AUTOMATED WEB TESTING: THE WAY HUMANS DO IT

MRIDHA MD NAFIS FUAD
Exam Roll: 111006
Session: 2021-22
Registration Number: 2016-215-049

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

AUTOMATED WEB TESTING: THE WAY HUMANS DO IT

MRIDHA MD NAFIS FUAD

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *my beloved parents*,
for the support, sacrifice, and blessings to protect me from all harm

Abstract

Web End-to-End (E2E) testing indicates the tester's interaction with the subject web Application Under Test (AUT) in such a way that validate real-user usage scenarios. Manually exercising all diverse usage scenarios can help testers identify failures earlier in the development life-cycle to reduce incurred cost of such impactful systems. However, this is not possible due to the frequent changes in code or requirements. In this situation testers resort to test automation scripts but these still require significant maintenance effort. Researchers have devised approaches ranging from aiding testers determine stable web element locators to automated testing techniques that reward coverage metrics. Yet, the industrial acceptance of such techniques usually fall short. This indicates the possibility of practitioner satisfaction with the inclusion of behavioral interaction patterns (that is, the tester's naturalness) in an automated testing technique.

To the best of knowledge, no dataset in literature represents how testers manually perform end-to-end web testing and further apply the extracted behavioral patterns in a fully automated end-to-end testing approach. This in turn would potentially improve the efficiency of failure detection. Previous research efforts do not consider the tester's motive to uncover failures and such work is yet unavailable for the web. From this motivation, this thesis firstly presents WebEV, a dataset encapsulating the tester's interactions. The WebEV initially filters open-source popular subject systems that rely on Cypress for test automation. These Cypress test specs are executed to extract the tester's interactions through a mock browser.

Moreover, snapshots of the application are used to retrieve - i) the current URL of the application, ii) the screenshot and HTML text of the entire page, and iii) the screenshot and HTML text of an operated UI element. This process is done both before and after each command execution to capture the perception of testers on each state transition, that is, extract their thought process during testing.

Taking prominent behavioral patterns, a fully automated model-free approach, eBAT, is proposed. This approach bridges the gap in automated testing by incorporating the tester's behavioral decisions taken during manual testing, such as non-redundant exploration and grouped action interaction. A behavior-driven action selection strategy is derived from these patterns to interact with the system. Actionable elements from the current state are detected, grouped based on functional relevance, and later filtered based on redundancy. The resulting actions are operated in a tree-based manner to exercise AUT usage scenarios. The effectiveness and efficiency of eBAT are evaluated as the unique number of failures detected and the detection rate respectively. Significant improvement in failure detection is reported comparing against the state-of-the-art, with similar code coverage. Moreover, eBAT outperforms the baseline failure detection rate in 5 out of 6 benchmark projects, within 5 out of 30 minutes of execution on average.

Acknowledgments

First and foremost, I would like to express my gratitude to Allah for granting me the opportunity to conduct this thesis. Surely Allah is the best planner.

Words cannot describe the impact of the motivation, support, effort, and guideline of my supervisor, Professor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka. Without his contribution, this thesis work would not have been the same, assumably worse.

Alongside my supervisor, the faculty and thesis committee members of the Institute of Information Technology, University of Dhaka deserve my utmost respect for their valuable feedback and criticism to improve my work.

A note of thanks is not enough for my classmates, research group members, and comrades of the Institute of Information Technology, University of Dhaka who have supported me directly or indirectly in this journey.

The devotion and sacrifice of my parents cannot be valued enough to put an expression to it.

Last but certainly not least, I am thankful to the Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No - 56.00.0000.052.33.002.22-42, dated: 12.06.2022. The scholarship due to which I was able to experience the IEEE/ACM International Conference on Software Engineering 2023 on-site in Melbourne, Australia.

List of Publications

1. "eBAT: An Efficient Automated Web Application Testing Approach Based on Tester's Behavior." in *Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 482-486, 2022.

2. "WebEV: A Dataset on the Behavior of Testers for Web Application End to End Testing" in *Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension (ICPC 2023)*, Co-located with ICSE 2023.

3. "Automated Web Testing Over the Last Decade: A Systematic Literature Review". In the *Systematic Literature Review and Meta-Analysis Journal* (Core Rank Unranked) (Accepted, feedback stage)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web application testing refers to the functionality validation process of a web Application Under Test (AUT) against specified requirements. In manual web application testing, a tester always tries to test a complete functionality at a time. They design test cases in such a way that fulfills each of the unique functionality. Their goal is to detect bugs with minimum redundant action execution [2] (such as clicking the same link multiple times is redundant). These common practices can be defined as the tester's behavior. Automated web testing approaches [3, 4] focus mostly on testing metrics like code, branch coverage or path diversity, etc., which may fail to explore the complete functionality (e.g., filling the inputs without submitting the form). As a result, these approaches may lead to partial but ineffective functional execution sequences.

An automated testing approach based on the tester's behavior may efficiently test a complete functional action sequence, combining the benefits of both manual and automated testing. However, determining such human-level behavioral patterns is challenging as manual testing practices may vary based on the tester or organization. A generic set of behavior needs to be designed such that it applies to any web application. These patterns might be obtained from examining manually written automation test scripts. Test scripts from the open-source community can

be selected as a baseline to remove any developer or organization-specific practices. The resulting set of behaviors incorporated into an automated testing approach may make the failure detection process efficient.

Unfortunately, no dataset can be found in literature that captures the tester's interactions during manual end-to-end web testing. Instead, research focus has been given to - automated testing techniques that reward coverage metrics [3, 4, 5], usage statistics based user behavior modeling [6], domain-specific language to generate tests [7], automatic page-object generation to encapsulate web pages [8], stable web locator generator for automation scripts [9], and so on. These efforts focus on either fully automated approaches that lack the tester's domain knowledge (maximizing code metrics) or improving automation script generation that is expensive to maintain [10]. Automated techniques in literature either require human domain knowledge for input or application model construction [4, 11]. Moreover, randomness in action selection policy in some approaches [12, 5] result in redundancy and inefficiency. The benefits from both styles of approaches may be combined to improve the state of automated web testing. That is, incorporating the tester's behavior to automatically execute human-like test cases.

