maitain web element locators [Leotta et al., 2014a]. A locator [Bajaj et al., 2015] is a specific command utilized by test automation tools to identify web elements before performing actions on web GUI. The actions of a test case can simulate users, such as clicking on a hyperlink or filling in a text field in a form. For example, when clicking on a hyperlink, the locator is used to identify and locate its web element.

In [Leotta et al., 2014a], Leotta et al. show that even minor modifications to Web Application Under Test (WAUT) will have a massive influence on locators. For example, web page layout changes or other simple change (such as renaming web elements) can disable the locator. Becausse the target element cannot be identified on the web page using this unusable locator. Therefore, compared to maintaining test cases for desktop applications, the specific characteristics of web applications make test cases more fragile, making it extremely difficult and expensive to maintain.

Figure 1.2 shows an example of test breakage, which illustrates how web evolution affects locator. As it depicts, there is a simplified web page for entering user information. The test case for release n of this page has four actions to fill out the form and submit. In this way, developers can test the functionality of inserting information in this web application. When excuting this test, it will automatically enter user information and click the submit button (lines from 1 to 4 in test case n correspond to tags from ① to ④ in order). For example, the first action (line 1 in test case n) locates the *Name* field through locator "#form>tr[1]>td[2]" and enters text content *Bob*. It is similar for actions 2 and 3 (lines 2 and 3 in test case n). Action 4 (line 4 in test case n) targets the *Submit Button* by the locator "#submit-button" and then clicks it. Therefore, the locator in action is very important for identifying web page elements.

As Figure 1.2 shows, the developer then commits a new release of this web application, Web Release n+k (n>0, k>0). The appearance of this web page remains the same, but the attribute of *Submit Button* is changed. Its ID is changed from "submit-button" to "submit". Action 4 will break because the locator "#submit-button" cannot select the *Submit Button* in new release n+k. Hence, the test case needs to be fixed by the developer. For the sake of clarity, we present this simplified example of a broken web test. There are other broken conditions in the actual test, which are not introduced in this section.

The main objective of our thesis is to repair the broken test cases of web applications. So we want to know if there is an abstraction that gathers the test cases for releases n and

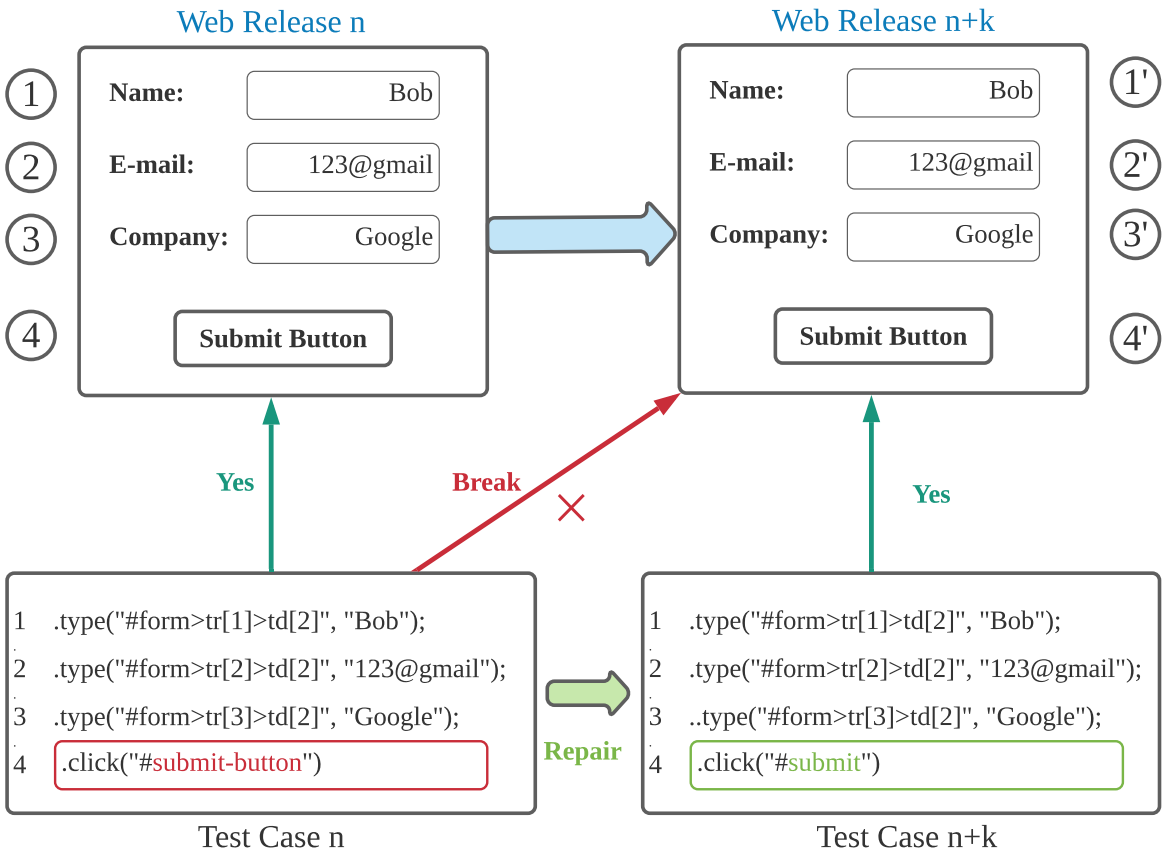


Figure 1.2 – The Example of Test Breakage

n+1, and that can be used to repair the broken test cases. This questioning leads us to the first research question that we investigate in this thesis:

— RQ1: Is there an abstraction that gathers the test cases for releases n and n+1, and that can be used to repair the broken test cases?

To answer this research question, we run the test cases on two releases of a web application. More specifically, it respectively runs original test cases on web release n and web release n+k to generate test suite graphs (abstract models) for these two releases. The test suite model consists of test cases, actions, and connection relation between actions. More details about the test suite abstract model will be presented in Chapter 4.

After generating the test suite abstract model for two releases of a web application, we try to compare them to repair broken tests. So we want to know how to automatically repair the broken web test suite or if it is useful for test repair by comparing test suite models of two web releases. This questioning leads us to the second research question:

— RQ2: Is it possible to automatically and efficiently evolve these test suite for large web applications by comparing their test suite models?

To solve this problem, we propose a novel approach, Web Test Suite Repair (WTSR), to automatically repair broken web test cases. By comparing these two graphs (abstract models), it updates the test suite model of release n+k and repairs the broken tests at the same time. Further, WTSR is developed for test suite evolution, so we want to know if it is effective or not. This questioning leads us to the third research question that we investigate in this thesis:

— RQ3: How effective is this proposed approach for the evolution of test suite?

To answer this question, we choose three real web applications for empirical verification. We use WTSR to create test suite models and to automatically repair broken web test cases. Then, we calculate the number of repaired test cases and their execution time.

1.3 Contributions

We present our contributions to address the issues that we have detailed previously. The ultimate goal of this thesis is to solve a few problems when developers encounter breakage of E2E tests for their web applications. To this extent, we propose the main contributions.

1.3.1 A systematic mapping study of web test case

Throughout the web test evolution cycle, the test suite has several different activities, such as generation, prevention of breakage, and repair. To help the researchers and testers, we systematically identify, summarize the existing literature of the web test suite in this paper. To the best of our knowledge, this paper is the first systematic mapping study in the area of the web test suite. The main contributions of our mapping study are:

— We provide a general classification scheme for categorizing papers in the field of the web test suite.

— We do a systematic mapping study in the field of the web test suite by capturing and analyzing the included papers to structure related research works over the past two decades.

— We present a demographic trend analysis and bibliometrics in the field of the web test suite.

— We identify the gaps in this area for future research.

1.3.2 An approach for DOM-based web test suite repair

To help web application developers who want evolve test suites, we overcome the difficulties and peopose an approach to repair test suite. We present contributions of our web test repair approach:

— We provide an automatic and efficient approach to generate test suite models for two different releases of web applications.

— We propose web test repair approach to automatically and efficiently repair the test suites for web applications by comparing their test suite models.

— We then present how effective is our approach can be used to repair test suites.

1.4 Thesis Outline

The remainder of this document is organized as follows. We first present in Chapter 2 the background in the field of web test cases. In Chapter 3, we do a systematic mapping study on web test cases to identify the gaps in this area for future research. Then, in Chapter 4, we present an approach to automatically identify candidate actions for broken actions to repair test scripts of web applications. Finally, we conclude in Chapter 5 by summarizing the contributions and the main perspectives.

CHAPTER

2 Background

In this chapter, we present the web testing concepts that are needed to understand the remainder of this thesis. We describe some basic definitions for web applications and E2E tests. We provide background information on web testing techniques. We explain how to automatically generate test cases for web applications. It then highlights how an evolution that is performed on a web application may cause an E2E test to break, due to a broken action.

Contents

2.1 Web Application and Evolution

In this section, we introduce an example of a web application and a web evolution. A web application is a client-server software system where the client part (including the user interface and client-side logic) runs in a web browser [1]. As we all know, web applications are related to many aspects of our society and daily lives [Tonella et al., 2014; Garousi et al., 2013]. Through the web application, people can easily study, go shopping, or conduct business.

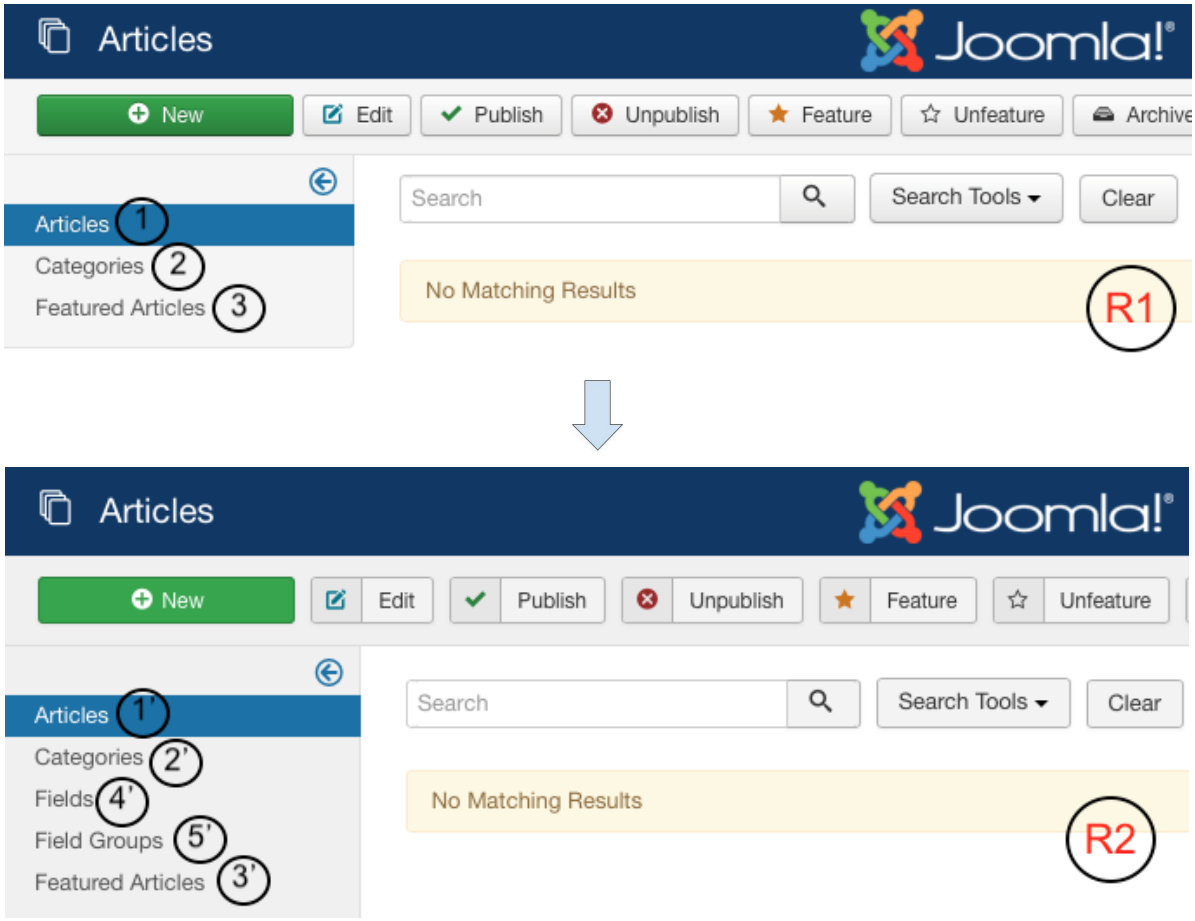


Figure 2.1 – Web application *Joomla*, its evolution from Release 3.6.0 to Release 3.7.0

In the architecture of the web, there are usually two entities, the client and the server. That can be referred to as the client-server architecture of the web application. When we request a web page on the client-side through a browser, the server will respond to the HTTP request of the client. After receiving the HTTP request, the server will analyze the

1. https://en.wikipedia.org/wiki/Web_application

URL, extract the requested document, obtain the data from the database, and provide the client with the requested result. In this thesis, the E2E test focuses on the client-side.

The Figure 2.1 presents a screen-shot of a real web application named Joomla[2] and its evolution. Joomla is a free and open-source content management system (CMS) for publishing web content. At the top of Figure 2.1, it is the client-side page of web release 3.6.0.

The HTML page can be described as a Document Object Model (DOM) with the structure and relationships between different elements on the web page.[3] The DOM of a web page represents a document that contains web elements, regarded as a logical hierarchical tree. Elements in the form of web pages, such as buttons or hyperlinks, can be accessed using a hierarchical object structure of DOM. These elements can be selected as objects by the locator, which can be used for automated testing.

To satisfy its users' needs, developers submit a new release 3.7.0 of web application Joomla. With the improvement in both the quality and quantity of web pages, web applications frequently evolve. Figure 2.1 illustrates a very basic evolution of web applicaiton. Joomla evolves from release 3.6.0 to release 3.7.0. For the sake of clarity, we consider these two releases of the web application as R1 and R2. On the web page of R1, there are three menus, *Articles* (tag ①), *Categories* (tag ②), *Featured Articles* (tag ③). However, on R2's web page, the layout is changed: *Fields* (tag ④') and *Field Groups* (tag ⑤') are added to the menu list.

2.2 Web Testing Techniques

In this section, we discuss some techniques related to End-to-end (E2E) testing. As a kind of classification, web testing can be divided into black-box testing and white box testing. End-to-end testing is a kind of black-box testing [Leotta et al., 2016a]. With the evolution of web applications, regression testing is needed. End-to-end testing is also a type of regression testing. Moreover, the test case can be generated by using Capture-Replay tools (i.e., Puppeteer and Selenium) or using programmable web testing techniques. These technologies will be described in detail as follows.

2.2.1 E2E Web Testing

End-to-end web testing is a technique that tests the entire software product from end view to ensure the application flow behaves as expected. End-to-end web testing is one of the main techniques to ensure the quality of web applications [Ricca and Tonella, 2001]. The goal of end-to-end web testing is to test the web application by detecting bugs or failures from the end of users' views. To achieve end-to-end web testing, developers need to

2. https://www.joomla.org/
3. https://www.w3.org/

write E2E tests. An E2E test is also called a test case or test script. The definition of E2E Test is:

> [. E2E Test] An E2E test is composed of a sequence of user actions that simulates actions performed by an user on its browser (open a page, click a button, fulfill a form, etc.). In addition to that, the E2E test is also composed of assertions that check the expected outputs.

For example, Figure 2.2 is a web page of Joomla to add a new user, and Figure 2.3 presents its E2E test which is composed of 7 user actions (written with the Puppeteer framework). This E2E test is written to test the functionality of adding new users.

Figure 2.2 – The web page of Joomla to add a new user.

When executing this test case, the browser will automatically launch and go to the target web page (line 4). It then runs the actions from line 5 to line 10 in Figure 2.3, which means it enters *Name*, *Login Name*, *Password*, *Confirm Password*, *Email*, and clicks the *Save* button (tags from ① to ⑥ in Figure 2.2. This simple example shows an E2E test that can simulate the behavior of users interacting with web pages. This test case is an ordered list of actions, which describes a scenario that can be completed by a web user through a browser. It treats the web application as a black box and only tests it from the end view.

2.2.2 Black-Box testing

As we discussed before, the functionality of adding new users of Joomla can be tested by treating the web application as a black box. Black-box testing is a software testing pro-

```
1  const puppeteer = require("puppeteer");
2  const browser = puppeteer.launch();
3  const page = await browser.newPage();
4  page.goto("http://localhost:8888/Joomla/administrator/newuser");
5  page.type("#name", "Jack");
6  page.type("#jform-username", "userJack");
7  page.type("#password", "123456");
8  page.type("#confirm-password", "123456");
9  page.type("#email", "12345@gmail.com");
10 page.click("#saveButton");
11 page.evaluate(function() {
12     title = document.getElementsByTagName("h1")[0].innerHTML;
13     title.should.equal("New-User-Complete");
14 });
15 page.close();
16 browser.close();
```

Figure 2.3 – The test script of web application Joomla to add a new user.

cedure designed to meet the functional requirements of the software, treating the software as a black box, and ignoring the internal content [Jan et al., 2016]. It means that the testers do not know the internal design of software while performing black-box testing[Pressman, 2005]. This type of testing is a technique in which testers do not need to access the source code. It focuses on system functionality according to requirements specification, so testers only check whether the system performs the expected work [Nidhra and Dondeti, 2012]. Black-box testing of web applications is usually performed during the execution of test cases [Halfond and Orso, 2007; Chapman and Evans, 2011]. This technique is used to ensure that all inputs required by the system are accepted in a specified way and provide the correct output [Pressman, 2005]. In addition to the web domain, black-box testing can also be used for other GUI tests [Memon et al., 2003].

Although generating and maintaining test cases is a challenge, black-box software testing techniques have the following advantages [Jan et al., 2016]. The test is entirely based on the user's point of view, so testers do not need to access the source code and do not need to have knowledge of a specific programming language [Nidhra and Dondeti, 2012]. Therefore, testers and programmers are independent of each other. It is an effective and very suitable technique for testing software with large code segments.

2.2.3 Regression testing

According to the definition of ISTQB [4] in 2018, regression testing is "A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software." After modification, test the previously tested components or systems to ensure that no defects are introduced or found in the unchanging areas of the software. When the source code of software changes, regression testing is required to ensure the quality of software because some source code changes will cause errors in the software system.

The technique [Andrews et al., 2010b] propose finite state model for regression testing of web applications. Some researchers propose behavioral regression testing approach by generating test cases for changed parts of source code [Orso and Xie, 2008; Jin et al., 2010]. When changes are made to existing software, regression testing will be performed [Andrews et al., 2010a]. The article [Gao et al., 2015b] presents a new way to regression testing. This new method can fully automatically generate test cases for GUI testing and oracle database testing. The purpose of regression testing is to ensure that newly introduced changes will not interfere with the behavior of the existing unchanged parts of the software [Yoo and Harman, 2012]. Yoo [Yoo and Harman, 2012] provides surveys about regression testing, including test suite minimization, test suite selection and test suite prioritization.

In order to implement regression testing, test cases need to be generated for the web application. Some functional and regression testing tools are available as free or commercial software, such as Selenium [Bruns et al., 2009] and Ranorex [KAKARAPARTHY, 2017]. Capture and replay tools [Leotta et al., 2013] support automatic regression testing by replaying a given test script on a new version of the Web Application Under Test (WAUT).

2.3 Test case generation

In this section, we will introduce the background information of test case generation, which is a very important part of web testing. There are a variety of techniques to generate web test cases, such as capture-replay techniques, crawl-based techniques, and model-based techniques.

Capture-replay: The capture and replay technique [Moreira et al., 2017] is designed to record the tester's interactions with the WAUT, such as mouse clicks and keyboard input, which can be used to generate test cases. When the system needs to be tested, these test cases will be replayed. Implementing test cases using this technique is a simple task [Leotta et al., 2013]. And the main advantage of capture-replay technique is that once the test cases are recorded, this technique can save the time of creating test cases without writing test cases again. The disadvantage of this technique is that manual intervention is required to record test cases.

4. https://glossary.istqb.org/en/search/regression%20testing

Selenium provides extensions to be used in browsers (i.e., Chrome, Firefox), allowing all interactions to be recorded and saved as test cases [Bruns et al., 2009]. It can be replayed later and used as a regression test. This is a semi-automated tool that requires manual interaction when recording testers' events on a web page. Testers can add an assertion to the test script after the recorded actions. When replaying a test case, it can automatically launch the browser and simulate user events, such as mouse clicks and text input.

Crawling-based: Crawljax [Mesbah et al., 2008] is a popular open-source web crawler that can process AJAX-based applications by dynamically analyzing changes of user interface state in web browsers. It is specifically developed to meet the needs of crawling websites that use AJAX elements. Some researchers build navigation models for web applications by using the crawling technique [Mesbah and Van Deursen, 2009]. Specifically, they use Crawljax to generate a state flow graph, which consists of states and transitions, and is used to model the web application under test. They then develop a tool called Atusa that uses this inferred model to generate test cases with predefined invariants.