This chapter first discusses the motivation of the research. Based on this motivation and gap discussed in literature, the research questions are formulated and a guideline is provided for answering these questions. After that, the contribution and achievement of this research are presented. Lastly, the organization of this thesis is mentioned for providing a guideline to the readers.

1.1 Motivation

Web applications are among the fastest-growing software type as they are universally accessible, requires no installation, and updates are enrolled automatically [13]. The advent of advanced web development framework enables rich and

dynamic interactions with web applications, making the system harder to test [14, 15]. This versatility of web applications is its strength and also its weakness as detecting failures late proves to be costly [16]. Moreover, manually interacting with the system under test to verify functional satisfaction based on requirements is labor-intensive and error-prone. Besides, manual testing is not only infeasible but impossible in modern-day rapid development styles such as Continuous Integration (CI) which is the norm [17]. For example, content delivery giants such as Netflix[1] and Facebook[2] integrate new releases 1-2 times daily [18], whereas IMVU[3] (a popular web chat application) is reported to release even 50 times per day [19]. Even automating the tester's browser interactions (using Selenium IDE, Cypress, etc.) in these scenarios is infeasible due to the cost incurred from maintaining these automation scripts. Even so, the testing process is challenged by deadlines and resource constraints (time, budget, and expertise).

The American National Institute of Standards and Technology (NIST) in 2002 reported that lack of software testing alone contributes to a negative impact of USD 62 billion per year [20]. Although the impact is different in other countries, this problem is indifferent. Only 32% of the software projects are successful, 44% failed, and 24% challenged according to the 2009 CHAOS report [21]. In this state, quality assurance is the key means to satisfy client expectations [22]. However, the software testing phase alone is reported to consume 80% of the total development cost [20]. Thus, reducing the cost of testing without compromising quality can benefit organizations and the respective country's economy overall. This benefit is not only specific to pioneers in software development but also to third-world countries like Bangladesh projecting revenue worth USD 5 billion from IT services by the year 2025 [23]. The role of quality assurance is unparalleled to attain this goal as the United Nations Committee for Development Policy (UNCDP) expects

[1]https://www.netflix.com/
[2]https://www.facebook.com/
[3]https://secure.imvu.com/

Bangladesh to become a Developing Country by 2026 [24].

One might think that browser automation frameworks and automated testing approaches in literature will suffice as an alternative to manual testing. But in reality, these techniques have some limitations that prevent proper industrial satisfaction. On one hand, test interaction browser automation either by Capture-Replay strategies or through programmable scripts incurs high maintenance costs. This is because web tests may break even with slight changes in functional behavior or UI. These changes force adding new test cases or modifying existing ones. On the other hand, automated web application testing approaches usually fail to mimic the interactions of a manual tester, that is, are unable to generate efficient interactions to completely test the diverse scenarios. For example, approaches might fill up some fields in a form and then navigate to another page without proper form submission. Here the interactions with the form become ineffective and complete functionality are not tested. This happens as approaches prioritize code coverage metric maximization instead of the tester's behavioral patterns.

Automated testing approaches are broadly categorized into model-based and model-free approaches. Model-based techniques [11, 4, 25] are one of the most popular types but are limited to generating test cases using a static navigation model. These test cases often require human domain knowledge for inputs or domain-specific description [4, 7]. Model-free approaches that select actions in a pseudo-random manner [12] are termed as random-based. They consider the dynamic nature of the Application Under Test (AUT) but generate redundant and invalid test cases. Zheng et al. [5] proposed a model-free reinforcement learning based fully automated approach to explore diverse application scenarios. However, the action selection policy accounts for much redundancy and randomness, which affects the test suite efficiency. Despite much advancements, no automated testing approach incorporates the tester's behavioral patterns to make the automated failure detection and test case generation process satisfactory.

1.2 Research Questions

This thesis considers the behavioral aspect of testers when they interact with any web application to validate functionalities. Section 1.1 demonstrates the inability of current approaches to satisfy practitioners as testers fail to relate to their interaction patterns. Furthermore, the generated test suite is considered ineffective and inefficient due to redundancy and partial functionality execution. Taking motivation from this situation, this research answers the following research question:

- **RQ:** What is the impact of the tester's behavioral patterns on automated web testing?

  Answering this research question requires the identification of the tester's behavioral patterns exhibited during manual testing. A mechanism needs to be devised to incorporate these behaviors in an automated web application testing approach. This research question will be answered by the following two sub-questions:

  - **SQ1:** *How effective is a behavior-driven automated web application testing approach in terms of failure detection?*

    Effectiveness in web testing literature is primarily measured from the number of detected unique failures and code coverage metrics. Although failure detection is more practical to attain the goal of testing and testers seldom use code metrics to measure the quality of testing. These metrics can be used to answer this question by comparing the proposed approach with the state-of-the-art. For sound evaluation, this comparison needs to be done on diverse benchmark projects to remove subjectivity in the application domain and framework used. Furthermore, the approaches need to be executed multiple times to eradicate the probability of "lucky" interactions. For example, randomness in exploration reveals failures in a specific deeply nested functionality but is

unable to do so in another trial. The deviation (uncertainty) in failure detection needs to be calculated. Multiple executions can enable the scope of employing statistical significance tests.

- **SQ2:** *How efficient is a behavior-driven automated testing approach in exploiting web applications?*

  To answer this question of efficiency, the rate of failure detection is measured to evaluate the practical application of the proposed approach. Given an infinite amount of time, automated approaches can theoretically explore the entire state space of the subject system. However, in the continuous style of development, resource allocation becomes more and more difficult. As such, automated testing approaches are practically useful when failures are detected as fast as possible. The proposed behavior-driven automated approach focuses on functionality completeness (like testers) rather than diversified exploration (maximizing code metric). The timestamp of each detected failure can be analyzed to generate the number of unique failures per minute. This can be done for each of the subject systems on multiple trials. The trials can be merged to compare the combined efficiencies. This question can evaluate the practical applicability between behavior-driven action selection and code metric maximization.

1.3 Contribution and Achievement

The contribution of this thesis is mainly twofold - i) develop a dataset incorporating manual tester's interaction patterns, and ii) propose an automated web testing approach combining manual tester's interaction patterns along with the advantage of automation. At first, a novel methodology to extract the tester's interactions from the automation script is proposed. These interactions capture

testing decisions by testers to exploit failures or complete functional exploration of the subject system. To the best of acquired knowledge, no such dataset exists that captures the tester's interaction patterns during end-to-end testing. Current automated web testing approaches in literature lack naturalness during interaction with AUT. This prevents widespread adoption among practitioners [10]. This thesis effort takes the tester's common interaction pattern and incorporates them in an automated testing approach. In a nutshell, the two major contributions of this thesis are given below:

- **Mining the tester's interaction patterns during end-to-end testing:** To the best of acquired knowledge, no research effort has been made to support practitioners on how diverse web scenarios are tested manually. Crowd-sourced user interactions have been stored in the case of Android applications and stored as a large-scale interactions dataset [26]. This dataset is used for a plethora of research contributions alongside automated testing approaches that preserve the naturalness of UI interactions. On the contrary, this thesis contributes to a dataset aimed at web application interactions made by testers. There is a significant difference between general user interaction and when testers aim to validate AUT functionalities. For this reason, the WebEV dataset stores usage patterns from automation scripts that represent how testers would manually perform web testing. The database contains the tester's interactions from popular 100 open-source projects that contain automation scripts. A novel method is used to extract the state snapshot of each action before and after the interaction. This stores the visual change in AUT during testing and diversity (corner cases) for each usage scenario.

  The representation of the test cases in WebEV aims to support a wide range of research applications. Furthermore, the method is generic enabling the extraction of interactions from projects outside the dataset. Incorporating more projects will increase the applicability of deep learning algorithms to

test like humans. The generic data and representation of WebEV motivate three major categories of use cases, as discussed in subsequent subsections. The methodology to generate the WebEV dataset is described in detail in Chapter 4. Our main contribution resides in devising a mechanism to store the tester's behavior from the automation script.

- **Naturalness in automated testing is more important than coverage metrics:** Current approaches in literature suffer from redundancy problems and fail to explore the complete functionality of the system [11]. This is due to the concentration of code coverage metrics rather than the naturalness of interactions. A model-free (independent of the static navigational representation of AUT) automated approach eBAT is proposed based on three major behaviors cross-checked from eBAT. These behaviors are hypothesized from executing state-of-the-art automated approaches. The behaviors are - i) non-cyclic navigation, ii) non-redundant action execution, and iii) proper form submission. The method is divided into state abstraction, redundancy reduction, behavioral grouping, and exploration completeness. Details on the individual steps can be found in Chapter 5.

  Similar to prior literature [4, 5], eBAT is evaluated on 6 open-source benchmark web projects. Comparison done with WebExplor [5] (the state-of-the-art) reveals the better performance of eBAT in terms of effectiveness in failure detection and detection efficiency. Statistical significance is determined in 5 out of 6 projects using Mann–Whitney U test. Furthermore, failures detected in these projects by the baseline are also detected by eBAT within only 5 minutes of execution (out of a total of 30 minutes). Detailed discussion on evaluation can be found in Chapter 5. Our main contribution lies in stating that naturalness in automated interaction is likely to be more effective than relying on coverage metrics.

1.4 Organization of the Thesis

The overview of the subsequent chapters is discussed in this section. Each chapter contributes towards a specific goal in the following way:

- **Chapter 2 Background Study:** This chapter creates the knowledge base for understanding automated end-to-end web testing. This chapter first describes the architecture of web applications and preliminary concepts such as - web state, test case, and coverage metric. Later on, provides web testing practices both manually and through automation. The classification of automation approaches is presented along with an automation tool demonstration. Next, automated testing is discussed along with its relevant concepts - state abstraction, navigation model, and exploration strategy. Lastly, this chapter ends by describing the importance and challenges of automated web testing approaches.

- **Chapter 3 Literature Review on Automated End-to-End Web Testing:** In this chapter, research contributions made on fully automated web testing is presented. Existing automated web testing approaches can be broadly classified into two categories namely model-based and model-free testing approaches. This classification is done based on the usage of a static navigation model to generate test cases. Here model-based techniques rely on a navigation model whereas model-free approaches do not. The trade-offs regarding the two styles of approaches are also discussed. The goal of this chapter is to present the limitation in the current literature.

- **Chapter 4 WebEV: A Dataset on the Behavior of Testers for Web Application End to End Testing:** This chapter presents the WebEV dataset containing interaction patterns of testers during end-to-end testing. Such a dataset is absent in literature utilizing which interactions of auto-

mated testing can be made natural. Existing automation scripts in open-source systems are extracted. For this, the selection criteria of automation tools and subject systems are discussed. Then, a novel method to extract the tester's usage scenario is described. Lastly, the use cases of such a dataset in the literature are listed.

- **Chapter 5 eBAT: An Efficient Automated Web Application Testing Approach Based on Tester's Behavior:** An automated testing approach eBAT is proposed in this chapter that interacts with the application under test based on the tester's behavior. These behaviors are cross-checked from WebEV's stored interactions. First, the approach derives abstracted states along with valid actions. Then the redundant actions are filtered. The process for exploration completeness through tree-based traversal is discussed next. Later, the process of evaluating eBAT against the state-of-the-art along with the results is reported. Finally, the threats to validity are marked out at the end of this chapter.

- **Chapter 6 Conclusion:** This chapter is a "look-back" to the entire thesis. That is, the summary of the thesis can be obtained along with the planned future work presented.