Moreover, the article [Mesbah and Prasad, 2011] provides a flexible plug-in architecture, which makes Crawljax very extensible. Some researchers [Mirshokraie and Mesbah, 2012; Silva and Campos, 2013] have used the plug-in architecture and extended Crawljax for different applications. Researcher Tanida [Tanida et al., 2011] proposes an automated approach for system testing of modern dynamic web applications. Tanida [Mesbah et al., 2012] then proposes a guided crawling tool to facilitate a more comprehensive and scalable crawling behavior, which helps testers to create test cases.

Search-based: Search-based web testing can use search techniques or algorithms to generate test cases automatically. Researchers investigated the use of search-based techniques and provide a survey of test generation [McMinn, 2004; Chen et al., 2013]. Researchers [Biagiola et al., 2017]have presented Subweb, a web testing tool for the joint generation of test inputs and feasible navigation paths using search-based and model-based techniques. They show that this method can guide search to generate test cases that are not affected by path infeasibility issues [Biagiola et al., 2017]. These algorithms can effectively guide the generation of test case input and are suitable for system-level testing [Zeller, 2017].

Search-based techniques iteratively sample the input space, selecting the fittest candidate test cases, and evolving the fittest ones using genetic search operators to create new test cases. Since these algorithms can effectively guide the generation of test cases even for large input spaces, they are suited for system-level testing [Zeller, 2017]. Concerning the web domain, an effective fitness function can be defined based on approximate information available in the navigational model specifically the actions' guards. Researchers have shown that this approach can guide the search toward generating test cases unaffected by the path infeasibility problem [Biagiola et al., 2017].

However, this approach needs the manual specification of all guards for each action, a task that is time-consuming and laborious for testers. Indeed, such information depends on the web application business logic and intended behavior, and thus cannot be gen-

erated fully automatically. Additionally, the evaluation of the fitness function is costly, because it requires a large number of candidates to be generated and executed in the browser before converging to an adequate set of tests.

Search-based software testing uses heuristic search techniques to develop algorithms to generate test cases automatically. These algorithms reduce the cost of the testing process while they maximize the acquirement of test goals. There are two approaches with search-based algorithms: genetic algorithm and combinations of different optimization algorithms, and dynamic execution of symbolic inputs. The Genetic algorithm tries to find the best feasible solution that meets all the constraints. Dynamic Symbolic Execution is a technique to manage data structures dynamically. It collects path constraints on input from predicates encountered in branch instructions. A survey of the use of this technique is provided in [Chen et al., 2013].

Model-based: In the field of end-to-end web testing, many researchers have proposed model-based techniques to generate test cases. Researchers propose event flow graphs of web applications [Kung et al., 2000; Memon et al., 2001]. This model represents the event flow of a Web GUI with nodes and edges. As an improvement, Memon merged different models into a scalable event flow model by using a semi-automatic reverse engineering algorithm [Memon, 2007].

Some researchers use both crawl-based and model-based techniques for web testing. They use Crawljax to create a navigation model of a web application, which includes DOM state events and their interactions [Mesbah et al., 2012]. Such a model is an event flow graph that can be utilized to generate test cases. This method uses the shortest path algorithm to extract test cases from the model and is implemented as Atusa [Mesbah et al., 2011]. In this method, Atusa sorts multiple paths by length and selects the shortest path to generate test cases.

In addition to the event flow graph, there are other web application models, such as finite state machines (FSM). Researchers [Andrews et al., 2005; Miao and Yang, 2010] found that by comparing FSM and EFG, in some cases (i.e., Dynamically modified GUI objects), FSM can be better modeled than EFG.

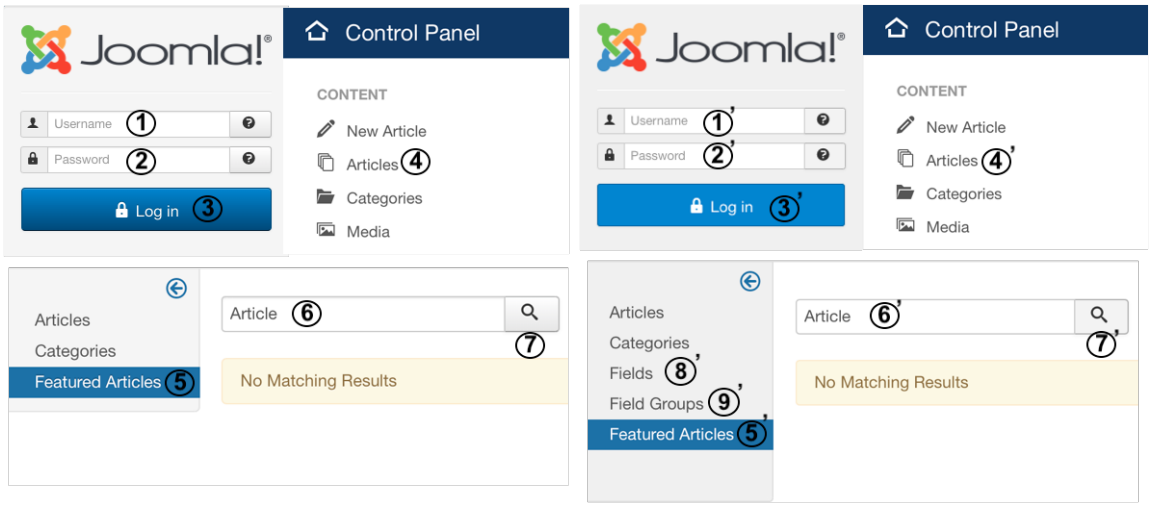
Researchers have proposed a method to improve the effectiveness of building models from GUI by using reverse engineering [Grilo et al., 2010]. And Maciel proposed a model-driven approach to keep test specifications consistent with requirements specifications [Maciel et al., 2019].

2.4 Test Breakage

Software systems undergo several changes during their evolution. For example, Joomla evolves from R1 to R2, illustrated in Figure 2.1. Unfortunately, such changes might affect the corresponding test cases. Even small changes to the DOM or GUI may break previously developed test cases. In the face of software evolution, E2E web testing is notoriously frag-

ile [Hammoudi et al., 2016b; Leotta et al., 2016a]. Then, the tests will need to be repaired to match the updated version of the application. The definition of broken test is following:

> [. Broken E2E Test, Broken Action] A broken E2E test is an E2E test that cannot be executed on a given version of a web application. A broken action is an action of a broken E2E test that cannot be performed.



(a) Web pages of Joomla R1.

(b) Web pages of Joomla R2.

Figure 2.4 – Web application *Joomla*, its evolution of function from R1 to R2

In addition, to make the scope of this study more clear, we should make a distinction between **breakage** and **error**. According to *Definition 2.4*, a breakage occurs that a test case T can be performed in web release R_n but break in the following web release R_{n+i} (i>0), which is caused by web evolution. Differently, an error is that a test case fails due to bugs in web applications. In our research, we focus on the breakages of test cases, not test errors.

In a recent study, some researchers have classified the reasons for the break of web tests, which are available in the literature [Hammoudi et al., 2016b]. They extract the causes of test case breakages and collect them into a single taxonomy. This taxonomy subsumes the taxonomy of causes of test case breakages for web applications [Hammoudi et al., 2016b]. They concluded that locator is the first major cause of breakage (73.62% of all causes of test breakages), and value is the second major cause of breakage (15.21% of all causes of test breakages) [Hammoudi et al., 2016b]. In this study, we focus on locator cause because it merits the greatest attention [Hammoudi et al., 2016b], and we do not target value reason because data of input is related to the backend database. For example, if password value

(see line 2 at the top of Figure 2.5) is the cause of test case breakage, then repairing the test case is a huge challenge.

Further, the locator breakage can be divided into **non-selection** breakage and **mis-selection** breakage [Hammoudi et al., 2016b; Imtiaz et al., 2019]. For non-selection breakage, a locator can target the DOM element in release R_n but is unable to select the target element in release R_{n+i} (i>0), which usually warnings that could not find the locator. For mis-selection breakage, a locator target a DOM element in release R_n but select a wrong element in release R_{n+i} (i>0), which causes the interruption of the following action or assertion.

```
1  puppeteer.page.type("input[name='username']", "admin")
2  puppeteer.page.type("input[name='password']", "123456")
3  puppeteer.page.click("#login")
4  puppeteer.page.click(".Articles")
5  puppeteer.page.click("#content > DIV:nth-child(3)")
6  puppeteer.page.type("input[name='search']", "Article")
7  puppeteer.page.click(".Search")
```

```
1  puppeteer.page.type("input[name='username']", "admin")
2  puppeteer.page.type("input[name='password']", "123456")
3  puppeteer.page.click("#login-button")
4  puppeteer.page.click(".Articles")
5  puppeteer.page.click("#content > DIV:nth-child(5)")
6  puppeteer.page.type("input[name='search']", "Article")
7  puppeteer.page.click(".Search")
```

Figure 2.5 – The test case in two releases of Joomla.

For non-selection example, the web application Joomla evolves from R1 to R2. The locator of button *Login* is changed from *"#login"* to *"#login-button"* (see line 3 at the top of Figure 2.5 and line 3 at the bottom of Figure 2.5). This is because the login button of web application is changed from <button id="login"> Login </button> (HTML for Figure 2.4a) to <button id="login-button"> Login </button> (HTML for Figure 2.4b). Therefore, the test case is broken because it targets a CSS selector *"#login"* that does not exist anymore. This example is a kind of non-selection breakage.

For mis-selection example, the layout of *Sub-Menu Page* is changed that *Fields* (tag ⑧' in Figure 2.4b) and *Fields Groups* (tag ⑨' in Figure 2.4b) are newly added in R2. The locator of *Featured Articles* in R1 (tag ⑤ in Figure 2.4a) is *"#content > DIV:nth-child(3)"* (line 5 at top of Figure 2.5). However, when the test case is performed on R2, this locator selects the *Fields* (tag ⑧' in Figure 2.4b) not *Featured Articles* (tag ⑤' in Figure 2.4b). Because of this mis-selection, the next action to input search content (see line 6 in Figure 2.5) is broken, which is an example of mis-selection breakage.

2.5 Summary

We summarize a background of web testing, including web evolution and test breakages. The state of the art brings a lot of information about the usage of E2E testing in the context of web applications. As a summary, we list the three main lessons that are learned from the existing literature:

— Our first question is about the generation of a model for test repair. We summarize the existing literature that is helpful for us to repair broken tests. Model-based technology is useful for generating test cases, and it inspired us to use it to repair broken tests. Therefore, we try to build test suite graphs as the model to repair tests. In this thesis, we first execute test cases on R1 of the web application to generate R1's test suite graph using crawling technology. We then execute test cases on R2 of the same web application to build R2's test suite graph.

— Our second problem aims to repair the broken tests. So we try to implement a tool named Web Test Suite Repair (WATER) to repair tests. After the generation of test suite graphs for two releases of a web application, we first compare them and repair the broken tests by identifying the candidate actions.

— Our third research question is related to the effectiveness of the test suite repair. To illustrate the effectiveness of our method for repairing test suites, we use three real web applications and their test cases to implement experimental verification research. We first choose three web applications as experiment subjects. And we perform the test cases on them to generate test suite graphs and repair the broken tests. We then calculate how many broken tests can be repaired and how much execution time is required using our approach.

CHAPTER

3 A systematic mapping study of web test case

We do a systematic mapping study to evaluate the existing literature to find gaps in this area. We first describe the systematic mapping study goal and research questions. We present the research methodology on how we do this mapping study. We then provide the classification scheme we have developed for the web test suite and the process used for constructing it. We also present the synthesis results of the extracted data from the selected studies and answers the research questions. We discuss the mapping study results and their implications for researchers and practitioners. In the end, we discuss the threats to validity and present the conclusions in this mapping study.

Contents

3.1 Introduction

Throughout the web testing evolution cycle, the test suite has many different aspects, such as creation, prevention of breakage, repair, dependence, and metrics. To help researchers and testers, we systematically identify and summarize the existing literature of the web test suite in this chapter.

To the best of our knowledge, this chapter conducts the first systematic mapping study in the field of web test suites. The main contributions of this chapter are:

— We provide a general classification scheme for categorizing papers in the field of the web test suite.

— We do a systematic mapping study in the field of the web test suite by capturing and analyzing 76 included papers to structure related research works over the past two decades.

— We present a demographic trend analysis and bibliometrics in the field of the web test suite.

— We identify the gaps in this area for future research.

3.2 Motivation

In this article, we follow the widely accepted guidelines [Petersen et al., 2008; Keele, 2007; Petersen et al., 2015; Kitchenham et al., 2009] to perform this mapping study. We first present the goal and questions. Then we design the methodology to perform this mapping study.

3.2.1 Goal and research questions

The main goal of this systematic mapping study is to analyze primary studies on Web Test Suite and to provide an overview of the web test suite. It aims to investigate a comprehensive understanding of web test suite from the perspective of testers and researchers in the context of web application evolution. That contributes to summarize the body of web test suite in the domain of software development knowledge. Moreover, it also collects direct efforts for the future research of web test suites during web evolution, to identify the existing problems of web test suites, and to figure out the research trend of web test suites. To that extent, we specify the following research questions:

RQ1: What is the current state-of-the-art in the field of web test script? And analysis of related web applications under test. This RQ can be divided into several sub-questions in the following:

— **RQ 1.1 – Type of contribution:** What are the contributions of different studies in the field of web test scripts? And how many studies present techniques, tools, frameworks, models, metrics, guidelines, and processes in the field of web test scripts?

These types of contributions are presented as a guideline in [Petersen et al., 2008]. So we can follow this guideline to extract contribution facet from each study and classify the article in the corresponding class.

— **RQ 1.2 – Type of research facet:** What type of research facets or methods are utilized in the published articles in the field of web test script? The guideline [Petersen et al., 2008] presents several types of research methods, such as solution research, validation research, evaluation research, and experience research. Each article can be categorized as at least one of these research methods.

— **RQ 1.3 – Web test case activity:** What type of web test script activities are presented in each article? We can divide articles into the following categories: test script generation, test script execution, test script break reasons, test script robustness, test script repair, and test script dependency. The activity of the web test script has important implications for researchers and developers. Because researchers and testers can easily select articles that are related to their own research.

— **RQ 1.4 – Techniques used:** What types of techniques are used for web test script in each study? The studies proposed approaches that can divide into the following categories: event-based approach, dom-based approach, page-pattern approach, state-based approach, visual approach, diversity-based approach.

— **RQ 1.5 – Location in web test script:** Which part of the web test script will be studied in each article? A test case consists of input, actions, and assertion. So we categories location in the test case as input, actions, assertion, and test case.

— **RQ 1.6 – Automated level of the techniques or tools:** What is the automatic level of the proposed techniques or tools for web test scripts? According to the level of manual intervention, we can category the techniques as automatic, semi-automatic, manual.

— **RQ 1.7 – Provided tools:** What is the name of the test suite tool proposed and described in each article? How many of them can be downloaded for free? Or how many tools of test script freely available for download?

— **RQ 1.8 – Types of web applications under test:** What types of web apps are chosen by each study to evaluate the approaches for web test scripts? To answer this question, we identify the web applications chosen by each article and summarize the size, type.

RQ2: What is the demographic data of the publications? It has several aspects, such as publication venue and publication year. We can use Zotero to view existing bibliometric studies to obtain demographic data. This RQ can be divided into several sub-questions in the following:

— **RQ 2.1 – Publication year:** What is the publication year of each study?

— **RQ 2.2 – Publication venue:** What is the name of the publication venue of each study?

— **RQ 2.3 – Citations:** What is the number of citations for each study?

3.3 Methodology

In this article, we follow the widely accepted guidelines [Petersen et al., 2015; Keele, 2007; Petersen et al., 2008; Kitchenham et al., 2009] to perform this mapping study. The whole procedure of this mapping study is shown in the following Figure 3.1. In this section, we designed the process of this methodology to search the papers in electronic databases and filter the studies we needed. Then snowballing was presented to supplement the papers in case that some studies were missing during the study search. After getting all the studies, we extracted the data from these selected studies and synthesized the data for this mapping study.

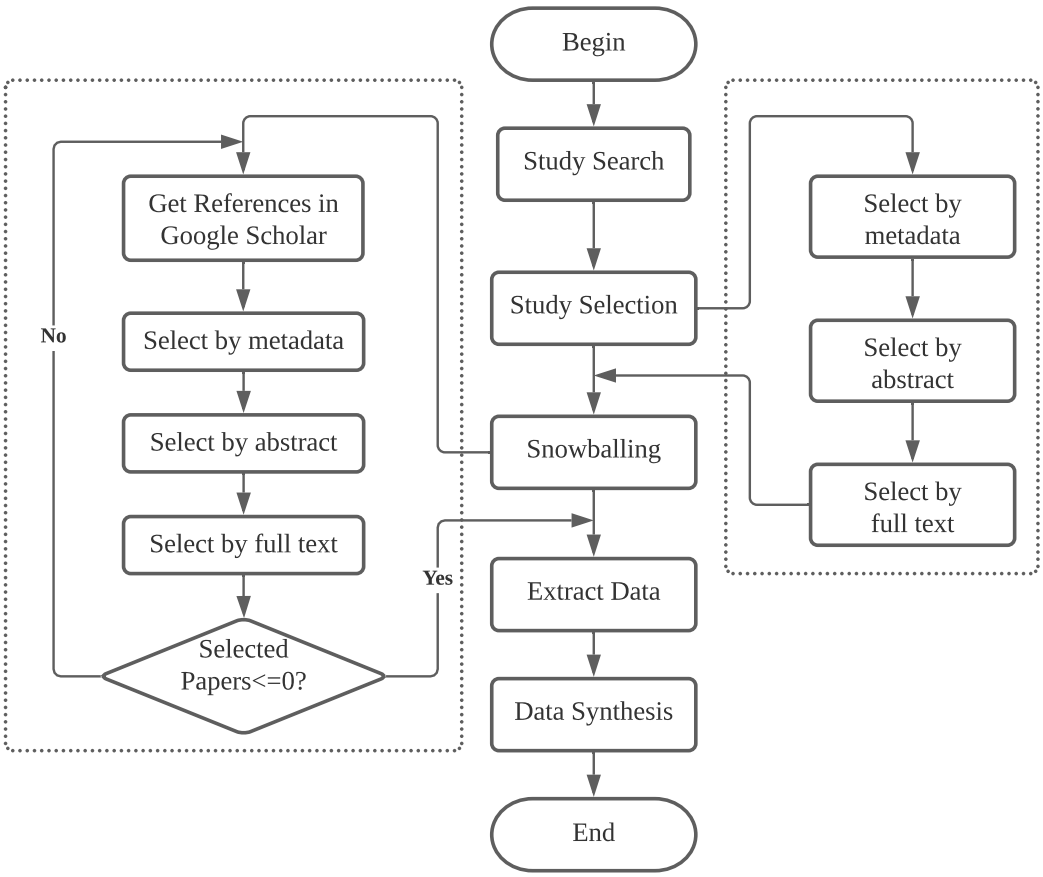


Figure 3.1 – The process of this systematic mapping study.

3.3.1 Study search

The first step of the procedure to perform this study is searching the publications from the electronic databases and retrieve the relevant studies for our mapping study on the web test suite. It is highly significant for a mapping study to define the search period time and electronic databases since it will affect the completeness of initial studies that will potentially associate with the research topic. During the search phase, it is also essential to design the search strategy for the same reason.

Search time period:

We choose 2000 as the start time of the search period. We choose 2019 as the end of the search period in that we started this mapping study in January 2020. In the end, the year covered by search is from 2000 (included) to 2019 (included).

Search electronic databases:

To perform this mapping study, we identify almost all the electronic databases that are related to software engineering, which were suggested in [Petersen et al., 2015; Keele, 2007; Brereton et al., 2007; Chen et al., 2010], listed in the table 3.1.

Table 3.1 – Search electronic databases.

	Database	Selected
DB1	IEEE Xplore	Yes
DB2	ACM Digital Library	Yes
DB3	Science Direct	Yes
DB4	Springer Link	Yes
DB5	Scopus	Yes
DB6	Inspect	Yes
DB7	Google Scholar	Yes
DB8	Wiley InterScience	No
DB9	CiteSeer	No
DB10	ISI Web of Science	No
DB11	EI Compendex	No

If we can, we would select and use all the electronic databases mentioned in table 3.1 to include all the related studies. For the last four databases Wiley InterScience (DB8), CiteSeer (DB9), ISI Web of Science (DB10), and EI Compendex (DB11), we are not accessible in our university. Most studies can be searched in the other selected databases(DB1-DB8). Moreover, we will do snowballing by searching for all the references to find the missing

publications. For these reasons, the last four databases(DB8-DB11) were not selected for this study.

Search keywords:

The topic of our mapping study is about web test suite, and to include all the publications of this topic, we try to search in the databases by applying the search string "test suite". However, the publications of the searching results showing that there are a considerable number of papers not interrelated to the web for the papers just including the "test suite" but in other domains such as desktop apps or mobile apps. Then we want to use "web test" as the keywords. But the scope is too big because it includes the other test techniques which are not related to testing suites. And considering that some papers do not use the term “web test suite” explicitly but use the other phrases, such as "web test script", "web test scenario" or "web test case". Therefore, we choose "Web AND Test AND (Suite OR Script OR Scenario OR Case)"as our searching string in the end.