Chapter 2

Background Study

The web testing process refers to exercising the web application under diverse test scenarios to maximize finding and fixing failures. This, in turn, ensures the quality of the delivered Application Under Test (AUT). The quality factor of web applications is crucial for the quality and efficiency of various social, economic, educational, enterprise, and scientific bodies [27, 10]. The popularity of web applications is due to them being - i) installation cost-free, ii) automatically upgradable to all/ selective users, and iii) universally accessible from any device with internet, explain the reason for them being such dominant type of software [28]. So, investing time in testing becomes unavoidable and one of the main approaches to assure quality is end-to-end (E2E) testing where testers mock real user scenarios to validate system functionalities [27, 29]. This chapter primarily focuses on terminologies and concepts revolving around E2E testing such as - web application architecture, web state, test case, etc. Next, the different styles and work processes of testers in conducting E2E tests are discussed. These discussions later on converge to the importance of automated web testing approaches and their relevant concepts such as - navigation model, state abstraction, etc. Finally, the key challenges in automated web application testing are listed.

2.1 Web Application Testing Concepts

End-to-End (E2E) web application testing verifies the functionalities and/or performance of a web system to work based on requirements in a real usage scenario (that is, in production). The knowledge of some relevant concepts like web application architecture, web state, test case, and coverage metric is necessary before understanding the E2E testing process and how testers actually conduct it. These concepts as well as their relevance with E2E testing are presented in this section.

2.1.1 Web Application Architecture

Due to technological advancements, any electronic device with the internet can access web applications. This is possible as web applications are designed in a Client-Server architecture where the clients (user-end devices) request some service to server(s) that receive and fulfill those functionalities [30]. This is the de facto due to eased processing, resource sharing, and data replication reduction. The visual web pages accessed by the clients are rendered by the browser.

As shown in Figure 2.1, the represented data and corresponding HTML, CSS, and JS scripts of these visuals are sent by the server on request. Therefore, every functionality accessed by the clients is an amalgam of these basic components. This makes the way web applications work indifferent to the end-user's eyes irrespective of the underlying technology used. The server side is thus usually invisible to most stakeholders. In their point of view, the functional quality of the client side determines the quality of the overall web application. Testers imitate the same conception while conducting E2E testing. The tests are conducted by interacting with the User Interface (UI) to exercise client usage scenarios.

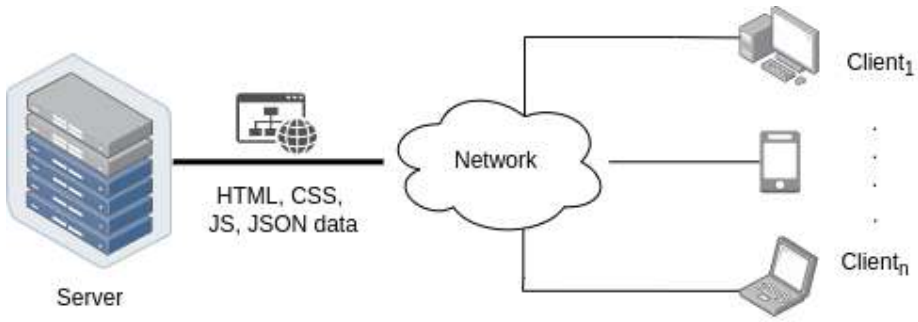


Figure 2.1: Client-Server Architecture for Web Applications

2.1.2 Web State

A number of steps need to be taken to accomplish a particular functionality, leading to visual transitions in the UI. The underlying representation of a web application (that is, the HTML document) at a particular time is its state. To the human eye, the screenshot images representing the HTML changes are the web state. To achieve the required functionality, each state contains a set of operable elements such as links, buttons, input fields, dropdowns, etc. The interaction with such valid operable elements is called actions in web testing. These interactions usually lead to a change in the URL, visuals, or functionality of the current web page [5]. To sum up, actions lead to state transition and are the basic unit of interaction during testing.

The set of actions defined is unique to each state and may be different in other states. For example, Figure 2.2a represents the initial state of a sample web application where clicking the *owners* dropdown button exposes new actions as seen in Figure 2.2b. These two states behaviorally expose different sets of action to interact with the subject system.

2.1.3 Test Case

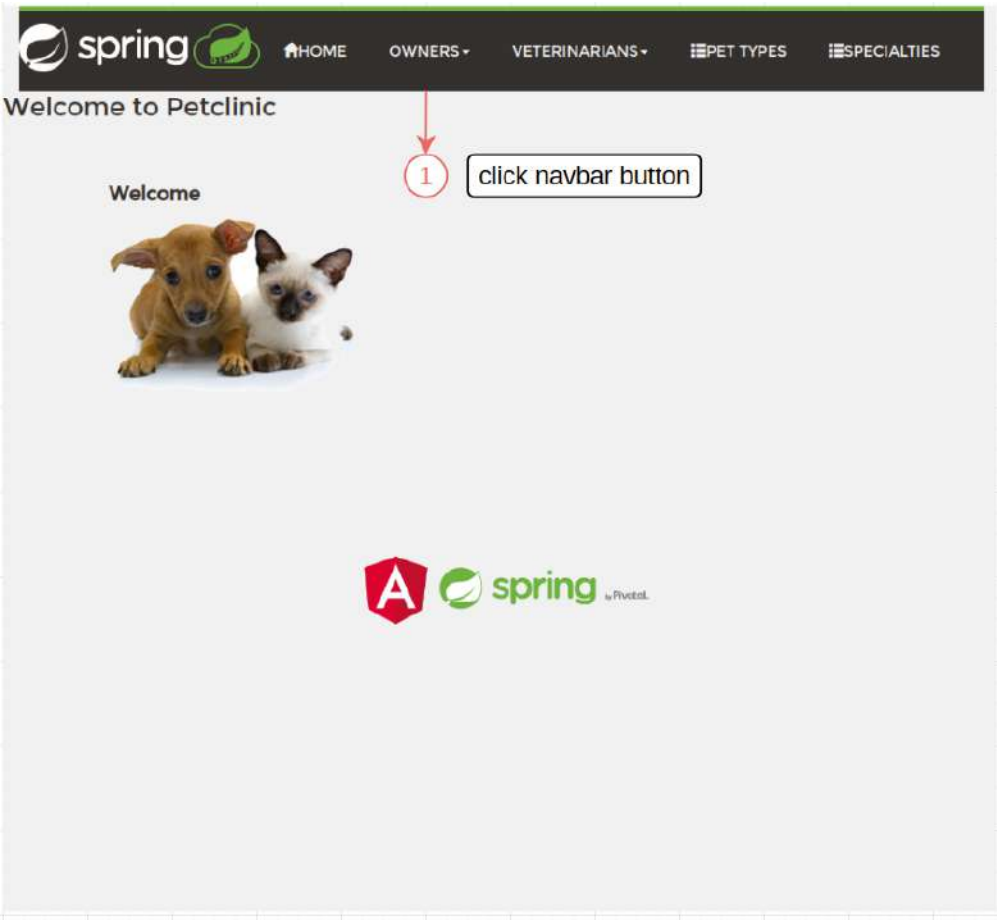
Formally, test cases are specifications stating inputs, execution conditions, and expected outputs [31], [32], developed for a particular objective such as exercising a specific functionality or detecting bugs. Simply put, test cases are sequences