3.3.2 Study selection

After the process of study search, the potentially relevant primary studies have been obtained, which need to be assessed for their actual relevance [Petersen et al., 2015; Keele, 2007]. To ensure that the results of the study selection are impartial and objective, we designed the study selection criteria and the study selection process.

Study selection criteria

To identify and select the studies that present the evidence of research questions, both inclusion and exclusion criteria are defined based on the questions mentioned above. Criteria are essential to reduce the bias of study selection and correctly classify the primary studies. The criteria were divided into two categories, inclusion criteria and exclusion criteria.

The following inclusion criterias:

C1: The paper should be peer-reviewed, i.e.,published in journals, conference proceeding, workshop proceedings, or book chapters.

C2: The paper should make a contribution to web test suite.

C3: If a paper generates the web test suite in details, which is helpful for developers and researchers to build the web test suite, it will be included.

C4: If a paper talks about the maintenance of web test suite, including the prevention or repairation of breakage of web test suite, it will be included.

C5: If a paper discusses execution of web test suite, including the replay of web test suite for regression test, it will be included.

C6: If a paper talks about the indicators of web test suite quality, including the coverage about web test suite, it will be included.

C7: If a paper makes contributes to improving the quality of a test suite, it will be included.

The following exclusion criteria:

C8: The website or blog which is not a publication will be excluded.

C9: Any paper not published in the English language will be excluded.

C10: Any paper published in the form of an abstract, tutorial, or talk is excluded.

C11: Papers that are not related to the web test suite should be excluded

C12: Books teaching how to use web test suite to do web testing will be excluded.

C13: Any paper just mentions the concept of the web test suite, without contribution to the web test suite, then this paper will be excluded.

C14: If a paper just mentions the design of a web test suite without the details of how to generate it, then this paper will be excluded.

C15: If a paper just uses a web test suite as a part of a program to do a web test, it will be excluded.

C16: If a paper just contributes to the test suite but does not in the web domain, it will be excluded. For example, papers that contribute to web service, web API, android apps, or desktop apps are not included.

Study selection process

Table 3.2 – Study selection process.

Step	Check object	Criterias
1	Metadata	C1/C8/C9/C10
2	Abstract	C2/C3/C4/C5/C6/C7 /C11/C12/C13/14/15/16
3	Full text	C2/C3/C4/C5/C6/C7 /C11/C12/C13/14/15/16

First, one researcher filtered papers based on metadata including title, keywords, and venue name applying the criteria in table 3.2. If the paper is not explicitly excluded, we will keep this paper for the next step of selection.

Second, two researchers independently filtered papers by reading the abstracts of the papers left in the first selection process and applying the criteria in table 3.2. If these two

developers have different inclusion results of the study paper, the other two developers filtered the conflicting papers by reading the abstracts of the papers. If our team did not get the conclusion that whether the paper should be excluded, this paper will be included temporarily and will be selected in the next step together with other include papers.

Last, two researchers independently filtered papers by reading the full text of the papers left in the second-round selection and applying the criteria in table 3.2. If these two researchers own different inclusion opinions of studies, our team will reread the studies and discuss them together to decide whether to include them.

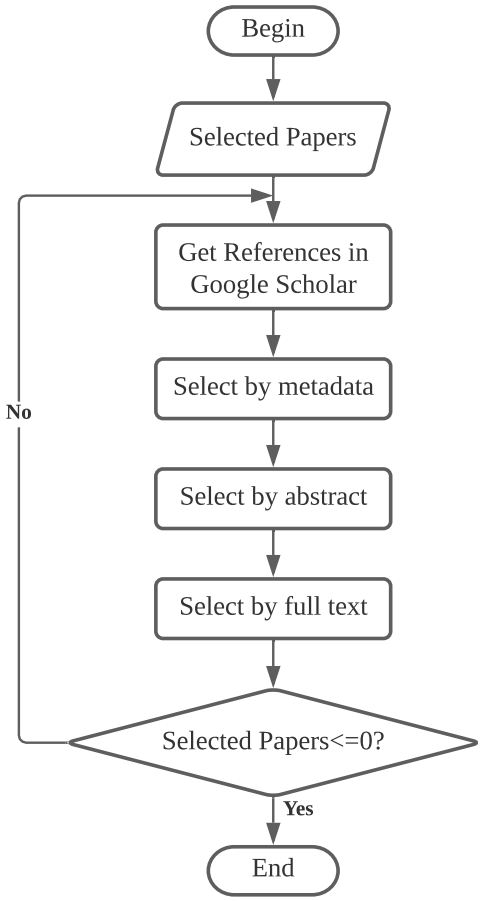


Figure 3.2 – The process of snowballing.

3.3.3 Snowballing

The snowballing procedure is essential for a systematic mapping study, which is presented by researchers to avoid missing relevant studies [Wohlin, 2014; Petersen et al., 2015]. The process of snowballing is shown in Figure 3.2.

After study searching and study selection, we get the relevant studies, which are the foundation data of the snowballing. We utilize these selected studies as input and find their references. We then conduct the study selection process to filter the references by metadata, abstract, and full text. If we get selected studies from the references, we do it again by applying the newly selected studies as input. This iterative process will be stopped until there are no selected papers.

3.3.4 Data Synthesis and Extraction Method

Table 3.3 – Data collection and classification scheme for research questions

RQs	Data categories	Possible outcomes	Multiple
RQ 1.1	Type of contribution	technique, tool, framework, model, metrics, guideline, and process.	√
RQ 1.2	Type of research facet	solution research, validation research, evaluation research, philosophical paper, opinion paper, experience research.	×
RQ 1.3	Web test case activity	test case generation, test case execution, test case break reasons, test case repair, test case robustness, test case dependency, test case minimization, test case migration.	√
RQ 1.4	Techniques presented	model-based, diversity-based, crawling, page object pattern, user-session-based, vision-based, search-based, state-based.	√
RQ 1.5	Location in test case	test case, input, action, assertion, locator.	√
RQ 1.6	Automated level	automated, semi-automated, manual.	×
RQ 1.7	Provided tools	Tool names provided (if any).	√
RQ 1.8	WAUT	Informations of WAUT: Name, Size of WAUT (LOC), Language, Description of WAUT, Quantity.	√

The principle of data synthesis is to simplify the evidence representation in nominated papers to simplify the data extraction process [Keele, 2007; Petersen et al., 2015]. To extract data for this systematic mapping study, a classification scheme needs to be obtained

through careful analysis of basic research [Petersen et al., 2008; Cornelissen et al., 2009]. This will help extract data from the paper to answer the research questions accurately.

Our classification scheme started from the initial version and improved during the data extraction process through attribute generalization and iterative refinement steps. We used Zotero to tag each study according to the scheme when we review them. After that, we summarize the extracted data in Google Docs, which is an online spreadsheet. Newly discovered categories will be added, and existing categories can be merged, split, or removed. If necessary, the classification scheme will be updated. When we add research to a category, we need to provide a short reason why it should fall into this category. Each study will be reviewed by at least two reviewers, and differences of opinion will be discussed in detail until a final decision is made.

Table 3.3 lists our final classification scheme, and the research questions (RQs) are answered by each attribute of the map. Now, we discuss the properties of the classification scheme. The columns of the table show the research question (RQ), the data categories, the possible results, and whether there are multiple results. For the "Multiple" parameter in the fourth column, this means that sometimes the possible outcomes are multiple rather than unique. For example, in Table 3.3, RQ 1.1 shows the contribution type of a paper, but the paper may have multiple contributions. The paper may propose a technique and provide a tool at the same time, which means multiplecontributions. According to the data categories in Table 3.3, we explain the classification schemes in following:

RQ 1.1 - Type of contribution:

In our system mapping research, we adopt the researcher's guidelines and suggestions [Petersen et al., 2015, 2008]. It proposes that a contribution facet (corresponding to RQ 1.1) denotes the type of contribution(s) proposed in each study: method/technique, tool, model, metric, process, or other. In the context of our web test case, these contribution facets would turn to the following: tool, technique, framework, and others.

RQ 1.2 - Type of research facet:

The type of research facet corresponds to RQ 1.2. The guidelines [Petersen et al., 2008, 2015] propose several types of research facet, including Solution Proposal, Validation Research, Evaluation Research, Philosophical Papers, Opinion Papers, and Experience Papers. In our systematic mapping study, we do not find Philosophical Papers, Opinion Papers, and Experience Papers in our selected studies. Therefore, we choose Solution Proposal, Validation Research, and Evaluation Research as the research types for our study.

Solution Proposal: It is a solution to a specific problem that can be a novel solution or a major extension of one existing technique. The potential benefits and applicability of this solution are shown through a good line of argumentation, a small example. Therefore, studies with only examples are classified in the solution proposal.

Validation Research: Techniques investigated are novel and have not yet been implemented in practice. These techniques are validated in a lab through experiments. So studies that have a verification part in a lab, but do not have all the contents of a systematic empirical study in practice, are classified in verification research.

Evaluation Research: Techniques are implemented in practice and an evaluation of the technique is conducted. This means that it shows how the technique is implemented in practice (solution implementation in the industry), and what are the pros and cons of the technique (evaluated as an actual project). If the study uses systematic empirical evaluation (such as a controlled experiment) to evaluate the proposed technique comprehensively, at the same time it also wholly discuss the advantages, disadvantages, and threats to the validity of the results, we then classify this study as an evaluation research.

RQ 1.3 - Web test case activity:

The fourth row in Table 3.3 is the activity of web test cases presented in each study (corresponding to RQ 1.3). In the mapping study, we divide the activities of test cases into the following 12 types: test case generation, test case dependence, test case repair, test case flakiness, test case metrics, test case prioritization, test minimization, test case isolation, test case robustness, test case evolution, test case regeneration, test case comparison, test case execution. We formed this specific classification by reviewing papers and used it for our classification scheme, which is very suitable for studies in our pool.

RQ 1.4 - Techniques used:

Researchers presented many techniques for web test cases, including model-based techniques, page object pattern, user-session-based techniques, crawling techniques, etc. This attribute is related to RQ 1.4, and these types of techniques are used to generate, maintain, or repair test cases. We review each article to generate the distribution of these used techniques, which will be analyzed as the mapping results in Section 3.4.

RQ 1.5 - Location in test case:

The location of a web test case can be input, action, locator, assertion, or the entire test case. Some articles present approaches to create the test cases of web applications, and some studies focus on generating the input data of a test case. Some papers focus on the locator of the action in the test case to improve its robustness. Also, some researchers target the assertion of the test case.

RQ 1.6 - Automated level :

The automation levels of the approaches in the articles are divided into three categories: manual, semi-automated, or automated. We review each study and classify it into one of three categories based on the human intervention of the approach in each study.

— Manual: The techniques are fully assisted by testers and always require manual intervention.

— Semi-automated: The proposed approach can automatically complete part of the test jobs, but some processes of this approach still require manual intervention.

— Automated: The proposed technique can automatically investigate web test cases without any manual intervention.

RQ 1.7 - Provided tools :

When researchers propose a technique or algorithm, they usually implement it as a tool, which is very significant for turning academic research into practical web applications in the industry. These tools are vital in the industry because they can be used directly for web testing to save testers time. Therefore, we summarize the existing tools proposed in each study.

RQ 1.8 - WAUT :

As shown in Table 3.3, we try to extract the following attributes for web applications under test (WAUT) used during empirical validations or evaluations in each paper.

— Number of WAUT in each paper

— Name of WAUT

— Lines of Code (LOC): the size of WAUT

— Development language of WAUT

— Descripbtion of WAUT

— Number of used times for each web application

3.4 Systematic Mapping Results

In this section, we will present the results of the system mapping study and use the extracted data to answer each of our research questions.

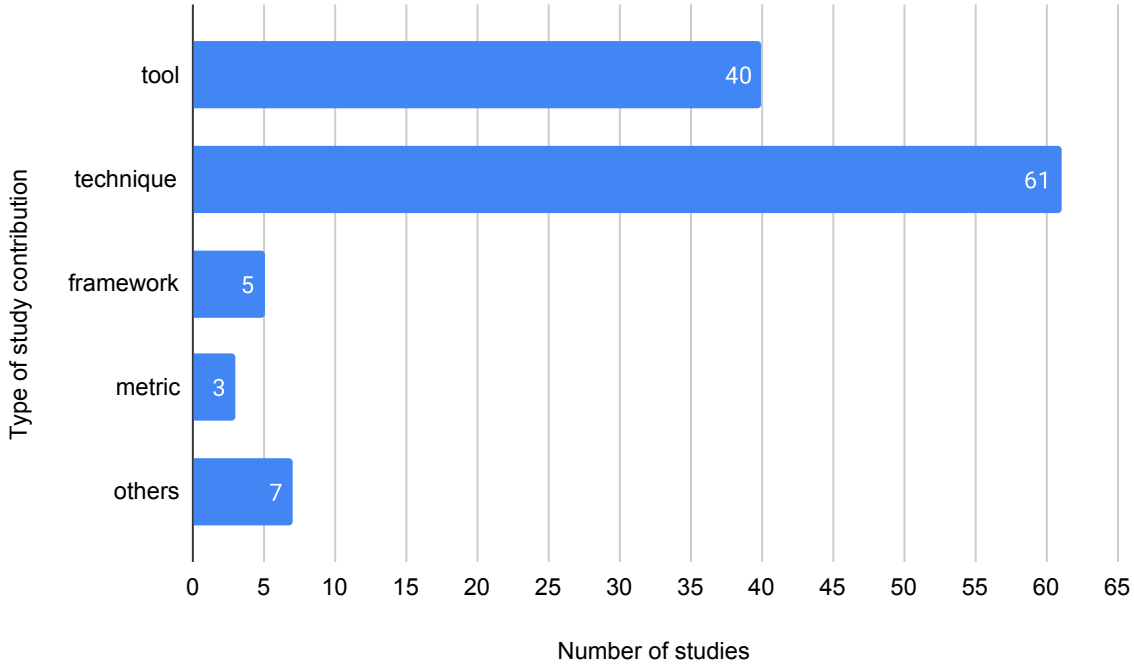


Figure 3.3 – Type of contribution.

3.4.1 RQ 1.1 - Type of contribution

Figure 3.3 shows the distribution of paper contribution types for all 76 papers included in our research. It shows that there are three main types of study contributions, technique, tool, framework, and others. As Figure 3.3 illustrates, most researchers have contributed to presenting techniques, not only proposing new techniques but also improving existing techniques. It attracted approximately 80.26% (61 out of 76 studies) to focus on techniques. Besides, a relatively high percentage of studies (about 52.63%, 40 out of 76) implement tools for web test cases. The other five studies provide frameworks for web test cases, and the proportion is about 6.58% (5 out of 76). Three studies (about 3.95%, 3 out of 76)focused on the metrics or indicators for Web test case evaluation. Because there are 7 papers that cannot be classified into these four contribution types, so we classify them as "other".

In terms of contribution aspects, some studies have more than one type of contribution. For example, S10 [Stocco et al., 2017] makes two types of contributions. First, it presents a new techniques for the automatic generation of page objects for web applications. Second, it implements a tool called APOGEN that can automatically derive test models by reverse-engineering the target web application, which is useful for generating test cases. For another example, S4 [Artzi et al., 2011] makes two contributions, too. It

implements a tool called Artemis and also provided a framework for feedback-oriented testing of JavaScript applications.

3.4.2 RQ 1.2 - Type of research facet

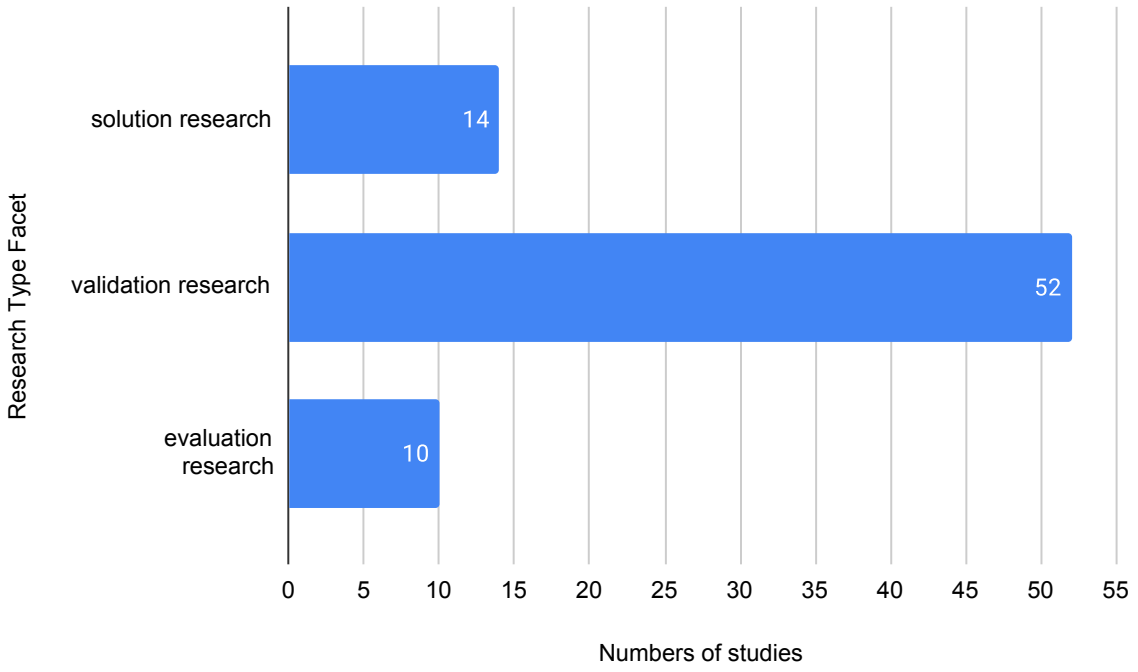


Figure 3.4 – Type of research facet.

Figure 3.4 illustrates paper types of research facet, which corresponds to RQ 1.2. The research of web test cases is mainly based on verification research. There are 52 pieces of such study, accounting for 68.42%. This means that many researchers not only propose a new technique but also conduct verification experiments on the applicability and effectiveness of the proposed technique. For example, S27 [Biagiola et al., 2019b] provides a diversity-based web test generation approach, which is implemented in a tool called DIG and evaluated by generating test cases for six different web applications. Solution research account 18.42% (14 out of 76). These studies propose solutions that are helpful for web test cases, only give a simple example to support their arguments without validation experiments. There are 10 studies that are categorized in evaluation research, which is a reasonable share in comprehensive experimental research (accounting for 13.16%).

3.4.3 RQ 1.3 - Web test case activity

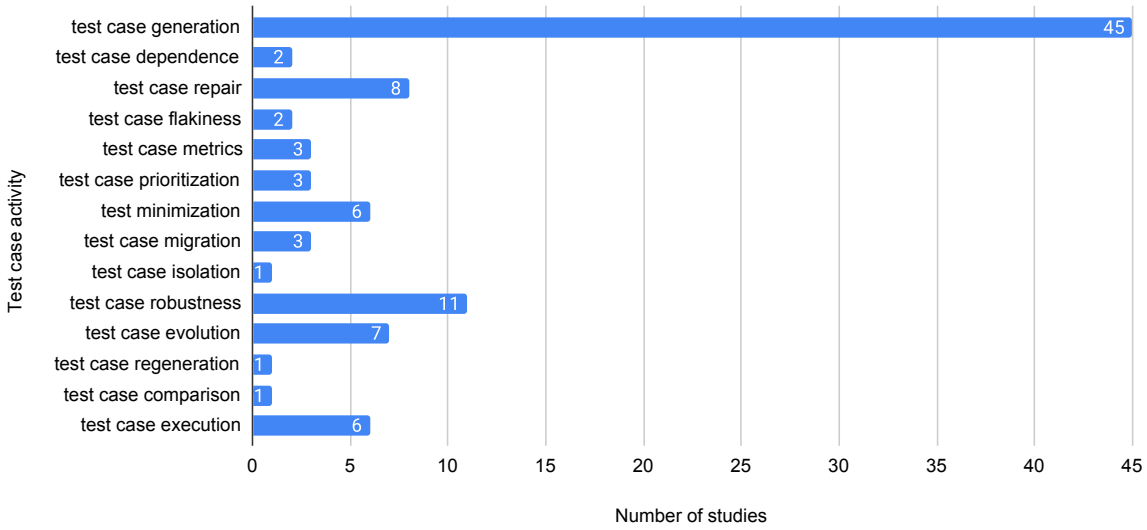


Figure 3.5 – Web test case activity.

Figure 3.5 depicts the distribution of web test case activity in the pool of our mapping study. Of the total 76 studies included in our mapping research pool, 45 articles (about 59.21%) generate test cases by proposing novel techniques or improving existing approaches. For example, S29 [Wassermann et al., 2008] proposes an automatic input test generation algorithm for web applications by dynamically using information from previous executions. S21 [Stocco et al., 2016] presents an approach to automatically create page objects, which is a model for generating test cases of web applications. The study S16 [Hanna et al., 2018] proposes a test automation framework named SAT, which can be successfully used to automate the creation of web test scripts.

As shown in Figure 3.5, 11 articles (approximately 14.47%) propose algorithms or techniques to improve the robustness of web test cases. These algorithms or techniques can prevent the breakage of test cases to a certain extent. S56 proposes a tool named ROBULA [Leotta et al., 2014b], which generates robust XPath-based locators to prevent reducing the aging of web test cases. S68 [Leotta et al., 2015b] proposes a voting algorithm to select the most robust DOM element locator from multi-locators to increase the robustness of the locators for web test cases.

Another popular research activity is test case repair, with a ratio of 10.53% (8 out of 76). The activity of test case evolution has attracted similar attention, accounting for 9.21% (7 out of 76). For example, Water (S72) [Choudhary et al., 2011] and Waterfall (S73) [Hammoudi et al., 2016a] have proposed DOM-based techniques to repair broken tests of web applications automatically. Besides, S70 proposes a tool named Vista [Stocco et al., 2018].

It uses a fast image-processing pipeline to analyze relevant visual pictures obtaining from test execution and suggest potential fixes of test breakages to testers.

In addition, test minimization and test case execution have also been described as test activities. Both of them have six studies, with the same ratio of 7.89% (6 out of 76). In Figure 3.5, there are also some papers that fall in other types of test activities, such as test case dependence, test case flakiness, test case metrics, test case migration, test case isolation, test case regeneration, and test case comparison.

3.4.4 RQ 1.4 - Techniques used

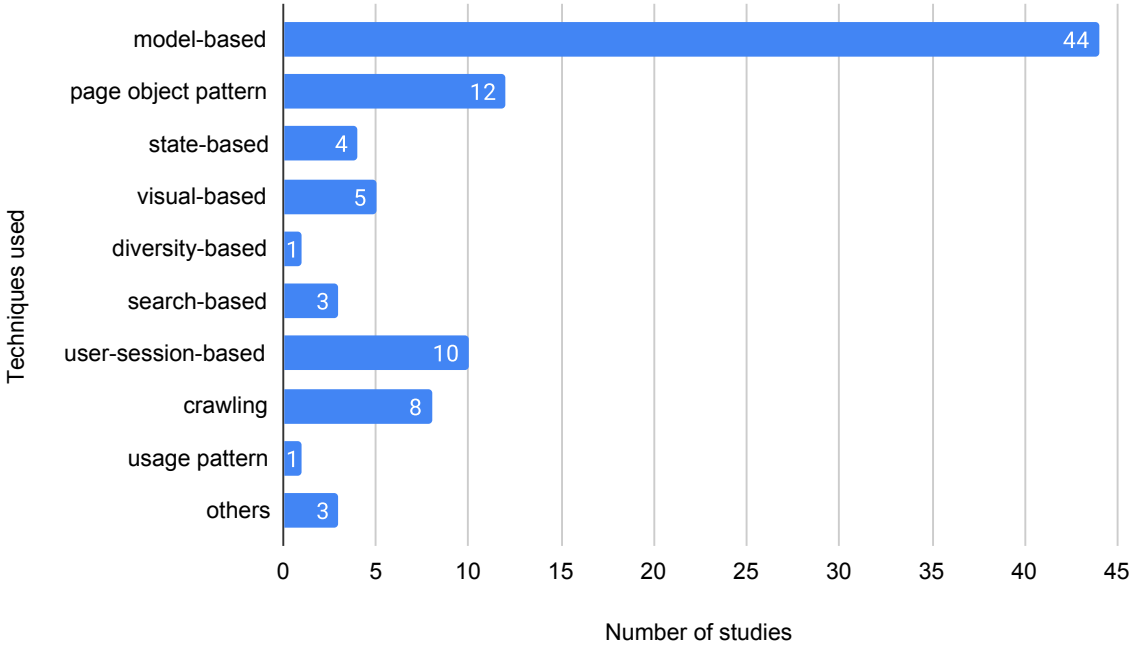


Figure 3.6 – Techniques used.

Figure 3.6 demonstrates the distribution of techniques used in the studies for web test cases. Model-based techniques are the most popular methods for researchers to deal with web test cases, accounting for 57.89% (44 out of 76). For example, S9 uses artificially constructed models to analyze and generate test cases of web applications. It uses model-based techniques provided by users to derive test cases. And 12 (about 15.79%) articles propose or utilize page object pattern to generate web test cases. S10 [Stocco et al., 2017] uses page specifications to automatically generate page objects, which is useful for web test generation through HTTP communication with the server. The third popular one is

user-session-based technique, which is account for 13.16% (10 out of 76). Crawling is also popular and used in 8 papers (10.53%).

On the other hand, vision-based and state-based techniques have also attracted the attention of researchers dedicated to web test cases. Vision-based techniques account for 6.58% (5 out of 76), and state-based approaches have four studies accounting for 5.26%. Moreover, the category of "other" techniques in this scheme included diversity-based technique to generate web test cases of S27 [Biagiola et al., 2019b]. For another example, S3 [Azizi and Do, 2018] proposed usage pattern approach for test case prioritization in web applications. S74 [Biagiola et al., 2019a] uses the NLP technique to present a test dependency for web test cases.

As we know from Figure 3.6, some articles use more than one technique to investigate web test cases. We can classify one paper as multiple types of techniques (if any). For example, S72 [Choudhary et al., 2011] uses two techniques (crawling, model-based) to repair test cases for web applications. S10 [Stocco et al., 2017] uses three techniques (crawling, page object pattern, and model-based) to create a test model of a web application, which can be utilized to generate a web test script in the context of a case study.

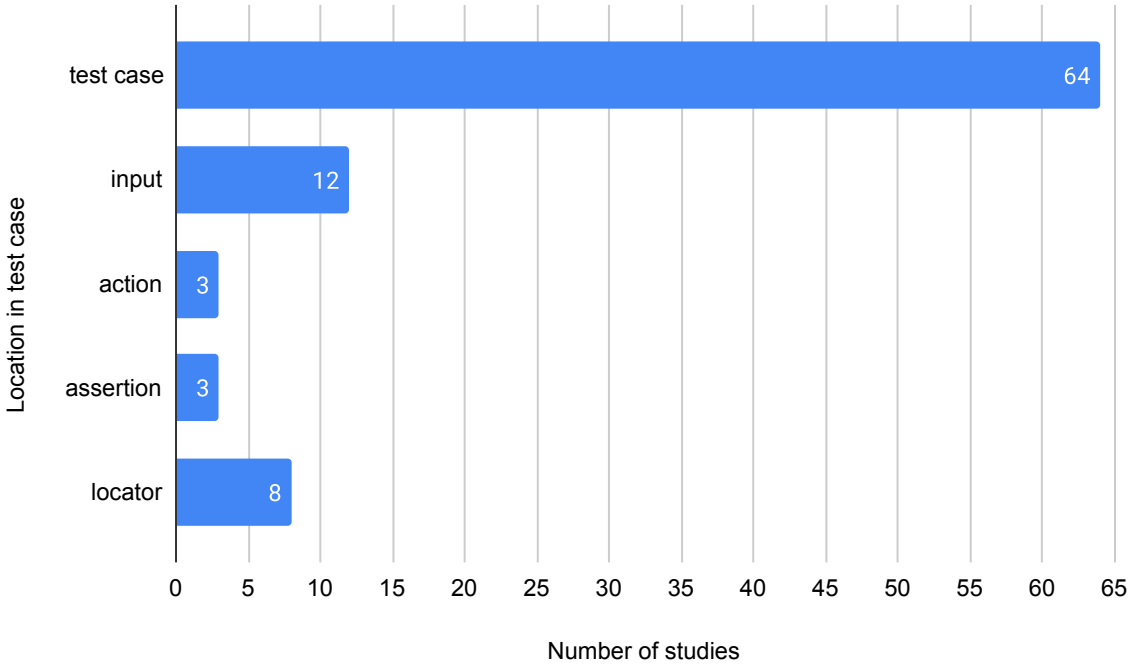


Figure 3.7 – Location in test case.

3.4.5 RQ 1.5 - Location in test case

The distribution of location in test case is shown in Figure 3.7. As it depicts, the majority of studies focus on the entire test case, accounting for 84.21% (64 out of 76 articles). For example, S74 [Biagiola et al., 2019a] proposed a test dependency technique for E2E web test cases based on string analysis and NLP implemented in a tool called TEDD. There are 12 studies that focus on the input of a test case, accounting for 15.79%. For example, S60 [Biagiola et al., 2017] generates input data for test cases of web applications. Some researchers (about eight articles) focused on the locators of test cases, which accounts for 10.53%. Take the locators as an example, S68 [Leotta et al., 2015b] use multi-locators to increase the robustness of web test cases. Some articles focus on the other two categories, namely actions and assertions. Each category has three articles, respectively accounting for 13%.

3.4.6 RQ 1.6 - Automated level

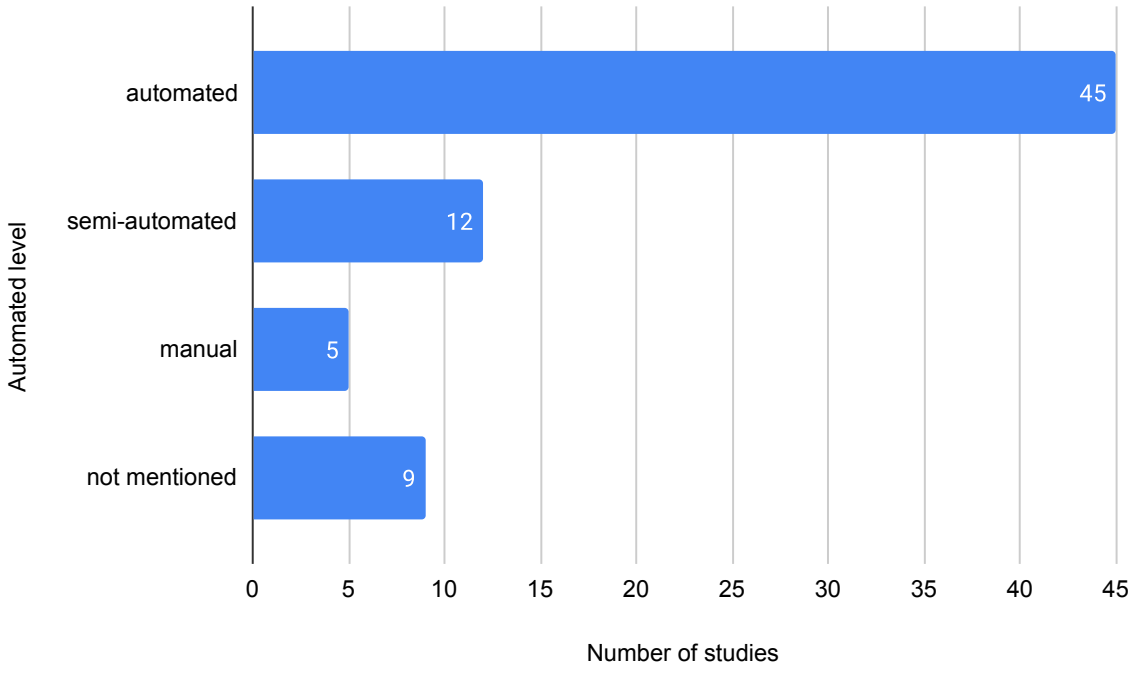


Figure 3.8 – Automated level.

As Figure 3.8 displays, each of 45 papers (about 59.21%) provides a fully automated approach for the investigation of web test cases. The automated approach for web test cases is the most popular topic because it can reduce the human effort for web testing. For

example, S11 [Leotta et al., 2015a] proposes an automated method for migrating test cases from DOM-based web tests to visual web tests. And S27 [Biagiola et al., 2019b] presents an automated approach for web test generation.

The approaches in 12 studies (approximately 15.76%) are semi-automated, including automated and manual aspects. Article S46 [Mariani et al., 2014] try to generate test inputs by exploiting the web data. To achieve this goal, it provides a tool called Link to create a model of input values and extract data to generate test inputs. As they stated, they can automatically generate test inputs from the model. However, manual intervention is required during model creation to redefine the model.

In another set of five papers (about 6.58%) presented manual approaches and then are classified as manual. And nine studies (11.84%) are not clear because of the lack of information on automatic levels.

3.4.7 RQ 1.7 - Provided tools

Table 3.4 demonstrates the different characteristics of existing tools developed by researchers for web applications. The first column is the study number, and not all studies in our pool provide tools for their techniques. In their papers, the researchers named the implemented tools, which are listed in column two. The third column is the target of each tool, such as test case generation, test minimization, test case robustness, test case dependency, test sequence repair, etc. The fourth column describes the year that each tool was implemented. And the last column represents if the tool is available to be download. The symbol $\surd$ indicates that a download link of each tool is provided in the paper (may still be downloadable, or the download link is not supported now). And the symbol $\times$ means that no download link of the tool is provided in the paper.

As shown in Table 3.4, there are 17 studies (S4, S9, S12, S16, S17, S27, S32, etc.) that provide tools for the generation of web test cases. Five tools (S22, S50, S70, S72, and S73) can repair damaged web test cases. Three tools (S56, S58, and S59) are designed to improve the robustness of web test cases. And three tools (S9, S10, and S43) attempt to create test models for web applications. Besides, other tools are provided for achieving other different goals (for example, test case priority, test case migration, test case metrics, test minimization, test case dependency, etc.).

Table 3.4 – Tools presented in studies

Study	Tool Name	Target	Year	Available
S3	Recommender	test case priority	2018	$\times$
S4	Artemis	test case generation	2011	$\surd$
S9	ReWeb TestWeb	model creation test case generation	2001	$\times$

Continued on next page

Table 3.4 – *Continued from previous page*

Study	Tool Name	Target	Year	Available
S10 S21 S75	APOGEN	model creation	2017 2016 2015	×
S11 S51 S52	PESTO	test case migration	2015 2014 2018	√
S12	MBUITC	test case generation	2019	√
S16	SAT	test case generation	2018	×
S17	SWAT	test case generation	2011	×
S18	DWASTIC	test case metrics	2010	×
S22	COLOR	test case repair	2019	×
S26	FlakcLoc	test case flakiness	2019	×
S27	DIG	test case generation	2019	√
S28	DomCovery	test case metrics	2014	√
S32	FEEDEX	test case generation	2013	√
S33	Testler	test minimization (reduction)	2018	√
S35	WATEG	test case generation	2013	×
S36	WAM	test case generation	2007	×
S39 S40	ATUSA	test case generation	2009 2012	√
S41	JSART	test case generation	2012	√
S42	Testilizer	test case generation	2014	√
S43	KeyjaxTest	model creation	2019	×
S50	ReAssert	test case repair	2010	√
S53	WAM-SE	test case generation	2009	×
S56	ROBULA	test case robustness	2014	√
S57	CRAWLJAX	test case generation	2010	√
S58	ROBULA+	test case robustness	2016	×
S59	ATA-QV	test case robustness	2014	√
S60	Subweb	test case generation	2017	×
S61	SART	test case regeneration	2012	×
S65	TEC	test case execution	2017	√
S70	VISTA	test case repair	2018	√
S71	WaRR	test case generation	2011	×
S72	WATER	test case repair	2011	×
S73	WATERFALL	test case repair	2016	×

Continued on next page

Table 3.4 – *Continued from previous page*

Study	Tool Name	Target	Year	Available
S74	TEDD	test case dependence	2019	√

3.4.8 RQ 1.8 - Web Applications Under Test

Figure 3.9 describes the number of WAUTs used in each article. As shown, 22 articles (about 28.95%) validated their approaches using one web application as a case study. And 7 articles do not use WAUT to validate their approaches. There were ten articles (about 13.16%) that evaluated their approaches with two WAUTs. And there were 5, 7, 5, 10 articles respectively used 3, 4, 5, 6 WAUTs for validation. Three articles (approximately 3.95%) used more than 10 WAUTs for experimental validation. The total number of WAUT (includes duplications) used in all articles is 273, and the average number of WAUTs used per article is 3.6. After removing duplicates, a total of 119 unique WAUTs were used.

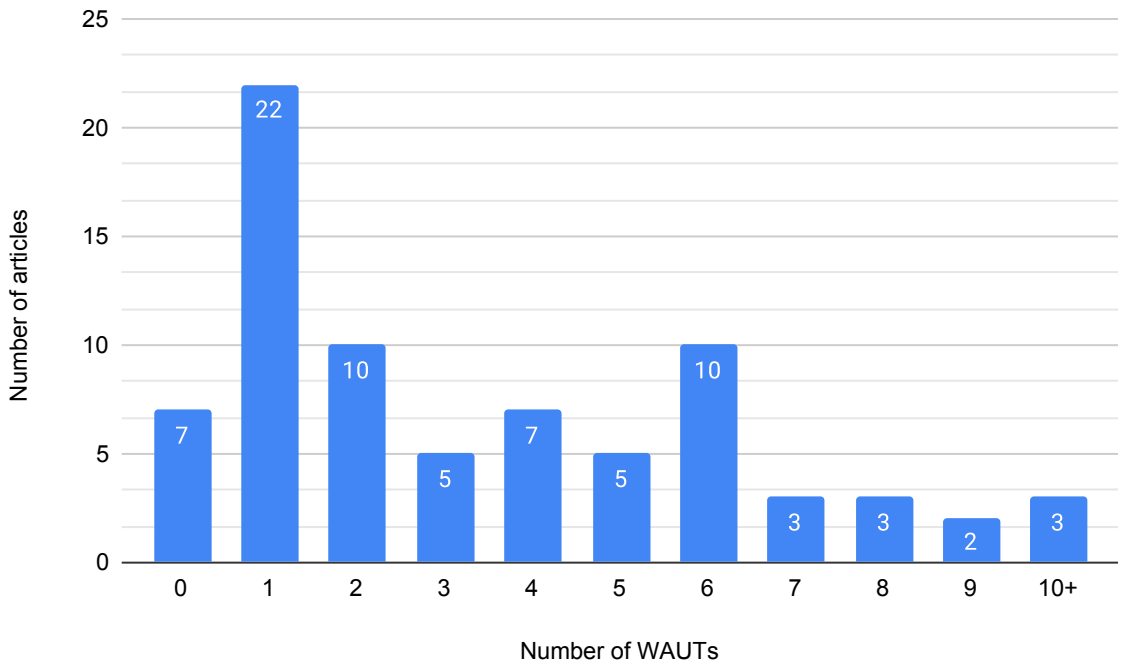


Figure 3.9 – The number of WAUT in each article.

There were too many different WAUTs used. For the clarity of this thesis, we list the top used 20 WAUTs in table 3.5. As it depicts, the first column shows the names of WAUTs

that be provided by researchers. The LOC of each WAUT is in the second column. The third column is the development language of each WAUT, such as PHP, JSP, or Java. A brief description of each WAUT is given in the fourth column. The last column represents the number of times each WAUT was selected to validate the research approach. We provide more information about WAUT on Google spreadsheets [1].

Table 3.5 – WAUT presented in at least three papers, ranked by amount of papers.

Web Names	LOC	Language	Description	Amount
AddressBook	35675	PHP	address/phone book	16
Bookstore	19402	JSP	an e-commerce bookstore site	13
Claroline	352537	PHP	collaborative learning environment	11
PPMA	575976	PHP	password manager	9
MRBS	34486	PHP	meeting rooms manager system	8
MantisBT	141607	PHP	a bug tracking system	7
Collabtive	264642	PHP	collaboration software	6
TuduList	23000	Java	management of personal todo lists	5
CPM	9300	Java	a course project manager	4
Joomla	312978	PHP	a content management system	4
PetClinic	6100	Java	a veterinary clinic web application	3
Schoolmate	8183	PHP	School admin system	3
Webchess	8589	PHP	Online chess game	3
Timeclock	23403	PHP	Employee time tracker	3
PHPBB	22280	PHP	Customisable web forum	3
PHP-Fusion	256899	PHP	a content management system	3
PhotoGallery	6000	PHP	an online photo gallery	3
WordPress	342097	PHP	a content management system	3
Classifieds	10759	JSP	Portal for advertisements	3
Events	7164	JSP	Portal for event announcements	3

We obtained 120 WAUTs from all articles, but not all articles reported the LOC size of WAUT during the article publication. Only 85 out of 120 WAUTs reported LOC size. Figure 3.10 shows the histogram of the LOC size of WAUTs. The average LOC size of WAUT is 89788.48. The minimum LOC size of WAUT (named DynamicArticles) is 156 in research S6 [Artzi et al., 2011]. The maximum LOC size of WAUT (called Zimbra) is 1025410 in research

1. https://docs.google.com/spreadsheets/d/1_9SDhR6TW98uMTSe0lDQg6a9L1kBoPsFkqpbg_p4Bk0/edit?usp=sharing

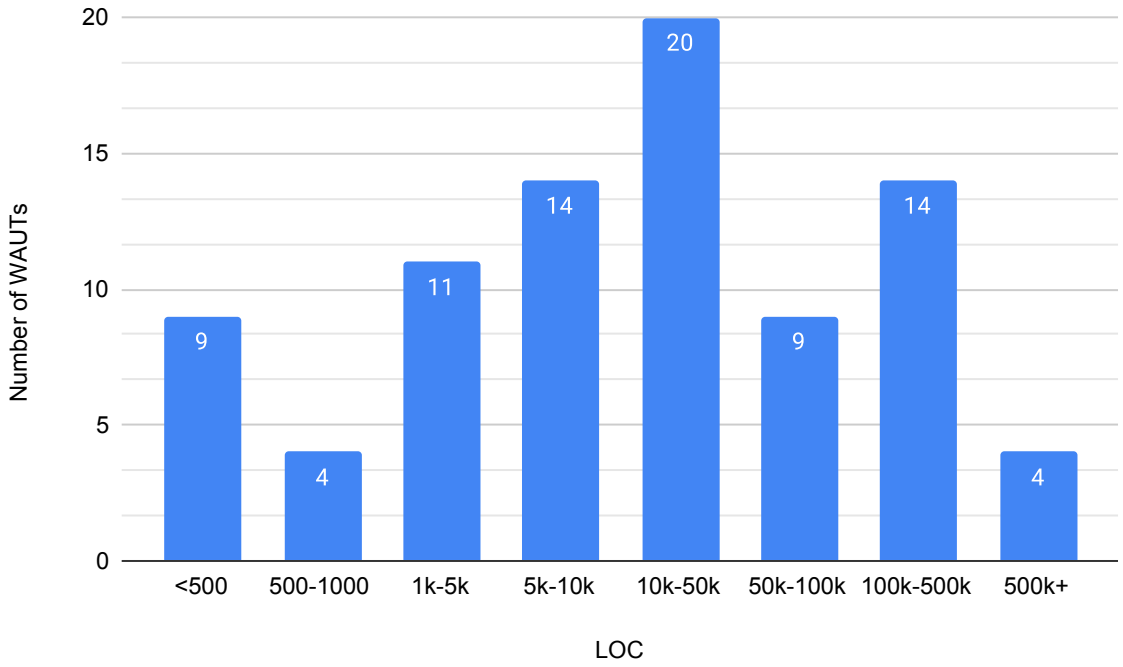


Figure 3.10 – The LOC of WAUTs.