of actions $(a_0, ..., a_n)$ with necessary input values to verify a predefined system behavior. Testers execute these test action sequences based on a predefined plan to obtain outputs. The obtained output is compared with the expected output based on requirements. The test case is said to fail if the actual and expected outputs do not match, indicating fix induction. Otherwise, if the assertion holds, the test case is considered passing and the AUT functionality is validated.

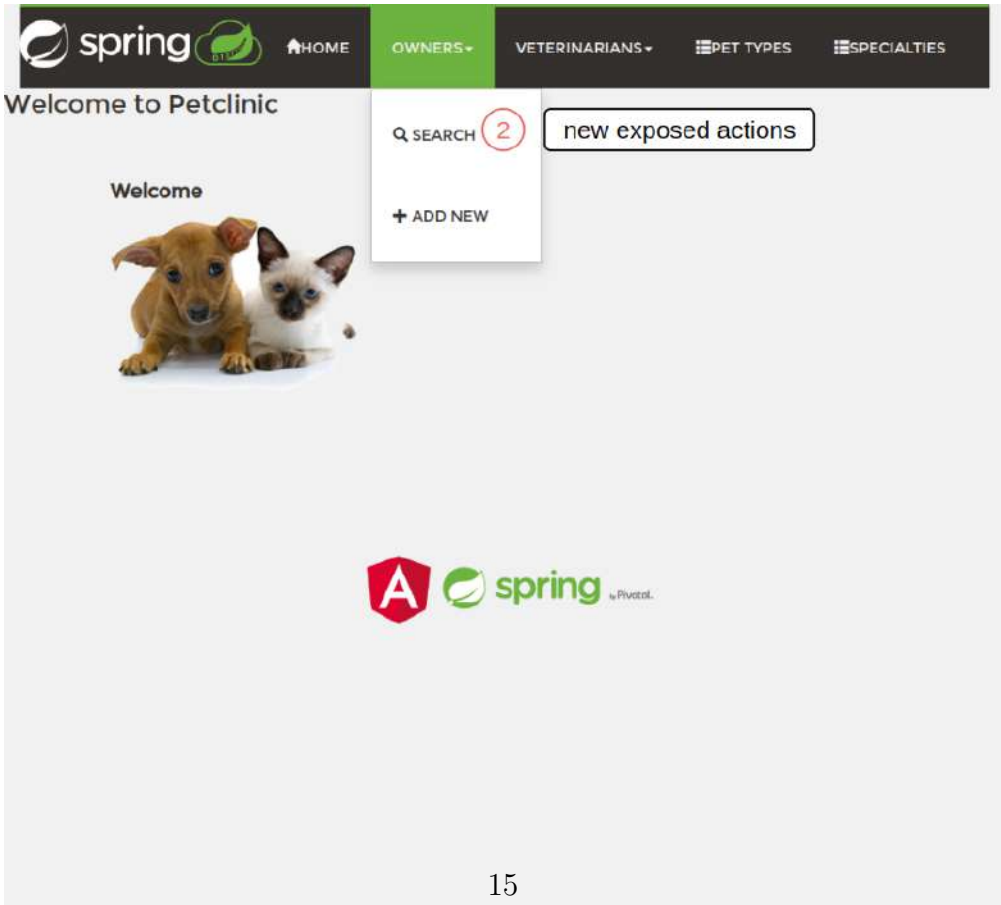
In the case of web application E2E testing, errors and exceptions from both client and server are reported in the browser due to JS runtime exception or response status code. These errors and exceptions can be collectively termed failures. For example, Figure 2.2 illustrates a simple test scenario where the tester validates the owner search functionality. Starting from the home page the tester would click at appropriate elements as in Figure 2.2a and Figure 2.2b. Afterwards, the owners are searched and compared with expected results as depicted in Figure 2.2c and Figure 2.2a respectively. Usually, this expected output is devised during the test planning phase, prior to exercising the test case. Therefore, a test case represents a specific pattern of interaction comparing the actual output against the excepted one based on requirements.

2.1.4 Coverage Metrics

In a practical setting, the amount of testing time possible to be allocated is very limited. Moreover, web interaction is time-consuming as browser interaction is slow. Hence testers need to exercise maximum usage scenario in minimum time. The efficiency of test cases executed (test suite) is usually measured in terms of AUT functionality coverage percentage. This unit of coverage after execution can be - i) code; percent of code lines executed, ii) branch; percentage of conditional execution paths exercised, or iii) state; percentage of unique states explored (similar to functional coverage). Higher coverage usually indicate better exercising of diverse AUT functionalities, enabling comparison between test suite effectiveness.

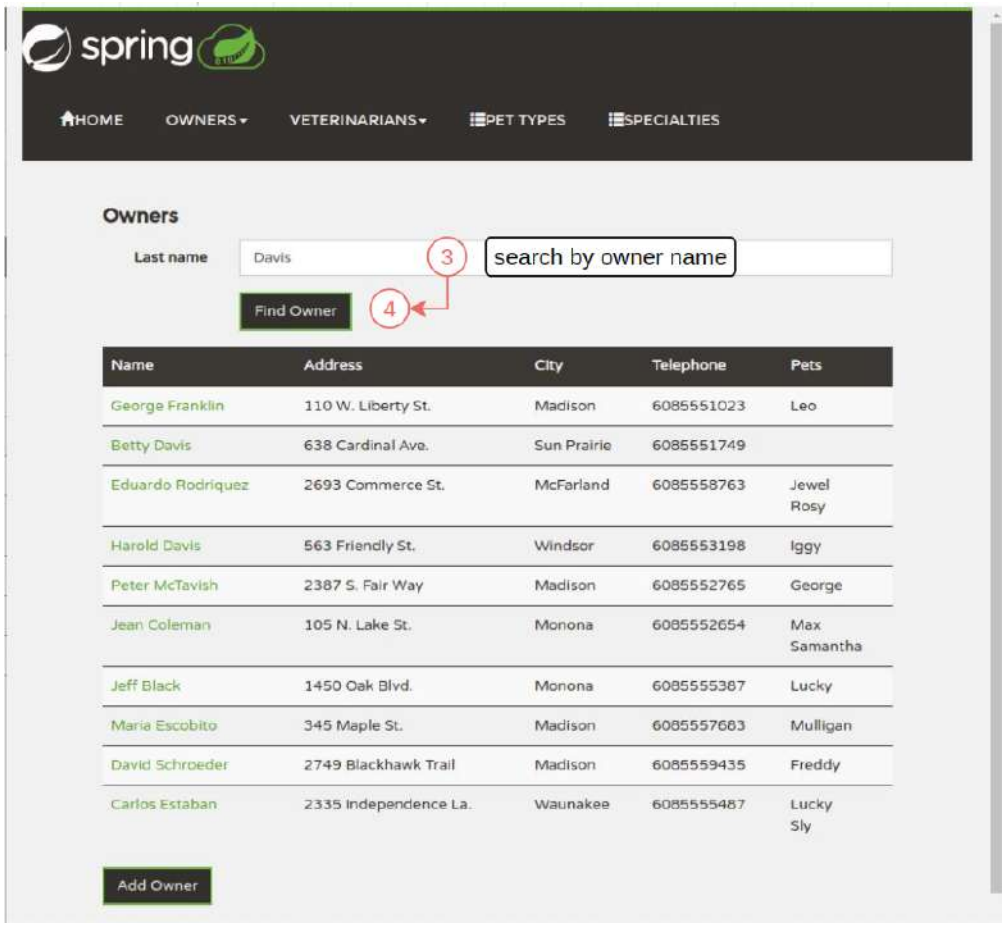


(a) Select navbar button to expand dropdown

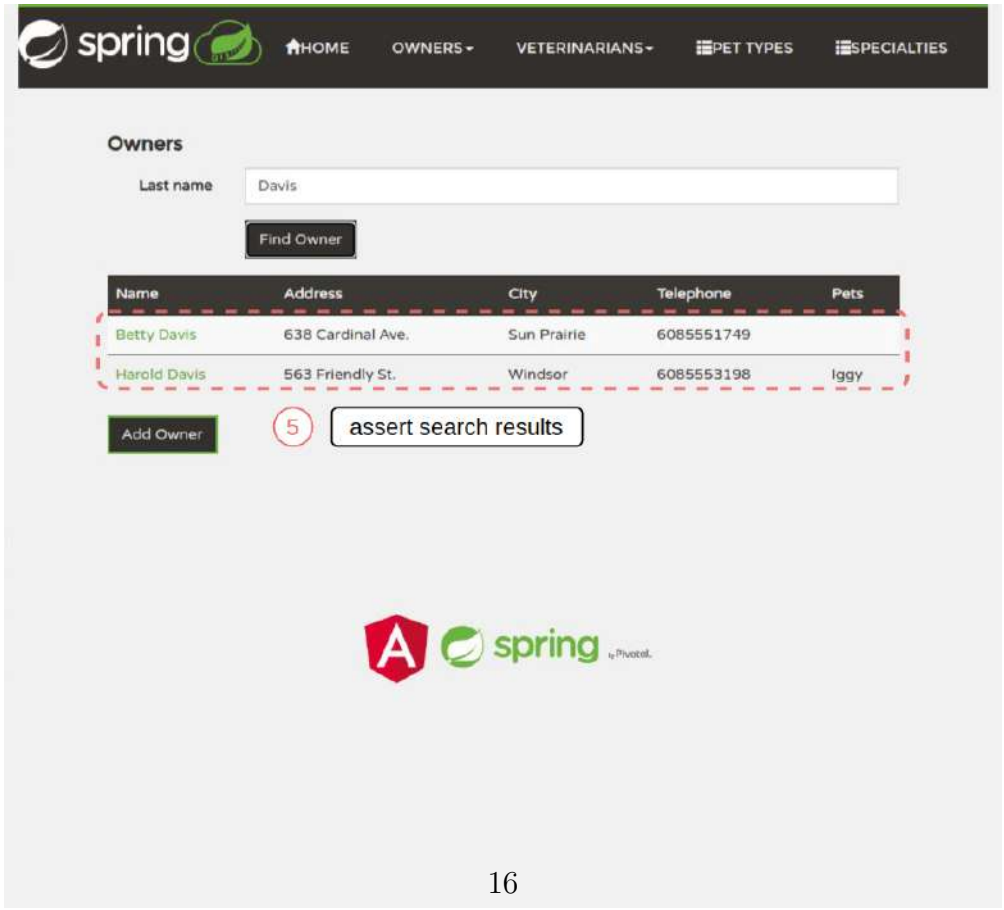


(b) Click *search-button* from newly exposed links

Figure 2.2: Example test case from Petclinic [1] web application



(c) Locate and type owner name input and click search button



(d) Match search results with expected users list

Figure 2.2: Example test case from Petclinic [1] web application (cont.)