S56 [Christophe et al., 2014]. As shown in Figure 3.10, many large WAUTs (more than 10 KLOC) are selected for validation in the articles.

As we all know, the LOC size of WAUT may be different in different articles, because researchers have chosen different versions of WAUT. And some articles do not choose the entire WAUT, only use the subsystem of WAUT, which also makes the LOC size of WAUT different. Therefore, when WAUTs with different LOC sizes appear in different articles, we choose the maximum LOC size of this WAUT.

Figure 3.11 shows the number of WAUT development languages used in the articles. From the distribution of development languages, we can see that PHP is the most commonly used language for WAUT. There are 36 kinds of WAUTs (about 30%) are developed using PHP. Not all WAUTs provide development languages in published articles. 29 WAUTs are classified as "N/A", meaning "not applicable". Java is the second popular language of WAUT in the article pool. WAUTs are also developed using ASP.NET, C++, JSP, JS, or Python.

We want to know the types of WAUT in the articles, such as academic, open-source, and commercial. Figure 3.12 depicts the histogram of WAUT types. 80 WAUTs (about 66.67%) are open source. Commercial WAUT accounts for 15% (18 WAUTs out of 120 WAUTs). And there are only seven academic experiments WAUTs. As shown in the figure, a large number

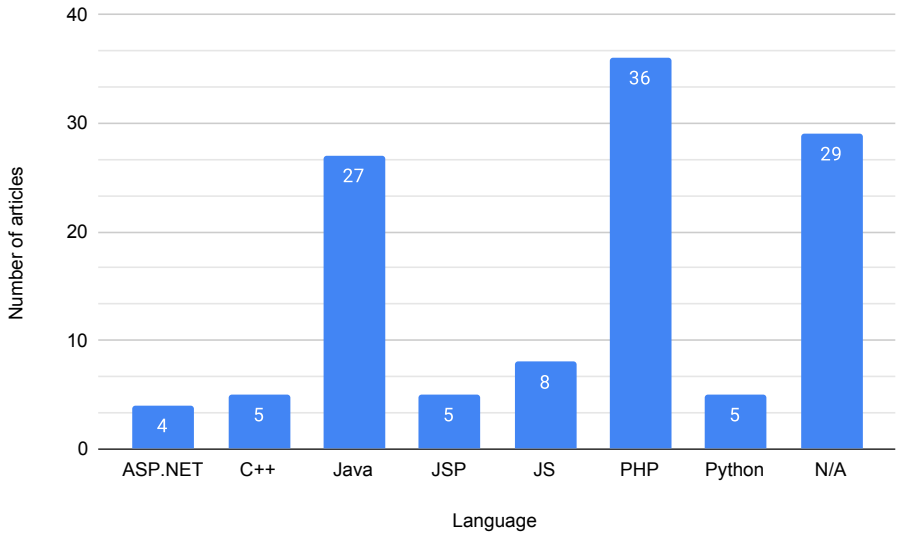


Figure 3.11 – The development languages of WAUT.

of open-source WAUTs were selected during the verification process of the article, while the number of academic WAUTs was relatively small.

3.4.9 RQ 2.1 - Publication trend per year

The annual publication volume of articles for web test cases is depicted in Figure 3.13. In terms of publication year, Figure 3.13 shows that the number of articles used for web test cases fluctuates. In other words, the publication volume trend for web test cases is relatively not stable. For example, there are 0 articles in 2002. There has been a relatively significant reduction in 2006, 2009, and 2015. However, it is an overall upward trend, which means more developers focus on web test cases to conduct automated web testing.

3.4.10 RQ 2.2 - Citation analysis of publications

In this subsection, we try to analyze the citation of each article based on counting of citations. We extracted the citation data of each article from Google Scholar on October 16, 2020. For papers that appear both as conference publications and as journal extensions, we will respectively count the citations of these two editions. Figure 3.14 visualizes the citation count of each article vs. publication year as an X–Y plot.

Since these research papers were published in different periods, the publication year of each paper should be considered when analyzing the number of citations. Therefore, we take the average of the normalized values [Nassiri et al., 2013] as the following:

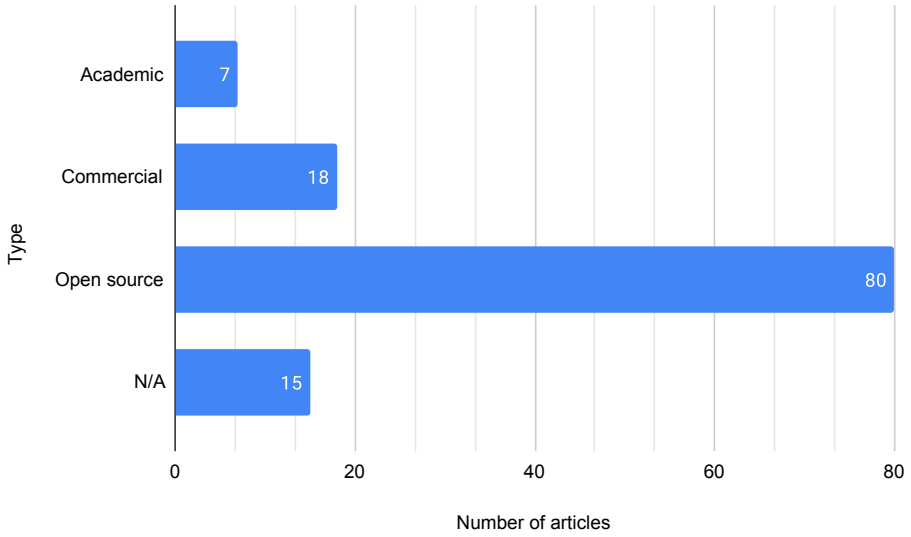


Figure 3.12 – The types of WAUT.

$$NSI = \frac{Citations}{2021 - Year} \tag{3.1}$$

For example, the approach Water of paper [Choudhary et al., 2011] has 64 total citations as of this writing and was published in 2011. Thus, its normalized citations is calculated as: NSI(Water) = 64/(2021 - 2011) = 6.4, and we generate Figure 3.15 after normalizing the citations.

As Figure 3.15 shown, the normalized citations returns the average number of citations of a paper each year since its publication year. The top publication with the normalized citations (37.11) is [Ricca and Tonella, 2001]. And paper [Andrews et al., 2005] is the second most cited publication based on the normalized citations (29.73).

3.4.11 RQ 2.3 - Top related venues

In order to rank the related venues, we count the number of our selected papers published in each venue. Table 3.6 shows the top related venues ranked by the number of papers. It indicates the top 10 venues in this table. Among them, there are 8 conferences/Symposium, and two journals. There are many major software engineering conferences and journals on this list. For example, the venue with the most papers (9 papers) is ISSTA, which accounts for 11.84% of the total number of papers. Recently, it has attracted and published many papers related to web test suites.

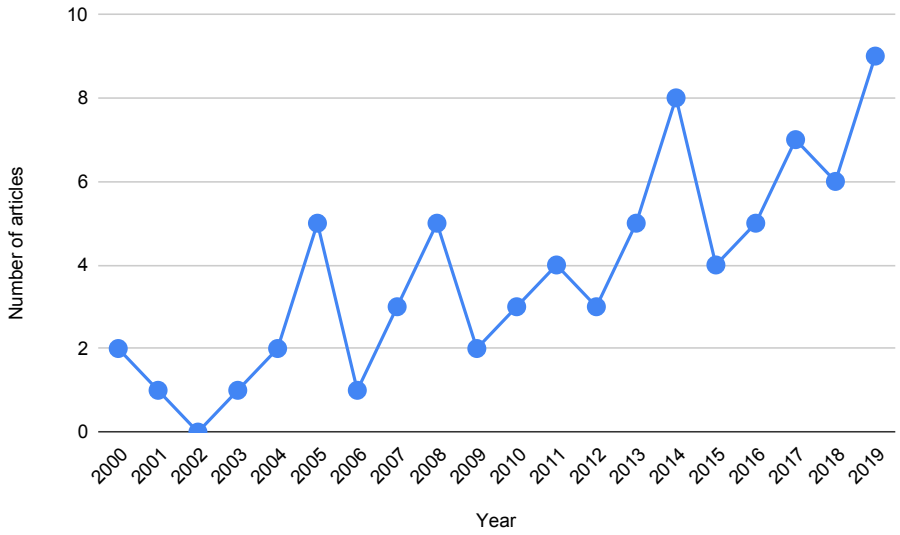


Figure 3.13 – The publication trend per year.

Table 3.6 – Venues ranked by the number of papers

Venue	Acronym	#
International Symposium on Software Testing and Analysis	ISSTA	9
International Symposium on Foundations of Software Engineering	FSE	6
International Conference on Software Engineering	ICSE	6
International Conference on Software Testing	ICST	6
International Conference on Automated Software Engineering	ASE	4
International Conference on Software Maintenance and Evolution	ICSME	4
IEEE Transactions on Software Engineering	TSE	2
International Symposium on Software Reliability Engineering	ISSRE	2
Journal of Systems and Software	JSS	2
ACM Symposium On Applied Computing	SAC	2

3.5 Discussions

In this section, we summarize our findings of this systematic mapping study and the threats to validation.

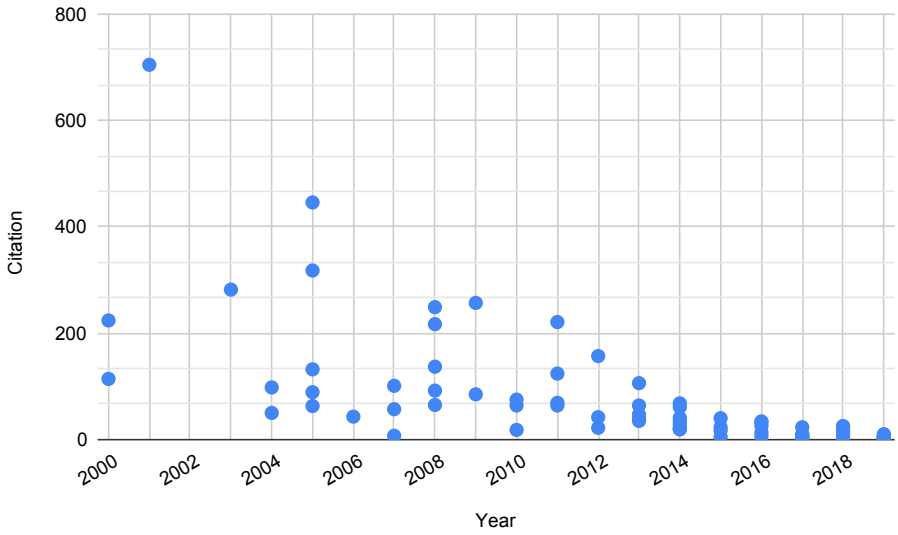


Figure 3.14 – Citations of each article vs. publication year.

3.5.1 Findings

RQ 1.1 shows that most of the articles (80.26%) have proposed techniques for web test cases. And some articles have made more than one contribution to web test cases. For RQ 1.2, most papers (68.42%) have empirically verified their approaches. However, these validations are conducted on empirical experiments, not industrial evaluations. Only ten articles (13.16%) conduct industrial evaluations. As a result, it still needs much attention for researchers to validate their techniques on industrial web applications.

RQ 1.3 shows that most articles focus on the generation of test cases, and other activities require more attention, such as repair of test cases, dependency of test cases, priority of test cases, etc. For RQ 1.4, most articles used model-based techniques for web test cases. A variety of various technologies have been used and proposed for web test cases. And there are some articles using more than one technique in their approaches.

In term of the location in test case, the majority of the work has focused the whole test case in the past (RQ 1.5). The second majority focused on the input of test case. A few articles paid attention to action and assertion of test case.

RQ 1.6 indicates that most articles proposed automated level approaches for web test cases. There are also a mix of manual and automatic methods. Few articles used manual approach in web testing.

RQ 1.7 shows that many articles proposed tools in their studies. Some tools (50%) can be downloaded for free, but 50% of the provided tools can not be download (many are not available). It needs more attention for researchers to provide available tools for testers.

For RQ 1.8, a lot of WAUTs are used by researchers to validate their approaches. Most

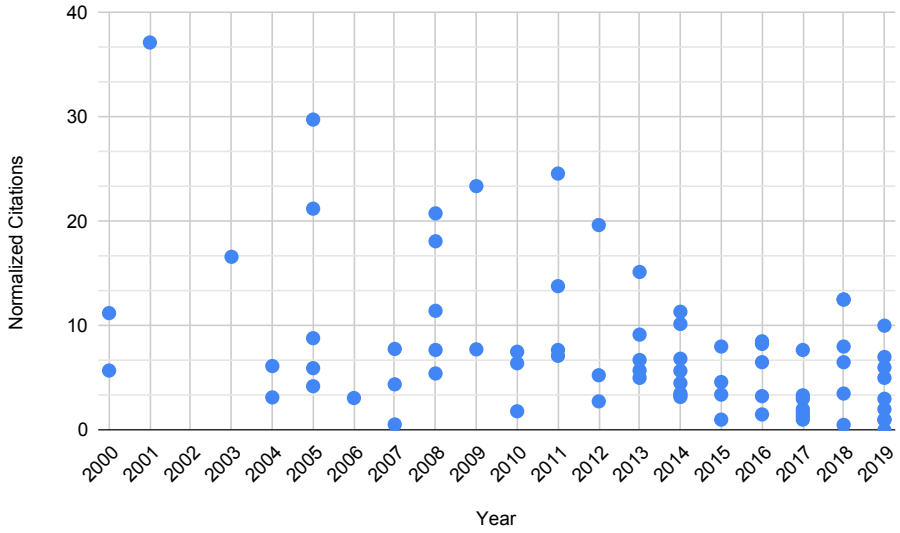


Figure 3.15 – Normalized citations of each article vs. publication year.

of articles only use one WAUT in the process of validation. Few articles use more than ten WAUTs for emprical verification. As Table 3.5 dipicts, the AddressBook are utilized by 16 articles, which are most commonly used. As shown in Figure 3.10, the LOC size of most WAUTs ranges from 10k to 50K. The WAUTs used in the article have various (LOC) sizes. Figure 3.11 illustrates that most of WAUT's development language is PHP. And the second most commonly used development language of WAUTs is Java. Figure 3.12 shows that most of the WAUTs are open source, which are avalible online freely.

RQ 2.1 reveals that as time goes on, the number of articles per year increases. RQ 2.2 presents the citations of each article and their normalized citations. The number of citations indicates the quality of each article, and there is no necessary correlation with the year. RQ 2.3 shows the venues ranked by the number of papers in our pool. And there are many major software engineering conferences and journals in Table 3.6.

3.5.2 Threats to validation

In this subsection, we discuss potential threats to validity of this mapping study. The results of systematic mapping study may be affected by many factors, such as the researcher who conducts the research, the selected data source, the search term, and the range of selected time.

As we discussed before, we have defined our inclusion and exclusion rules. We have divided our groups to minimize the personal biases of our team members. When our team members disagree with each other, we will hold a meeting and discuss together until we

reach an agreement. However, like other systematic mapping studies, personal bias can not be eliminated. It is a threat to our mapping study. But we believe that our main conclusions drawn from the identified articles should not deviate from our findings.

In Section 3.3.1, we have introduced the search terms and databases utilized in this mapping study. To obtain a complete study set that can cover the research topic, we systematically generate search keywords. However, since there are multiple choices and different combinations of terms for web test cases, the list may be incomplete, and alternative terms may affect the number of articles found, which is one of the threats to our mapping research. We have tried the primary electronic databases that our university can access. We also performed a snowballing process to complete the research paper set. However, some articles in the alternative databases may not be found, which is a threat to this mapping study.

3.6 Conclusion

Due to the complexity and dynamics of web applications, it is well known that test automation is a huge challenge for web developers. This is why many researchers work in this domain. In this paper, we present the first systematic mapping of articles in the area of web test cases, published between 2000 and 2019. Moreover, we provide the first bibliometric analysis of web test cases to understand annual publication trends, citations and venues.

Our system mapping study shows that web test automation is an active research area, and the number of publications of web test cases is increasing. This mapping study provides guidance by discovering research areas that require more attention to help researchers plan future work. For example, compared with test case generation, researchers also need pay attention to other aspects, such as test case repair, test case dependence, etc.

Our mapping study present guidence for testers to choose the tools according to the activities of web test case. For example, developers and testers can find tools with target activities in Table 3.4. In addition, researchers interested in web test cases can shoose the WAUTs shown in our map.

CHAPTER

4 An approach for DOM-based web test suite repair

How to repair test scripts during web test evolution? The rest of this chapter is structured as follows: We first introduce the motivation. We then describes our approach that proposes substitutes for broken actions efficiently. We also conduct an empirical evaluation to validate our method. Finally, we get empirical results and conclusions.

Contents

4.1 Introduction

As we presented in Chapter 2, the test cases sometimes break due to the evolution of web application. To scope our approach, we also presented the break type of DOM-based web test case that we are target to repair in Chapter 2. In this chapter, we present our approach to repair the broken tests. We also summarize our contributions for web test repair.

In this chapter, we try to focus on web domains other than other GUI apps to repair DOM-based test cases. We build Test Suite Graphs (TSG) for different releases of a web application and compare these two TSGs to help developers repair their broken tests. To this extent, we propose an approach named WebTestSuiteRepair, which will compare TSGs to identify substitutes for broken actions, and hence that developers can utilize this approach to repair broken tests automatically. In this study, we make the following contributions:

— An approach to generate test suite graphs of web applications, which helps to repair web test cases.

— An algorithm to automatically repair the DOM-based test suite by comparing TSGs of web applications.

— An implemented tool WTSR for testers or developers to repair broken web test cases.

— An empirical evaluation of our approach to repair broken tests for three real web applications.

4.2 Methodology

This section presents our approach WebTestSuiteRepair (WTSR), which aims to repair the test suite for web applications. Atif Memon[Memon et al., 2003] and Zebao Gao[Gao et al., 2015a] present the event-flow graph (EFG) of Graphical User Interfaces for software test. Different from the existing technology, our approach WTSR generates Test Suite Graphs (TSGs) from a test suite of test cases of the web application. It creates TSGs for two releases of a web application separately. And it repairs broken test cases by comparing these two TSGs. In this section, we first introduce an overview of WTSR and then detail all of its parts.

4.2.1 Overview

Fig. 4.1 presents an overview of WTSR. We assume that testers capture the initial test cases from *Release* 1 of a web app.

First, WTSR tries to execute the initial E2E tests on *Release* 1 to build the TSG. For the sake of clarity, the graph for *Release* 1 is named *TestSuiteGraphRelease1* (TSGR1). The graph for *Release* 2 is named *TestSuiteGraphRelease2* (TSGR2). TSGR1 and TSGR2 are both TSG, corresponding to different releases.