2.2 Web Testing Practice

Manually interacting with the web application is the most simple and straightforward way to verify the web application's functionality against expectations. However, this approach requiring re-validation of the entire AUT on each change is time-consuming, labor-intensive, and error-prone. As such, most commonly, automation tools are used that require initial manual intervention to record test actions [4]. Afterward, regression testing can be done that automates the tester's interactions recorded previously. This section describes the tester's behavior in manual testing and approaches taken for automation.

2.2.1 Manual Testing Behavior

A tester exploits the web application by prioritizing diverse scenarios that do not overlap in functional goals [3, 4]. Tests involving cyclic or duplicated behaviors are avoided in manual web application testing, saving time consumption from unnecessary browser interaction. The Petclinic open-source project, as shown in Figure 2.3, is used for demonstration purposes. It contains deeply nested pages, resembling real-world web applications [33]. In this application, when testing the *add-owner* form (page 2 in Figure 2.3) starting from the *home* page, navigating back to *home* using the navbar link would result in an ineffective cyclic execution. Moreover, navigation from *home* -> *owner-detail* -> *add-owner* form will be inefficient as the targeted page is accessible directly from the *home* page via the navbar dropdown link. Therefore, avoiding such navigation will not have an impact on exploration rather result in inefficiency due to ineffective execution.

Proper testing must exploit the functionalities of the system using both expected and unexpected sequences of interaction to maximize failure discovery. However, manual testing is often restricted to verifying the expected behavior only [34]. It is expected that a user would provide input in all fields to submit

the form. This expected behavior needs to be tested along with the unexpected scenario where partial or no fields are filled [35]. Moreover, there may be business-specific logic associated with the sequence in which inputs are given, e.g., input fields can be exposed only after a dropdown is selected. While interacting with the form, focusing on other components (such as navbar links or table rows) will also test an unexpected scenario, resulting in an ineffective test case. Therefore, the success of testing depends on the efficient exploitation of diverse system scenarios.

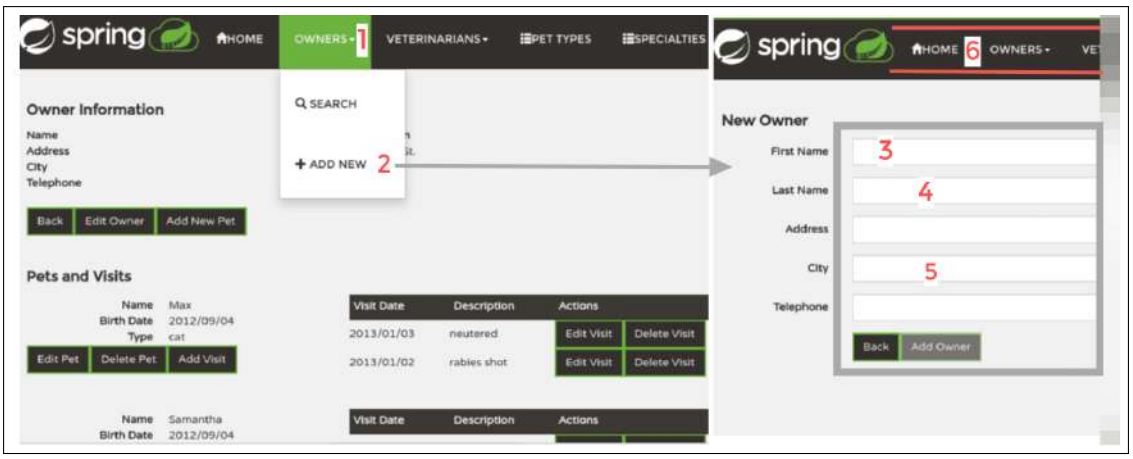


Figure 2.3: Petclinic sample project (ineffective execution scenario *owner-detail* -> *add-owner* illustrated by actions 1-6)

2.2.2 Approaches to E2E Testing Automation

Frequent source code changes in modern continuous integration style development inflict re-validating the AUT functionalities repeatedly. Manually exercising the same test scenarios is often neglected due to its monotonous and time-consuming nature. Researchers and practitioners have contributed to broadly two categories of automation approaches, namely - i) Capture-Replay based, and ii) Programmable-based test automation. These approaches along with their advantages and disadvantages are elaborated in this subsection. Alongside, an E2E web testing automation tool is demonstrated to make readers familiar with the process.

2.2.2.1 Capture-Replay Based Testing Approach

Capture-Replay (C&R) based testing tools work by recording the tester's interaction sessions and then rerunning them automatically without further human intervention. These tools record events such as navigating to different web pages, clicks and key presses on web elements, etc., and store information in a script. Lastly, some assertions are added to the scripts to ensure expected functionality. These scripts are executed to conduct automatic interaction each time (also called regression testing) and validate system functionality. C&R tools broadly use two types of techniques to track operated elements - i) DOM based, and ii) Visual-based. Document Object Model (DOM) based tools track the subject web element using one of its locators such as ID, name, CSS selector, XPath, etc. On the other hand, Visual based approaches are able to track visual locators of elements such as element snapshot or coordinate.

The main advantage of the Capture-Replay based approach is that the initial task of script generation is simple and does not require any programming knowledge [10]. However, the drawbacks are quite noticeable too. The test scripts are tightly coupled with the corresponding web page, that is, even slight changes in the web page may cause tests to break and require change [9, 8]. Moreover, the values and assertions are often hard-coded and need a change in case of the evolution of the subject system. Additionally, the generated scripts contain a high amount of duplicated code. For instance, duplicate methods and code is generated to test different corner cases of the same login page.

2.2.2.2 Programmable Testing Approach

Programmable web testing solves the rigidly coupled test script problem by providing flexibility through manual script creation. These scripts can be written with the help of frameworks, specialized languages, or general languages with library support to mock the browser behavior. Such libraries provide a user-friendly Ap-

plication Programming Interface (API) to simulate button clicks, input, assertions, etc. Programmable testing reaps benefits from condition-dependent scenarios and data-driven tests (reusing the same code with different parameter/input values) in spite of requiring more initial effort. Furthermore with the introduction of Page Object pattern [36, 37, 8], now web page functionalities can be semi-automatically encapsulated into a series of objects and methods (for example, using Selenium IDE). This promotes *Separation of Concerns* design principle to minimize coupling and make maintenance easier [38, 39]. This approach is more suitable to handle the complex nature of modern web applications.