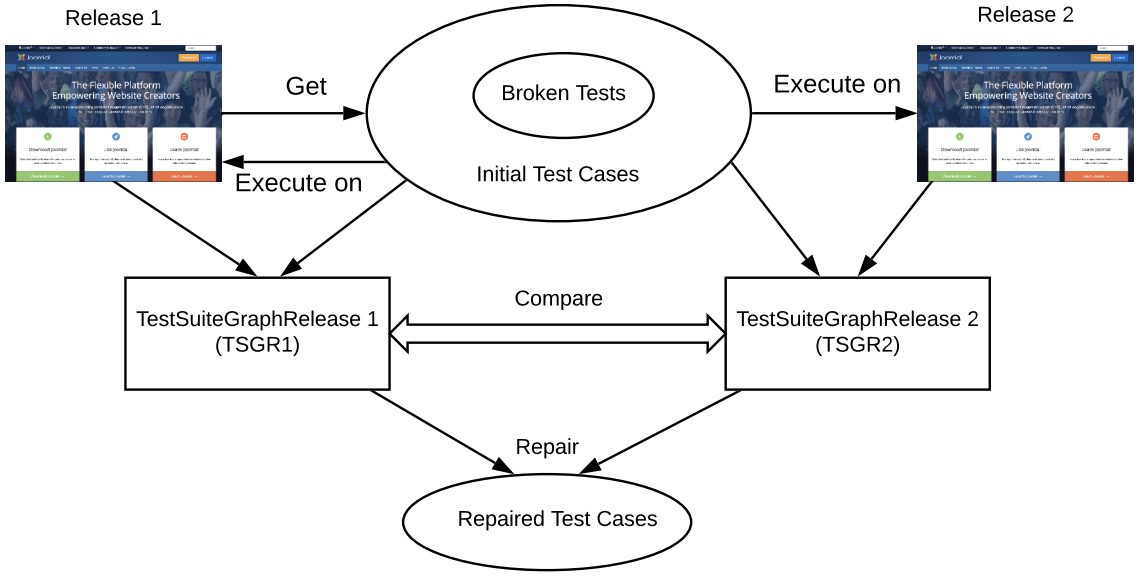


Figure 4.1 – The architecture of web test suite repair.

Second, WTSR runs these initial test cases on Release 2 to build the TSGR2. It creates the same type of 5-tuple graph through the same process of TSGR1. During this process, it will crawl HTML element data from the web page after each action in the test case to create the selectors. After creating efficient and robust CSS selectors for HTML elements, WTSR can generate different kinds of actions. These crawled candidate actions are part of TSG.

Third, WTSR compares these two TSGs to repair the test suite. It repairs broken cases by utilizing a substitute to replace broken action in *Release* 2. It set the no longer useful test cases as a delete state by finding the deleted actions in the TSGR2. The following subsections will present all these steps in detail.

4.2.2 Create Test Suite Graph Release 1

For web application developers, we assume that they have written test suite composed of test cases for web application's *Release* 1, which can be described as an initial test suite, the input of our approach WTSR. WTSR plays these test cases over the web application's *Release* 1 to create the TSGR1 (line 2 in Algorithm 1). To create TSG, it will run each test case (from line 7 to line 9).

To play a test case, it will perform cach action to interact with the web application (from line 12 to line 31 in Algorithm 1). We use Puppeteer [1] to run each action and get the web page (line 14). On different pages, actions with the same content or selector maybe have a different meaning. So we use URLs to distinguish these actions. For each test case, it

1. https://github.com/puppeteer/puppeteer

Algorithm 1 Algorithm to automatically repair web test suite

Require: Initial Test Suite (ITS_{set}) // A set of Test Cases
Ensure: The Repaired Test Suite
1: **procedure** WEBTESTSUITEREPAIR:
2: $TSGR1 \leftarrow CreateTSG(ITS_{set})$
3: $TSGR2 \leftarrow CreateTSG(ITS_{set})$
4: $TestRepair \leftarrow Compare(TSGR1, TSGR2)$
5: **end procedure**
6: **function** CREATETSG(ITS_{set})
7: **for** $Each(testCase_i)$ in ITS_{set} **do**
8: $TSG \leftarrow Play(testCase_i)$
9: **end for**
10: **return** TSG
11: **end function**
12: **function** PLAY($testCase_i$)
13: **for** $Each(action_i)$ in $testCase_i$ **do**
14: $WebPage \leftarrow browser.run(action_i)$
15: **if** $WebPage.true$ **then**
16: $TSG.url \leftarrow saveUrls(WebPage.url)$
17: $TSG.action \leftarrow saveRunnedActions(action_i)$
18: $TSG.link \leftarrow saveLinks(action_{i-1}, action_i)$
19: $Element_{set} \leftarrow browser.crawl(WebPage)$
20: $CA_{set} \leftarrow generateCAs(Element_{set})$
21: $TSG.candidate \leftarrow CA_{set}$
22: **else**
23: $TSG.breakInfo \leftarrow getBrokenAction(WebPage)$
24: $TSG.suite \leftarrow saveBrokenCase(testCase_i)$
25: **end if**
26: **end for**
27: **if** $ActionsInTestCase_i RunSuccessfully$ **then**
28: $TSG.suite \leftarrow saveSucceedCase(testCase_i)$
29: **end if**
30: **return** TSG
31: **end function**

records the URL of each action and saves these URLs to MongoDB (line 16). It can extract the actions and links between actions from initial test cases (lines 17 and 18). And, it crawls web pages to obtain DOM to extract web elements (line 20). The target web elements (e.g., a hypertext reference link, a text field, a single box, etc.) are the parameters of the candidate actions. It then creates the candidate action set using these elements (line 21). To generate

candidate actions, we use the JS library [2] that were developed before. If an action of test case fails, this failed test case and its interrupt information will be saved to MongoDB (lines 23 and 24). If the test case runs successfully, this succeeded case will be saved to MongoDB (line 28 in Algorithm 1).

The TSG can be regarded as a 5-tuple structure <S, U, A, L, C>:

— S is a test suite that consists of a set of test cases representing all tests.

— U is a set of URLs representing all web pages of test cases.

— A is a set of actions representing object events in the URLs, and actions in different URLs are different.

— L ⊆ A × A is a set of links that may follow edges between actions. (Ai, Aj) means Aj executes immediately after Ai.

— C is a set of candidate actions after each test action.

In Section 2, we have presented a test case example. Now we use two similar test cases to explain 5-tuple <S, U, A, L, C> architecture of TSG. For example, TSGR1 in Fig. 4.2 keeps these two test cases in the mongo database. In Fig. 4.2, TS_1 includes A_1, A_2, A_3, A_4, and assertion. TS_2 includes A_1, A_5, A_6, A_7, and assertion. TSGR1 also keeps these actions and their links in the database. Every action belongs to a web page (URL), and we use the URL and its selector to distinguish them. So the URL of every action is stored in MongoDB. The link (A1, A2) means *Action* 2 is performed directly after *Action* 1. During the execution of the test case, WTSR crawls the candidate actions after each action and saves them in the database. For TSGR1, it is only a simple example in Fig. 4.2. TSGR1 is more complicated in an actual web application that contains more test cases and a lot of actions.

Candidate actions can be crawled from the web page after each test case command. These candidate actions are potentially used to replace broken action to repair a breaking case.

> **[. Candidate Action]** Candidate Action is an action that can be performed after the previous action. Candidate actions are represented as $CA_i = CA_1, CA_2, \ldots$ In the case of a given web page, all CSS selectors on the web page can be crawled to generate a set of candidate actions.

For example, every action on the left side of Fig. 4.2 has many candidate actions that are recorded from the HTML web page. On the right side of the Fig. 4.2, $CA_1, CA_2, CA_3, \ldots CA_n$ are the candidates actions after the previous action A_i. The candidate actions of A_i constitute the action set that can be performed after A_i. Candidate action has the same 4-tuple structure < type, selector, text, URL> as the actions in the initial test case.

2. https://github.com/webautotester/scenario

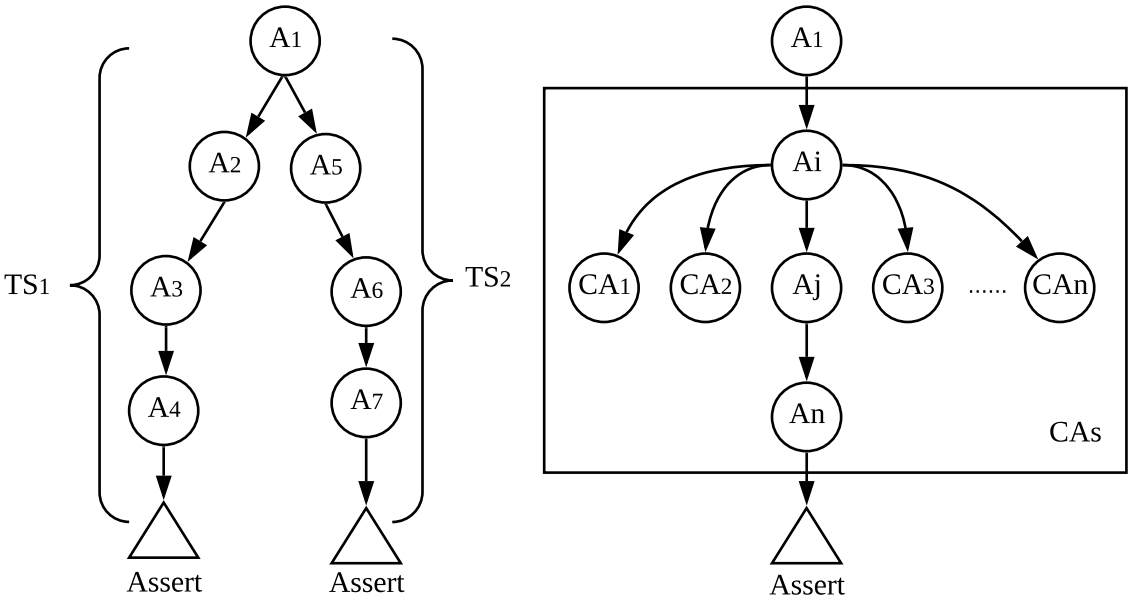


Figure 4.2 – The test suite graph release 1.

4.2.3 Generate Test Suite Graph Release 2

In this step, we aim to generate TSGR2 by executing all the test cases on web application release 2. This process for generating TSGR2 is the same as the process of creating TSGR1. In Algorithm 1, input test suite and use the same function to create TSGR2 (line 3). For all the test cases, run them on web app release 2 from line 7 to line 9. The function to play each test case is from line 12 to line 31, which has been presented details in Section **??**. Actions and links are also saved to TSGR2 as the same as TSGR1 (lines 17 and 18). And there appeared interruptions (lines 23 and 24) in some test cases during this process of creating TSGR2 because of the evolution of web application.

In Algorithm 1, after performing each action, WTSR will also crawl the web page to obtain DOM to extract web elements, which can be used to generate candidate actions (from line 19 to line 21). However, during the process of creating TSGR2, some test cases are broken due to web evolution. So WTSR needs to get the broken test cases and the stopped actions (lines 23 and 24). And it will repair these damaged cases in the next Section **??**. For example, A_3 on the left side of Fig. 4.3 is a broken action that can not be performed. So TS_1' keeps the breaking information with A_3. Fig. 4.3 presents an example of TSGR2 that needs to be replenished and updated in the next Section **??** by comparing TSGs.

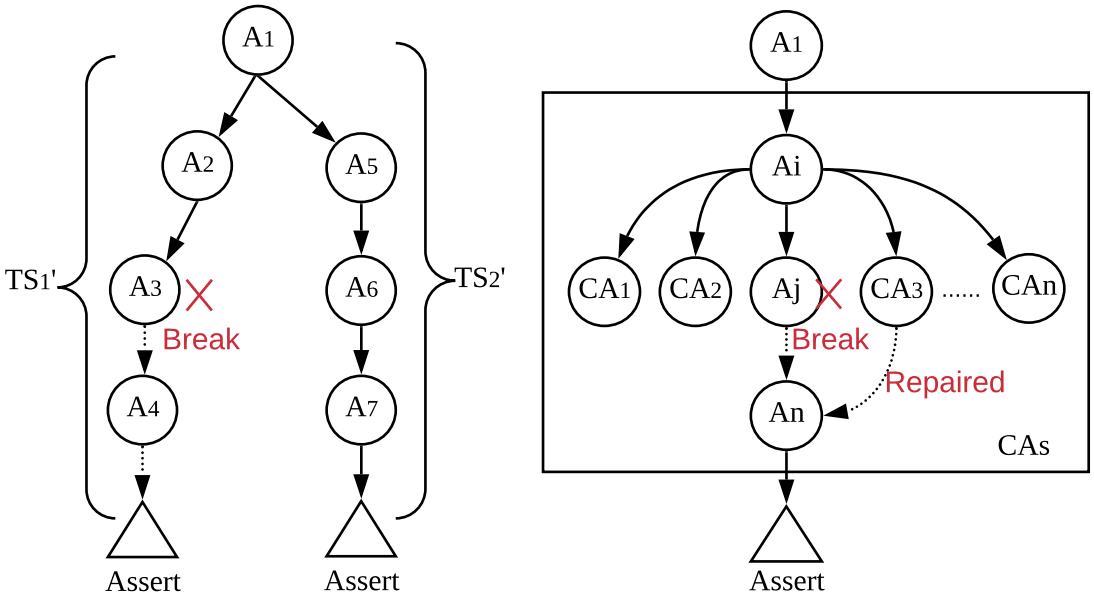


Figure 4.3 – The test suite graph release 2.

4.2.4 Compare TSGs

After creating these two different TSGs, the next step is to compare them to repair test cases. In Algorithm 2, it compares the TSGs to find the broken tests (line 2). For these damaged tests, try to identify a substitute action from the candidate set as a replacement of broken action to repair the breaking test case (from line 3 to line 20). During the process of repairing the test case, WTSR gets the candidate action set (line 6). It then compares the candidates for different releases to get the candidate set(line 7). And WTSR orders these candidate actions (line 8) owing to that top-ranked action is more likely to be a substitute. This is because elements close to each other are more similar [Heil et al., 2016]. For instance, Action 4 is broken in Fig. 4.3, and we identify the substitute action CA3 from ordered candidates to repair the break test in Fig. 4.3.

In Algorithm 2, WTSR then tries to find the correct substitute to replace the broken path on the right side of Fig. 4.3. For this purpose, we present a loop in approach WTSR to try each candidate action to repair the broken test from line 9 to line 18 in Algorithm 2. And play the new case (line 11) to judge whether this replacement is correct. If this new case can run successfully (line 12), then it means the replacement is correct, and we will push to store it in *RepairedTestCase* (line 13). If this test case has more than one breakage, WTSR will redo the repair process again for other broken actions (line 16). The function of RedoRepair is from line 4 to line 19. And Algorithm2 feedbacks to TSGs and repair results in Algorithm 1. If we can not find the substitute action, there may be a mis-selection in

Algorithm 2 Algorithm to compare TSGs

Require: TSGR1 and TSGR2
Ensure: The Repaired Test Cases

```
1:  function COMPARE(TSGR1, TSGR2)
2:      BrokenTestR2_set ← getBrokenTests(TSGR2)
3:      for Each(TestCase_i) in BrokenTestR2_set do
4:          breakLocation ← get(TestCase_i.breakLocation)
5:          for j ← 1 to breakLocation do
6:              (R1CA_set, R2CA_set) ← getCA(action_i)
7:              CA_set ← compareCA(R1CA_set, R2CA_set)
8:              OCA_set ← Order(CA_set)
9:              for Each(CA_i) in OCA_set do
10:                 CandiTestCase ← replace(breakAction, CA_i)
11:                 CandiResult ← play(CandiTestCase)
12:                 if CandiResult.runSuccess then
13:                     RepairedTestCase.push(CandiTestCase)
14:                 end if
15:                 if CandiResult.anotherBreak then
16:                     RedoRepair(CandiTestCase)
17:                 end if
18:             end for
19:         end for
20:     end for
21: end function
```

previous actions. For example, the locator of action A_i in Fig. 4.3 does not change, but the corresponding element on the web page is different from R1. So action A_i is a mis-selection resulting in the breakage of action A_j. We use the same method to find a substitute for action A_i to repair the broken test. Therefore, we make a loop to find a substitute for mis-selection action to repair broken tests (from line 5 to line 19).

To rank these candidate actions, WTSR will calculate the distance between candidate actions and broken action (from line 2 to line 4 in Algorithm 3). By comparing the selectors (i.e. #id-left > DIV:nth-child(1) > A:nth-child(1)) between broken action and each candidate action, WTSR can get their distances. WTSR then sorts these candidates according to their distances with broken action from near to far (line 6 in Algorithm 3). WTSR then feedback these ordered candidates to TSGs (line 8 in Algorithm 2). WTSR tries to identify correct action from sorted candidate actions that can repair the broken test case (line 10 in Algorithm 2).

[. Action Distance] Action Distance is the distance between two action nodes of a DOM Tree.

$$distance(n1, n2) = Depth(n1) + Depth(n2) - 2 * Depth(LCA) \quad (4.1)$$

"n1" and "n2" are the two given action nodes. "root" is the root of a given DOM Tree. "LCA" is Lowest Common Ancestor [a] of n1 and n2. *Depth* is a function to get the depth of a node. Intuitively, the depth of a node is the number of edges from the node to the tree's root node. *distance(n1, n2)* is the function to calculate the distance between n1 and n2.

a. https://en.wikipedia.org/wiki/Lowest_common_ancestor

Algorithm 3 Algorithm to order candidate actions

Require: Candidate actions, Broken action
Ensure: The ordered candidate actions

```
    function OrderCandidate(Candidate, BrokenAction)
2:      for Each(CA_i) in CA_set do
            CA_i.distace ← distance(break.locator, CA_i.locator)
4:      end for
        for Each(CA_i) in CA_set do
6:          OCA_set.push(identifyShortestDistance(CA_i))
        end for
8:  end function
```

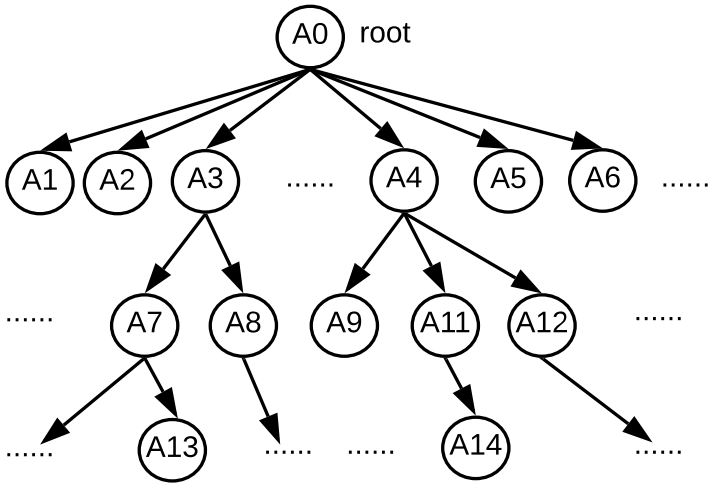


Figure 4.4 – A distance example between two action nodes in a DOM tree of a webpage.

To calculate the distance between candidate action and the broken action, WTSR uses a DOM tree of a web page. All the actions in Fig. 4.4, from A0 to A14 and other actions of ellipsis, are on the same web page. One of them is broken action, and other actions are candidates. We assume A7 is the broken action to explain how to calculate distance. WTSR will calculate the distance between A7 and other candidates. For example in Fig. 4.4, distance(A7, A8) = Depth(A7) + Depth(A8) -2*Depth(A3) = 2 + 2 - 2*1 = 2. The result of distance(A7, A2) is 3. And the distance(A7, A9) is 4. WTSR then sorts these actions according to the distances.

4.3 Evaluation

To evaluate the feasibility and efficiency of our approach to evolving the test suite, we choose three research subjects and present a set of experiments with quantitative analysis. We follow the guidelines by Wohlin [Wohlin et al., 2012] on designing and reporting empirical studies in software engineering. Then we present experimental results and threats to the validity of our approach.

To evaluate the effectiveness of our approach WTSR in this study, we try to answer three research questions:

— RQ1: Is it possible to generate test suite graphs for two different releases of a web application?

— RQ2: Is it possible to automatically and efficiently repair these test cases for web applications by comparing their test suite graphs?

— RQ3: How effective is our test suite evolution method?

The experiment verification is based on three real web applications. For RQ1, we count the actions and links in each TSG. We count the candidates after each action in TSG. And we count broken tests in TSG. We also record the execution time that indicates how long it takes to create TSG. For RQ2, we count the number of repaired test cases. For RQ3, we record how long it takes to repair broken test cases.

4.3.1 E2E Test Subjects

Based on this research, we implemented a tool called WTSR [3]. To apply our tool to E2E tests of web applications, we have searched and selected web apps on SOURCEFORGE as our subjects of this experiment. When we search the test subjects, we input the "web" as keywords to search on SOURCEFORGE. Then, we got 65,074 programs as search results (searched on April 18, 2019). However, there are too many programs to select by reviewing each one of them. Therefore, we designed a filtering process and criteria to systematically

3. https://github.com/WebTestSuiteRepair/WebTestSuiteRepair

select subjects that: (i) sort them by "most popular", (ii) filter from the most popular 100 samples, (iii) the program should be a web application, not a desktop or mobile application, (iv) the program is a library, package or tool that is not a web application should be excluded, (v) the program has at least 100 releases.

Table 4.1 – E2E Test Subjects of Web Applications

Web App	Releases	# Release 1	# Release 2
Joomla	228	3.6.0 (Jul 12, 2016)	3.7.0 (Apr 25, 2017)
Moodle	325	3.5.0 (May 16, 2018)	3.6.0 (Dec 2, 2018)
Dolibarr	112	5.0.0(Feb 27, 2017)	6.0.0 (Aug 30, 2017)

As shown in Table 4.1, after we filter the searching results, we choose three web apps Joomla, Moodle, and Dolibarr. They are real subjects that can contribute to evaluating the potential performance and efficiency of our framework in a real test environment. Joomla [4] is a Content Management System (web system) that enables people to build websites. Moodle [5] is an open source learning platform (web platform). Dolibarr [6] is open-source web software to manage an organization's activity (contacts, suppliers, orders, etc.).

To obtain different releases of the web applications, we checked their releases under each repository at Sourceforge and Github. They are sorted by release date, and we can get all the releases for each target web application. The second column in Table 4.1 presents the total release number of each web application (searched on April 18, 2019). From these releases, we randomly select two different major releases for each web application. For these two major releases, we assume that the lower is Release 1 (R1) and the higher is Release 2 (R2). In Table 4.1, the third column is Release 1 of each web app with its release date. The fourth column is Release 2 of each web app with its release date. Therefore, R1 and R2 of each web application are arbitrary to avoid prejudice.

4.3.2 Process

After obtaining two releases of each web application, we try to design the experimental process. We first committed to creating an original test suite (a set of test cases) for each web application. Then we run these test cases on two releases of each web application to build TSGs and use our approach to evolve the test suite by comparing TSGs. At last, we compare WTSR with Water [Choudhary et al., 2011] to validate the efficiency and correctness of our approach.

4. https://sourceforge.net/software/product/Joomla/
5. https://sourceforge.net/projects/moodle/
6. https://sourceforge.net/projects/dolibarr/

Write Original Test Suite: To simplify the writing of test cases, we used the previously developed chrome plugin [7]. It is an extension to record the actions into a test case. We use this tool to create test cases for web applications. To create the original test suite, we design criteria for creating test cases: (i) all the test cases start from the index web page of the web application (e.g., $goto$("$http$://$localhost$:8888/$Joomla$_3_6_0")), (ii) test cases need to cover the main functionalities and events of a web app, (iii) test cases cannot be duplicated and should be unique.

Repair Test Suite: As mentioned in Section 4.2.2, TSG is a 5-tuple <S, U, A, L, C> architecture. We run the original test suite on two releases of each web application to build TSGs. During execution, WTSR can get the URL of each action and crawl candidates after each action. And WTSR saves these two TSGs to MongoDB. WTSR then compares these two TSGs by comparing their test cases. For damaged test cases, WTSR tries to fix them by finding a substitute that can replace the broken action in the sorted candidates. After that, we can get the evolved test suite for R2, including repaired test cases.

Check Efficiency: During the evolution of a test suite, WTSR records how much time it takes to build TSGs and how long it needs to repair broken tests. And we analyze data about test suite evolution, such as how many test cases are successfully repaired. We then use the same subjects and same test suites to perform on Water [Choudhary et al., 2011], comparing the repair time and repair number with our approach WTSR.

4.3.3 Results

After getting the results of this implementation, we then conduct a qualitative analysis in this subsection.

Table 4.2 – Test Suite Graphs of Web Applications

WebApp	Re	Suite(B/T)	URL(ge/up)	Action(ge/up)	Link(ge/up)	CA	Time(s)
Joomla	R1	96	50	214	205	94.8	986
	R2	38/96	28/51	142/209	145/211	97.73	845
Moodle	R1	39	27	72	77	45.47	393
	R2	36/39	5/29	10/73	7/80	35.86	201
Dolibarr	R1	47	57	130	131	29.58	423
	R2	27/47	35/55	63/132	63/134	29.57	352

For RQ1: WTSR generates two TSGs for three real web apps, and the data of TSGs are shown in Table 4.2. The first column presents the name of web applications, such as Joomla, Moodle, and Dolibarr. The second column is the releases (Abbreviated as Re) of each web app subject. From the third column to the seventh column is the <S, U, A, L, C>.

7. https://github.com/webautotester/wat/tree/master/chrome_plugin

The third column is the test suite, which contains the number of test cases, including the number of broken test cases and the total number of test cases. *B* represents the number of *Broken* test cases, and *T* represents the *Total* number of test cases in R2. For example, a total of 96 test cases have been written for Joomla R1. However, when executed on R2, 38 test cases were broken. The fourth column is the number of URLs. The same URL has been excluded, which means that the same URL can only be counted once. For example, a total of 96 test cases for Joomla has 50 different URLs during performed on R1. However, WTSR only *generates* (abbreviated as ge in Table 4.2) 28 URLs when performed on Joomla R2 because some URLs can not be recorded due to broken cases. When comparing TSGs, WTSR can explore more URLs by repairing broken tests. After the *update* (abbreviated as up in Table 4.2), there are 51 URLs of Joomla R2. The fifth column is the number of actions, and the sixth column is the number of links. The seventh column is the average number of candidate actions (CA) after each action. And the last column is the total cost time to generate TSG for each web app release. For instance, WTSR takes 986 seconds to create TSG for Joomla R1, including the navigation time between web pages.

The numbers of breakage for each app are different. As shown in Table 4.2, Joomla has 38 broken test cases over a total of 96 tests and the corresponding breaking percentage over the total number of tests is 39.58%. Moodle has a total of 39 test cases created from R1, and there are 36 breaks when they are running on R2. The breaking radio of Moodle is 92.31%. Dolibarr has 27 breaks over a total of 47 test cases and the percentage of test breakage is 57.45%. The test case was interrupted due to changes in the web application evolution, Hammoudi [Hammoudi et al., 2016b] has detailed the reasons for these breaks. The breakages show that not all the test cases are still usable for the next release when web applications evolve. This illustrates that test cases are fragile and not robust to overcome the problem of web application evolution. So it is necessary for testers to find an approach like our framework WTSR to overcome the evolution issue of test cases. Furthermore, the time cost of generating TSG is related to the number of test cases. More precisely, the time cost of generating TSG is related to the number of actions in test cases. As Table 4.2 depicts, the time has a linear relationship with the number of total actions. The more actions it runs, the more time it takes.

Table 4.3 – Web Application Test Suite Repair Results

	WTSR				Water			
WebApp	#(R/B)	ReRatio	Time(s)	AT(s)	#(R/B)	ReRatio	Time(s)	AT(s)
Joomla	33/38	86.84%	1182	35.81	25/38	65.79%	1073	42.92
Moodle	28/36	77.78%	674	24.07	19/36	52.78%	721	37.95
Dolibarr	22/27	81.48%	896	40.73	16/27	59.26%	834	52.13
Total	83/101	82.17%	2752	33.16	60/101	59.41%	2628	43.80

For RQ2 and RQ3: After comparing test cases and candidates in TSGs, WTSR evolves

the test suite by repairing broken test cases. We count the number of repaired tests and record the execution time for test suite repair. Table 4.3 indicates test suite repair results, including the repaired number of test cases, execution time, and repair ratios. The first column is the web app subjects. The second column shows the number (#) of test cases repaired (R) by WTSR from the broken (B) tests. The repair ratio (ReRatio) of each web app using WTSR is in the third column. The fourth column presents the total execution time for repairing broken test cases. And the fifth column is the average time (AT) to repair one test case. Table 4.3 also illustrates the running result of Water from column 6 to 9.

The repair time is related to the number of test cases that need to be repaired. When the number of these cases is larger, the execution time will be longer. For the repair ratio of each web app, the second column in Table 4.3 illustrated details. Joomla's repair ratio of broken tests is 86.84%. For web app Moodle, the ratio of repairing tests is 77.78%. And Dolibarr's percentage of test repair is 81.48%. For these three web subjects, WTSR has repaired 83 cases of a total of 101 broken tests and the corresponding repair ratio is 82.17%. Water can repair 60 cases of 101 broken tests, which repair ratio is 59.41%. Water's execution time is less than WTSR, but the difference is small. Therefore, building TSGs is useful for the repair of test cases. Our approach can help testers repair broken tests effectively.

Table 4.4 – Number of Broken Actions of Each Test Case

WebApp	one	two	three	four	five	six
Joomla	28	8	2	0	0	0
Moodle	11	13	5	6	0	1
Dolibarr	18	5	1	1	1	1
Total	57	26	8	7	1	2

Each test case may be interrupted in multiple places. Table 4.4 shows the number of broken actions in each test case. The second column means the number of test cases that has one broken action. The third column is the number of test cases that have two broken actions. Take Joomla as an example, 28 test cases have one interrupted action in each test case. By observing the data in each row, the number of test cases is inversely proportional to the number of interrupts. With the number of broken actions in each test case increase, the number of test cases decreases.

Because there is sometimes more than one interruption in a test case, WTSR executes in a loop to fix more broken actions in the test case. Fig. 4.5 presents the repair ratio of the test case with a different number of interrupts. The horizontal axis means the number of broken actions in each test case. And the vertical axis is the repair ratio. For example, Joomla has 28 test cases with one interrupt, 27 of which have been successfully repaired, and the repair ratio is 96.42% for the test case with one breakage. Fig. 4.5 indicates that the repair ratio is inversely proportional to the number of broken actions. With the number of

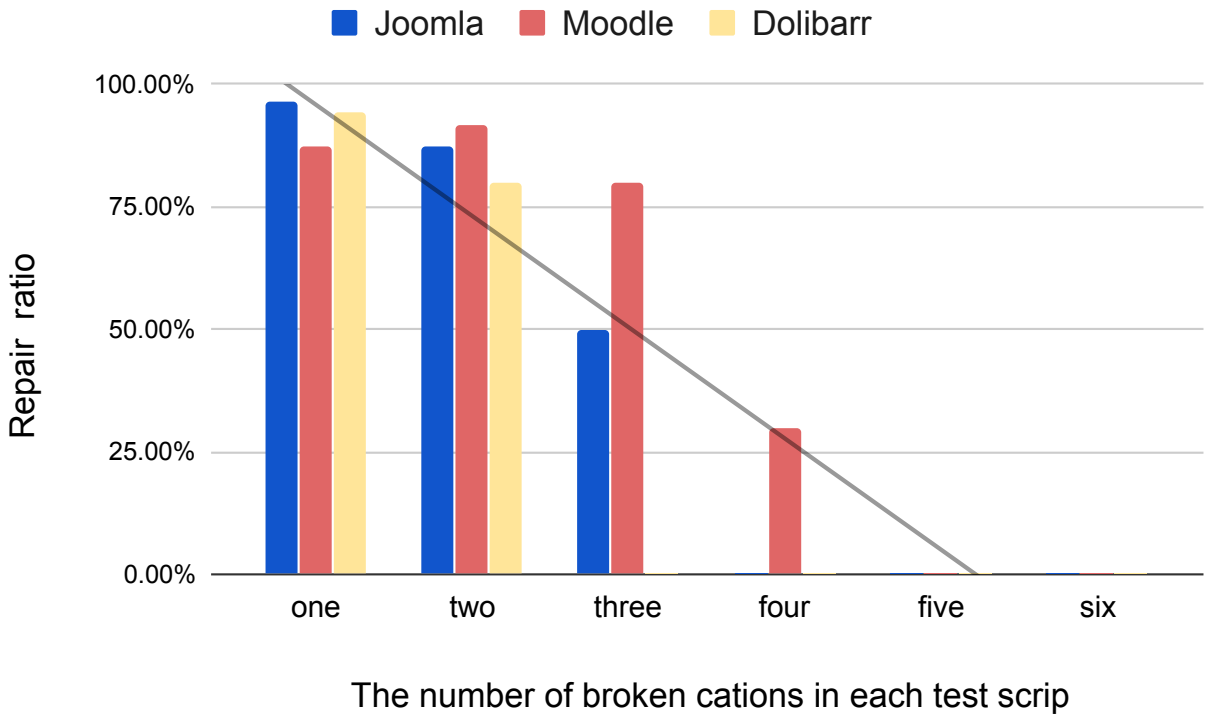


Figure 4.5 – The repair ratio of test case with multiple breaks.

broken actions in each test case increases, the repair ratio will decrease. The test case with more than five broken actions can not be repaired anymore.

4.3.4 Threats to Validity

In this subsection, we discuss threats to the validity of our framework that need to be cautious about. One threat to validity is that the number of web applications and test cases is limited in the experiment. However, it is enough to prove the effectiveness of our approach in experiments, which can apply to general web test repair. Moreover, the test executions may be affected by the changes of external third-party components because some libraries are called in approach WTSR. Furthermore, because of the robot restriction of the website, the case may not be executed due to it that our approach will be forbidden when requesting too many times on this website.

4.4 Conclusion

With the development of web applications, some damaged test cases need to be repaired. So test cases need to evolve according to the evolution of web applications. For this reason, we proposed a novel framework WTSR that efficiently evolve test cases by repairing broken tests. Our experiments show that: (1) the cost of time spent to build TSGs is related to the number of actions, and they have a positive linear correlation; (2) building TSGs is useful for the evolution of test cases; (3) the key issue in the evolution of web testing is to repair broken tests; (4) the more actions break in a test case, the harder it is to repair this case; (5) WTSR is effective to repair test case for web apps. Therefore, WTSR can evolve test cases corresponding to the evolution of web applications. We believe that our approach can apply to general web test evolution, which is helpful to developers to repair test cases.

In our future work, we plan to: (1) Collect user feedbacks on our technique to make our approach better for web test repair; (2) Experiment our approach to the web applications with bugs to study how do bugs affect the usefulness of this test repair approach; (3) Do a case study of all the web test repair tools by comparing their efficiency; (4) Update our intelligent algorithm to continuously improve the efficiency of test suites repair and try to deal with value-caused breakages.

CHAPTER

5 Conclusion

Given the recent advances in the field of web engineering, test automation with test cases has become the leading solution for testing web applications. Generally, many researchers have proposed approaches to create web test cases for these solutions, and these approaches are provided to many customers in the form of dedicated web test tools. However, some of the generated web test cases break due to the evolution of web applications. Although it is possible to perform the repair of this broken test case manually, it can be a tedious and error-prone task to complete. Some researchers have proposed methods to repair the broken tests, but it is still a challenging task to complete. Effectively repairing damaged test cases is not a simple task, especially when considering many different types of breakages. In this concluding chapter, we summarize our contributions in web test cases and present some interesting research perspectives.

5.1 Summary of contributions

In this thesis, we conducted a systematic mapping study on web test cases to find out the gap between the required techniques and the existing methods that are introduced in the literature. Furthermore, we propose a method to automatically repair broken test cases by using a test model of the web application. We then provide the contribution details in this thesis.

As presented in Chapter 3, our first contribution aims to systematically identify, analyze, and classify the publications and provide an overview of the trends in the field of web test cases. We conducted a systematic mapping study to form knowledge views related to web test cases by reviewing existing articles. We first define inclusion and exclusion criteria to select articles from the publications. After selecting, our study includes 76 articles published in the field of web test cases between 2000 and 2019. Second, we systematically

develop and refine classification schemes to extract data from selected articles. Based on these data, we can build the entire picture in the field of web test cases. Then, we derive the observed trends, such as test case generation, test case repair, or test case dependency.

Our second contribution, presented in Chapter 4, aims at identifying and repair the broken actions of web test cases from the end-points of view. To achieve this goal, it needs a solution to quickly identify damaged actions and provide a way to find replacements for these damaged cations in test cases. For this purpose, we present an approach called WTSR to generate TestSuiteGraphs, which can be used to repair broken tests. Our WTSR approach first creates TestSuiteGraph 1 by running test cases on web release 1. It then runs these test cases on web release 2 to build TestSuiteGraph 2. By comparing these two kinds of TestSuiteGraphs, WTSR identifies the substitutes of broken actions and repair the broken test cases by replacing broken actions. This contribution has been published in the *International Conference on Web Engineering 2021 (Accepted)*.

5.2 Perspectives

As demonstrated throughout this dissertation, the web test case of test automation is a complex domain of research, requiring expertise in several fields such as software engineering, web engineering, web testing, and test automation. Also, the application of these areas in an industrial environment brings its challenges, which leaves room for a lot of interesting research axes worth investigating. We propose several perspectives of web test cases in this section.

5.2.1 The dependence of web test cases

In Chapter 3, we conducted a systematic mapping study on web test cases to create a knowledge view using end-to-end web testing, including the activities of web test cases such as generation, maintenance, repair, and dependency. Based on this mapping study, one of the interesting research axes is the dependence of web test cases. Now only a few researchers try to detect test dependencies present in E2E web test suites. Due to the heterogeneous and multi-layered nature of modern web applications, E2E web test suites are prone to test dependencies, which makes it difficult for developers to create isolated test cases. And it is worth investigating using the test dependency technique to generate or repair E2E web test cases.

5.2.2 Refining test repair strategies in WTSR using machine learning techniques

In Chapter 4, we presented an approach to efficiently repair broken web test cases based on test models, aiming to improve the efficiency of test automation as much as pos-

sible. Since most repair work is performed at a given frequency, it is possible to gradually estimate the substitute of a broken action for a given test case, thereby improving the strategy for identifying substitutes over time. Therefore, we envision an improved strategy to identify the replacement of broken actions in the test case. Using machine-learning techniques, a runtime profile can be established for each repair, which can be used to improve repair accuracy.

APPENDIX

A Résumé en Français

Les développeurs s'appuient de plus en plus sur les tests End To End (E2E) pour tester les applications Web qu'ils développent et pour vérifier qu'ils n'ont pas de bogue du point de vue de l'utilisateur final. Un test E2E simule les actions effectuées par l'utilisateur avec son navigateur et vérifie que l'application Web renvoie les sorties attendues. Il considère qu'une application Web est une boîte noire, et ne sait que quelles sont les actions de l'utilisateur et quelles sont leurs sorties attendues. Toutefois, une fois que certaines évolutions sont effectuées sur une application Web, les actions de l'utilisateur peuvent changer (déplacer le bouton vers un autre emplacement, ajouter un nouveau bouton ou supprimer un bouton). En conséquence, le test E2E doit évoluer avec l'évolution des applications Web, telles que la réparation du test cassé, ajouter le nouveau test, et supprimer le test obsolète. Mais il faut beaucoup de temps pour faire évoluer les tests E2E, en particulier pour les grandes applications web. En tant que tel, nous présentons une approche, nommée WebTestSuiteRepair (WTSR), pour aider les développeurs qui font face à cette situation. Dans cette thèse, WTSR vise à comparer les scripts de test graphique pour réparer les actions cassées, d'identifier de nouvelles actions, et de supprimer l'action obsolète, contribue donc à faire évoluer efficacement les tests E2E pour les applications Web automatiquement.

Le Chapitre 1 présente d'abord brièvement le contexte, nous présentons ensuite les problèmes et les défis liés à l'évolution des tests E2E dans le Chapitre 1 qui couvre l'introduction. Dans ce même Chapitre 1, trois questions de recherche sont identifiées et deux contributions principales sont résumées, notamment une étude cartographique systématique des tests d'applications Web et l'approche WTSR.

Dans cette thèse, nous visons à aider les développeurs à utiliser la réparation des cas de test endommagés des applications web. Nous essayons d'introduire les trois questions de recherche. RQ1: Y a-t-il une abstraction qui rassemble les cas de test pour les versions

n et n + 1, et qui peut être utilisée pour réparer les cas de test cassés? Pour répondre à cette question de recherche, nous exécutons les cas de test sur deux versions d'une application web. Plus spécifiquement, il exécute respectivement des cas de test originaux sur la version web n et la version web n + k pour générer des graphiques de suite de tests (modèles abstraits) pour ces deux versions.

Après avoir généré des modèles abstraits de suite de tests, il aurait la deuxième question RQ2: Est-il possible de réparer automatiquement et efficacement ces suites de tests pour les applications Web en comparant leurs modèles de suites de tests? Pour résoudre ce problème, nous proposons une nouvelle approche, Web Test Suite Repair (WTSR), pour réparer automatiquement les cas de test Web défectueux. En comparant ces deux modèles abstraits, il met à jour le modèle de la suite de tests de la version n + k et répare les tests cassés en même temps.

WTSR est développé pour l'évolution des suites de tests, nous voulons donc savoir s'il est efficace ou non. Ce questionnement nous amène à la troisième question de recherche que nous explorons dans cette thèse: RQ3: Quelle est l'efficacité de l'approche proposée pour l'évolution de la suite? Pour répondre à cette question, nous choisissons trois applications Web réelles pour une vérification empirique. Nous utilisons WTSR pour créer des modèles de suite de tests et pour réparer automatiquement les cas de test Web défectueux. Ensuite, nous calculons le nombre de cas de test réparés et leur temps d'exécution.

À la fin du Chapitre 1, nous résumons également les grandes lignes du reste de la thèse. Le reste de ce document est organisé comme suit. Nous présentons d'abord dans le Chapitre 2 le contexte dans le domaine des cas de test Web. Au Chapitre 3, nous effectuons une étude cartographique systématique sur des cas de test Web afin d'identifier les lacunes dans ce domaine pour les recherches futures. Ensuite, dans le Chapitre 4, nous présentons une approche pour identifier automatiquement les actions candidates pour les actions cassées afin de réparer les scripts de test des applications Web. Enfin, nous concluons au Chapitre 5 en résumant les contributions et les principales perspectives.

Le Chapitre 2 de cette thèse s'intitule Contexte et contient dix pages. Il couvre le contexte requis pour renforcer la compréhension des défis abordés par cette thèse dans le domaine des tests Web. Nous présentons principalement Application Web et Evolution et les techniques existantes de test d'application Web et de génération de cas de test. Plus précisément, ce chapitre, en deux parties principales, donne un aperçu des techniques de test Web. La première partie illustre un exemple d'évolution logicielle qui conduit à la rupture de tests. Il nous donne l'opportunité de présenter différentes techniques de test telles que les tests de bout en bout, en boîte noire et de régression. La deuxième partie se concentre sur la génération de cas de test en introduisant diverses techniques pour générer des cas de test Web telles que les techniques de capture-rejeu, d'exploration et de modèle. Nous vous expliquons de manière claire ces différentes techniques. Enfin, le concept de rupture de test est introduit avec une distinction claire entre rupture de test et erreur. Il peut représenter une bonne base pour les chercheurs travaillant sur les tests d'applications Web.

Dans ce Chapitre 2, nous présentons les concepts de test Web qui sont nécessaires pour comprendre le reste de cette thèse. Nous décrivons quelques définitions de base pour les applications Web et les tests E2E. Nous fournissons des informations générales sur les techniques de test Web. Nous expliquons comment générer automatiquement des cas de test pour les applications Web. Il met ensuite en évidence comment une évolution effectuée sur une application web peut provoquer l'échec d'un test E2E, en raison d'une action cassée.

Notre première contribution est présentée au Chapitre 3. Nous effectuons une étude cartographique systématique pour évaluer la littérature existante afin de trouver des lacunes dans ce domaine. Nous décrivons d'abord l'objectif de l'étude de cartographie systématique et les questions de recherche. Nous présentons la méthodologie de recherche sur la façon dont nous réalisons cette étude cartographique. Nous fournissons ensuite le schéma de classification que nous avons développé pour la suite de tests Web et le processus utilisé pour la construire. Nous présentons également les résultats de synthèse des données extraites des études sélectionnées et répondons aux questions de recherche. Nous discutons des résultats de l'étude cartographique et de leurs implications pour les chercheurs et les praticiens.

Tout au long du cycle d'évolution des tests Web, la suite de tests comporte de nombreux aspects différents, tels que la création, la prévention de la casse, la réparation, la dépendance et les métriques. Pour aider les chercheurs et les testeurs, nous identifions et résumons systématiquement la littérature existante de la suite de tests Web dans ce chapitre. L'objectif principal de cette étude cartographique systématique est d'analyser les études primaires sur Web Test Suite et de fournir une vue d'ensemble de la suite de tests Web. Il vise à étudier une compréhension globale de la suite de tests Web du point de vue des testeurs et des chercheurs dans le contexte de l'évolution des applications Web. Cela contribue à résumer l'ensemble des suites de tests Web dans le domaine des connaissances en développement logiciel. De plus, il recueille également des efforts directs pour la recherche future de suites de tests Web au cours de l'évolution du Web, pour identifier les problèmes existants des suites de tests Web et pour déterminer la tendance de recherche des suites de tests Web. Nous introduisons ensuite les questions de recherche associées en suivant les lignes directrices publiées dans la littérature. Cela comprend deux questions de recherche principales et 13 questions de recherche secondaires. Ensuite, nous concevons la méthodologie pour réaliser cette étude cartographique.

Dans le Chapitre 3, nous avons conçu le processus de cette méthodologie pour rechercher les articles dans des bases de données électroniques et filtrer les études dont nous avions besoin. Pour garantir que les résultats de la sélection des études sont impartiaux et objectifs, nous avons conçu les critères de sélection des études et le processus de sélection des études. Après la recherche d'études et la sélection d'études, nous obtenons les études pertinentes, qui sont les données de base de l'effet boule de neige. Nous utilisons ces études sélectionnées comme entrée et trouvons leurs références. Nous menons ensuite le processus de sélection des études pour filtrer les références par méta-

données, résumé et texte intégral. Si nous obtenons des études sélectionnées à partir des références, nous le faisons à nouveau en appliquant les études nouvellement sélectionnées en entrée. Ce processus itératif sera interrompu jusqu'à ce qu'il n'y ait plus d'articles sélectionnés. Ensuite, des boules de neige ont été présentées pour compléter les articles au cas où certaines études seraient manquantes lors de la recherche d'études. Après avoir obtenu toutes les études, nous avons extrait les données de ces études sélectionnées et synthétisé les données pour cette étude cartographique. Toutes les données utilisées sont présentées et discutées. Enfin, nous discutons des résultats et des conclusions de cette étude cartographique.

Notre deuxième contribution est présentée au Chapitre 4. Nous proposons une approche nommée WebTestSuiteRepair, qui comparera les TSG pour identifier les substituts aux actions interrompues, et donc que les développeurs peuvent utiliser cette approche pour réparer automatiquement les tests cassés. En détail, nous proposons une approche pour générer des graphiques de suites de tests d'applications web, ce qui permet de réparer les cas de test web. Nous présentons un algorithme pour réparer automatiquement la suite de tests DOM en comparant les TSG des applications web. Nous implémentons l'outil WTSR pour les testeurs ou les développeurs afin de réparer les cas de test web défectueux. Nous effectuons une évaluation empirique de notre approche pour réparer les tests cassés pour trois applications web réelles.

Dans le Chapitre 4, nous présentons un aperçu de WTSR. Nous supposons que les testeurs capturent les cas de test initiaux de la version 1 d'une application Web. Tout d'abord, WTSR essaie d'exécuter les tests E2E initiaux sur la version 1 pour créer le TSG. Par souci de clarté, le graphique de la version 1 est nommé TestSuiteGraphRelease1 (TSGR1). Le graphique de la version 2 est nommé TestSuiteGraphRelease2 (TSGR2). TSGR1 et TSGR2 sont tous deux des TSG, correspondant à des versions différentes. Deuxièmement, WTSR exécute ces cas de test initiaux sur la version 2 pour construire le TSGR2. Il crée le même type de graphe à 5 tuples via le même processus de TSGR1. Au cours de ce processus, il explorera les données des éléments HTML à partir de la page Web après chaque action dans le scénario de test pour créer les sélecteurs. Après avoir créé des sélecteurs CSS efficaces et robustes pour les éléments HTML, WTSR peut générer différents types d'actions. Ces actions candidates explorées font partie du TSG. Troisièmement, WTSR compare ces deux TSG pour réparer la suite de tests. Il répare les cas cassés en utilisant un substitut pour remplacer l'action cassée dans la version 2. Il définit les cas de test qui ne sont plus utiles comme état de suppression en trouvant les actions supprimées dans le TSGR2. En même temps, le TSGR2 est mis à jour pendant le processus de réparation.

Pour évaluer la faisabilité et l'efficacité de notre approche pour faire évoluer la suite de tests, nous choisissons trois sujets de recherche et présentons un ensemble d'expériences avec analyse quantitative. Nous choisissons trois applications Web Joomla, Moodle et Dolibarr. Ce sont de véritables applications web qui peuvent contribuer à évaluer les performances potentielles et l'efficacité de notre framework dans un environnement de test réel. Joomla est un système de gestion de contenu (système Web) qui permet aux util-

isateurs de créer des sites Web. Moodle est une plateforme d'apprentissage open source (plateforme web). Dolibarr est un logiciel web open source pour gérer l'activité d'une organisation (contacts, fournisseurs, commandes, etc.). Dans l'ensemble, il apparaît que le WTSR a de bonnes performances en termes de taux de réparation et de coût en temps. Ces résultats montrent notre approche pratique et efficace. Nous montrons également les bénéfices des tests cassés identifiés, notamment avec un taux de réparation de 77% à 86%. Enfin, nous discutons de certaines menaces à la validité liées à l'approche proposée et présentons honnêtement ses limites.

Enfin, le Chapitre 5 conclut cette thèse en résumant nos contributions, et en présentant plusieurs perspectives possibles pour élargir notre travail. Nous résumons les travaux de la thèse en rappelant la démarche suivie, les contributions obtenues, et en donnant quelques pistes de travaux futurs. Nous concluons la thèse en discutant de la manière dont les différentes contributions ont répondu aux défis initiaux. Nous discutons également d'un ensemble de perspectives qui ouvrent de nombreuses pistes de recherche intéressantes.

Bibliography

Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. A. (1993). Incremental regression testing. In *1993 Conference on Software Maintenance*, pages 348–357. IEEE. Cited page 2.

Ahmad, A., Leifler, O., and Sandahl, K. (2019). Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. *arXiv preprint arXiv:1906.00673*. Cited page 2.

Alegroth, E., Nass, M., and Olsson, H. H. (2013). Jautomate: A tool for system-and acceptance-test automation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 439–446. IEEE. Cited page 3.

Andrews, A., Azghandi, S., and Pilskalns, O. (2010a). Regression testing of web applications using fsmweb. In *Proceedings of the International Conference on Software Engineering and Applications*. Cited page 14.

Andrews, A. A., Offutt, J., and Alexander, R. T. (2005). Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4(3):326–345. Cited pages 16 and 45.

Andrews, A. A., Offutt, J., Dyreson, C., Mallery, C. J., Jerath, K., and Alexander, R. (2010b). Scalability issues with using fsmweb to test web applications. *Information and Software Technology*, 52(1):52–66. Cited page 14.

Artzi, S., Dolby, J., Jensen, S. H., Møller, A., and Tip, F. (2011). A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 571–580. Cited pages 33 and 42.

Azizi, M. and Do, H. (2018). A collaborative filtering recommender system for test case prioritization in web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1560–1567. Cited page 37.

Bajaj, K., Pattabiraman, K., and Mesbah, A. (2015). Synthesizing web element locators (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 331–341. IEEE. Cited pages 4 and 5.

Benedikt, M., Freire, J., and Godefroid, P. (2002). Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WW W'2002*. Citeseer. Not cited.

Berner, S., Weber, R., and Keller, R. K. (2005). Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579. Cited page 2.

Berners-Lee, T., Cailliau, R., Groff, J.-F., and Pollermann, B. (1992). World-wide web: the information universe. *Internet Research*. Cited page 2.

Biagiola, M., Ricca, F., and Tonella, P. (2017). Search based path and input data generation for web application testing. In *International Symposium on Search Based Software Engineering*, pages 18–32. Springer. Cited pages 15 and 38.

Biagiola, M., Stocco, A., Mesbah, A., Ricca, F., and Tonella, P. (2019a). 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*, pages 154–164. Cited pages 37 and 38.

Biagiola, M., Stocco, A., Ricca, F., and Tonella, P. (2019b). Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 142–153. Cited pages 34, 37, and 39.

Binder, R. V. (1996). Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252. Cited page 2.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583. Cited page 25.

Bruns, A., Kornstadt, A., and Wichmann, D. (2009). Web application tests with selenium. *IEEE software*, 26(5):88–91. Cited pages 14 and 15.

Chang, T.-H., Yeh, T., and Miller, R. C. (2010). Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1535–1544. Cited page 3.

Chapman, P. and Evans, D. (2011). Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 263–274. Cited page 13.

Chen, L., Ali Babar, M., and Zhang, H. (2010). Towards an evidence-based understanding of electronic data sources. Cited page 25.

Chen, T., Zhang, X.-s., Guo, S.-z., Li, H.-y., and Wu, Y. (2013). State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773. Cited pages 15 and 16.

Chen, Y.-F., Rosenblum, D. S., and Vo, K.-P. (1994). Testtube: A system for selective regression testing. In *Proceedings of 16th International Conference on Software Engineering*, pages 211–220. IEEE. Cited page 2.

Choudhary, S. R., Zhao, D., Versee, H., and Orso, A. (2011). Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29. Cited pages 3, 35, 37, 45, 61, and 62.

Christophe, L., Stevens, R., De Roover, C., and De Meuter, W. (2014). Prevalence and maintenance of automated functional tests for web applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 141–150. IEEE. Cited pages 3 and 43.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702. Cited page 30.

Daniel, B., Gvero, T., and Marinov, D. (2010). On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 207–218. Not cited.

Dobolyi, K., Soechting, E., and Weimer, W. (2011). Automating regression testing using web-based application similarities. *International journal on software tools for technology transfer*, 13(2):111–129. Not cited.

Elbaum, S., Karre, S., and Rothermel, G. (2003). Improving web application testing with user session data. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 49–59. IEEE. Not cited.

Fewster, M. and Graham, D. (1999). *Software test automation*. Addison-Wesley Reading. Cited page 2.

Gao, Z., Chen, Z., Zou, Y., and Memon, A. M. (2015a). Sitar: Gui test script repair. *Ieee transactions on software engineering*, 42(2):170–186. Cited page 52.

Gao, Z., Fang, C., and Memon, A. M. (2015b). Pushing the limits on automation in gui regression testing. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pages 565–575. IEEE. Cited pages 2 and 14.

Garousi, V., Mesbah, A., Betin-Can, A., and Mirshokraie, S. (2013). A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374–1396. Cited page 10.

Grechanik, M., Xie, Q., and Fu, C. (2009). Maintaining and evolving gui-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE. Cited page 3.

Grilo, A. M., Paiva, A. C., and Faria, J. P. (2010). Reverse engineering of gui models for testing. In *5th Iberian Conference on Information Systems and Technologies*, pages 1–6. IEEE. Cited page 16.

Guarnieri, M., Tsankov, P., Buchs, T., Torabi Dashti, M., and Basin, D. (2017). Test execution checkpointing for web applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 203–214. ACM. Not cited.

Halfond, W. G. and Orso, A. (2007). Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154. Cited page 13.

Hammoudi, M., Rothermel, G., and Stocco, A. (2016a). 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. Cited pages 3 and 35.

Hammoudi, M., Rothermel, G., and Tonella, P. (2016b). Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE. Cited pages 3, 5, 17, 18, and 63.

Hanna, M., Aboutabl, A. E., and Mostafa, M.-S. M. (2018). Automated software testing framework for web applications. *International Journal of Applied Engineering Research*, 13(11):9758–9767. Cited page 35.

Heil, S., Bakaev, M., and Gaedke, M. (2016). Measuring and ensuring similarity of user interfaces: The impact of web layout. In *International Conference on Web Information Systems Engineering*. Cited page 57.

Imtiaz, J., Sherin, S., Khan, M. U., and Iqbal, M. Z. (2019). A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology*, 113:1–19. Cited page 18.

Jacob, P. M. and Prasanna, M. (2016). A comparative analysis on black box testing strategies. In *2016 International Conference on Information Science (ICIS)*, pages 1–6. IEEE. Not cited.

Jan, S. R., Shah, S. T. U., Johar, Z. U., Shah, Y., and Khan, F. (2016). An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN*, pages 2395–1990. Cited page 13.

Jin, W., Orso, A., and Xie, T. (2010). Automated behavioral regression testing. In *2010 Third international conference on software testing, verification and validation*, pages 137–146. IEEE. Cited page 14.

KAKARAPARTHY, D. (2017). Overview and analysis of automated testing tools: Ranorex, test complete, selenium. Cited page 14.

Keele, S. (2007). Guidelines for performing systematic literature reviews in software engineering. In *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn. Cited pages 22, 24, 25, 26, and 29.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. (2009). Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, 51(1):7–15. Cited pages 22 and 24.

Kung, D. C., Liu, C.-H., and Hsia, P. (2000). An object-oriented web test model for testing web applications. In *Proceedings First Asia-Pacific Conference on Quality Software*, pages 111–120. IEEE. Cited page 16.

Lemos, O. A. L., Silveira, F. F., Ferrari, F. C., and Garcia, A. (2018). The impact of software testing education on code reliability: An empirical assessment. *Journal of Systems and Software*, 137:497–511. Not cited.

Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2013). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281. IEEE. Cited page 14.

Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2014a). Visual vs. dom-based web locators: An empirical study. In *International Conference on Web Engineering*, pages 322–340. Springer. Cited page 5.

Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2016a). Approaches and tools for automated end-to-end web testing. In *Advances in Computers*, volume 101, pages 193–237. Elsevier. Cited pages 3, 11, and 17.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2014b). Reducing web test cases aging by means of robust xpath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 449–454. IEEE. Cited pages 3, 4, and 35.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2015a). Automated generation of visual web tests from dom-based web tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 775–782. Cited page 39.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2015b). Using multi-locators to increase the robustness of web test cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE. Cited pages 3, 4, 35, and 38.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2016b). Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204. Cited page 3.

Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., and Li, X. (2017). Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171. IEEE. Not cited.

Li, Y.-F., Das, P. K., and Dowe, D. L. (2014). Two decades of web application testing—a survey of recent advances. *Information Systems*, 43:20–54. Cited page 2.

Maciel, D., Paiva, A. C., and da Silva, A. R. (2019). From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. Cited page 16.

Mariani, L., Pezzè, M., Riganelli, O., and Santoro, M. (2014). Link: exploiting the web of data to generate test inputs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 373–384. Cited page 39.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156. Cited page 15.

Memon, A., Banerjee, I., and Nagarajan, A. (2003). Gui ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269. Citeseer. Cited pages 13 and 52.

Memon, A. M. (2007). An event-flow model of gui-based applications for testing. *Software testing, verification and reliability*, 17(3):137–157. Cited page 16.

Memon, A. M. (2008). Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):1–36. Not cited.

Memon, A. M. and Soffa, M. L. (2003). Regression testing of guis. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127. Cited page 3.

Memon, A. M., Soffa, M. L., and Pollack, M. E. (2001). Coverage criteria for gui testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267. Cited page 16.

Mesbah, A., Bozdag, E., and Van Deursen, A. (2008). Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE. Cited page 15.

Mesbah, A. and Prasad, M. R. (2011). Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570. Cited page 15.

Mesbah, A. and Van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *2009 IEEE 31st International Conference on Software Engineering*, pages 210–220. IEEE. Cited page 15.

Mesbah, A., Van Deursen, A., and Lenselink, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):1–30. Cited pages 15 and 16.

Mesbah, A., Van Deursen, A., and Roest, D. (2011). Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53. Cited page 16.

Meszaros, G. (2003). Agile regression testing using record & playback. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 353–360. Not cited.

Miao, Y. and Yang, X. (2010). An fsm based gui test automation model. In *2010 11th International Conference on Control Automation Robotics & Vision*, pages 120–126. IEEE. Cited page 16.

Mirshokraie, S. and Mesbah, A. (2012). Jsart: Javascript assertion-based regression testing. In *International Conference on Web Engineering*, pages 238–252. Springer. Cited page 15.

Mittal, S. and Mattela, V. (2019). A survey of techniques for improving efficiency of mobile web browsing. *Concurrency and Computation: Practice and Experience*, 31(15):e5126. Not cited.

Moreira, R. M., Paiva, A. C., Nabuco, M., and Memon, A. (2017). Pattern-based gui testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629. Cited page 14.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons. Cited page 2.

Nassiri, I., Masoudi-Nejad, A., Jalili, M., and Moeini, A. (2013). Normalized similarity index: An adjusted index to prioritize article citations. *Journal of Informetrics*, 7(1):91–98. Cited page 44.

Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, 21(1):65–105. Cited page 3.

Nidhra, S. and Dondeti, J. (2012). Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50. Cited page 13.

Orso, A. and Xie, T. (2008). Bert: Behavioral regression testing. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 36–42. Cited page 14.

Panichella, A., Kifetew, F. M., and Tonella, P. (2017). Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158. Not cited.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, pages 1–10. Cited pages 22, 23, 24, and 30.

Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18. Cited pages 22, 24, 25, 26, 29, and 30.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave macmillan. Cited page 13.

Ricca, F., Leotta, M., and Stocco, A. (2019). Three open problems in the context of e2e web testing and a vision: Neonate. In *Advances in Computers*, volume 113, pages 89–133. Elsevier. Cited pages 2 and 3.

Ricca, F. and Tonella, P. (2001). Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 25–34. IEEE. Cited pages 2, 11, and 45.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948. Cited page 3.

Silva, C. E. and Campos, J. C. (2013). Combining static and dynamic analysis for the reverse engineering of web applications. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 107–112. Cited page 15.

Sprenkle, S., Sampath, S., Gibson, E., Pollock, L., and Souter, A. (2005). An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 587–596. IEEE. Not cited.

Stocco, A., Leotta, M., Ricca, F., and Tonella, P. (2016). Clustering-aided page object generation for web testing. In *International Conference on Web Engineering*, pages 132–151. Springer. Cited pages 2 and 35.

Stocco, A., Leotta, M., Ricca, F., and Tonella, P. (2017). Apogen: automatic page object generator for web testing. *Software Quality Journal*, 25(3):1007–1039. Cited pages 33, 36, and 37.

Stocco, A., Yandrapally, R., and Mesbah, A. (2018). Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 503–514. Cited pages 3 and 35.

Tanida, H., Prasad, M. R., Rajan, S. P., and Fujita, M. (2011). Automated system testing of dynamic web applications. In *International Conference on Software and Data Technologies*, pages 181–196. Springer. Cited page 15.

Tonella, P., Ricca, F., and Marchetto, A. (2014). Recent advances in web testing. In *Advances in Computers*, volume 93, pages 1–51. Elsevier. Cited pages 2 and 10.

Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., and Su, Z. (2008). Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260. Cited page 35.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. ACM. Cited page 29.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media. Cited page 60.

Wong, W. E., Horgan, J. R., London, S., and Agrawal, H. (1997). A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE. Cited page 3.

Yakovlev, I. V. (2007). Web 2.0: Is it evolutionary or revolutionary? *IT Professional Magazine*, 9(6):43. Not cited.

Yandrapally, R., Thummalapenta, S., Sinha, S., and Chandra, S. (2014). Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 304–314. Cited page 3.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120. Cited pages 2 and 14.

Zeller, A. (2017). Search-based testing and system testing: a marriage in heaven. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 49–50. IEEE. Cited page 15.

List of Figures

List of Tables