2.2.2.3 Cypress

Cypress is selected as the demonstration tool as it incorporates both C&R and programmable types of web automation. Moreover, Cypress is the most popular E2E web testing tool based on GitHub Stars. Other alternatives in this category are Selenium WebDriver and Playwright. Cypress library when installed via npm is preloaded with assertion, mocking, and stubbing support to imitate manual testing. Multiple test cases are grouped as specs in Cypress. The specs serve as a grouping means of test cases exercising similar usage scenarios. Whereas each test case defines a series of commands (web actions) that simulate real-user interaction. Running a spec file means running its test cases, and each test case executes the commands. Assertions can be made at any point of test execution.

Listing 2.1 shows an example of a spec file taken from cypress-real-world [1] open source app. Here on line 2, the *beforeEach* function triggers prior to executing each test case to reset the web application state, database, cache, etc.

[1] https://github.com/cypress-io/cypress-realworld-app

```
describe("User Sign-up and Login", function () {
 beforeEach(function () {
   cy.task("db:seed");
 });

 it("should redirect unauthenticated user to signin page", function () {
   cy.visit("/personal");
   cy.location("pathname").should("equal", "/signin");
   cy.visualSnapshot("Redirect to SignIn");
 });

 it("should redirect to the home page after login", function () {
   cy.database("find", "users").then((user: User) => {
     cy.login(user.username, "s3cret", { rememberUser: true });
   });
   cy.location("pathname").should("equal", "/");
 });

 it("should error for an invalid password for existing user", function () {
   cy.database("find", "users").then((user: User) => {
     cy.login(user.username, "INVALID");
   });

   cy.getBySel("signin-error")
     .should("be.visible")
     .and("have.text", "Username or password is invalid");
   cy.visualSnapshot("Sign In, Invalid Username, Username or Password is
   Invalid");
 });
}
```

Listing 2.1: An example Cypress spec file

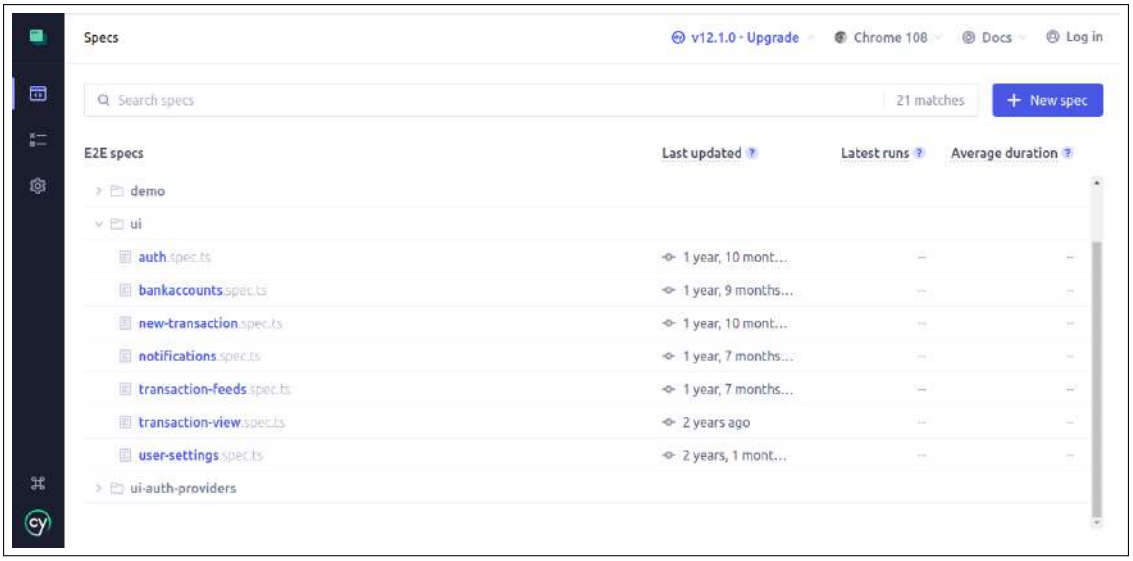


Figure 2.4: Cypress Runner's spec listing for execution

Test cases defined on lines 6, 12, and 19 contain the test title, commands, and assertions. These spec files can either be programmatically generated or through Capture-Replay using the Cypress Runner. The Cypress Runner provides GUI to select spec files for execution as shown in Figure 2.4. The test cases on the selected spec get executed sequentially on a mock browser instance, storing snapshots of each page before and after each command. Time traveling is possible by selecting each command to preview the state of the web application at that moment. State change before and after the command is visible to testers.

2.3 Automated Web Testing

Automated web testing is the technique of automatically crawling the subject web application to model its behavior and execute diverse test scenarios, ideally without any human intervention. Automated approaches explore the dynamic web state space of modern web applications and derive the set of valid interactions (actions) on operable web elements (such as buttons, text, dropdowns, etc.). These actions are prioritized for operation based on some exploration strategy. Actions requiring input values can be generated randomly, through search-based rules,

or genetically. Action sequences (in other words, test cases) that report failures during exploration are stored for further examination.

This section first describes the importance of automated web testing. Secondly, the relevant concepts required to understand the different steps in automated testing are briefly presented. Finally, the challenges faced by practitioners and researchers in automated testing are listed.

2.3.1 Importance of Automated Web Testing

The importance of automated when testing is listed below:

- **Testing process improvement:** Automated testing is not aimed at replacing human knowledge but rather to assist in the overall testing process. In addition to manual testing and automation scripts, automated tools can guide testers to identify failures and drive exploration to less explored areas of AUT. For example, failing to explore deeply nested page logic as they require a specific pattern of interaction to reach [5, 4]. Similar scenarios are frequent in any organization due to the tester's lack of domain knowledge, skill level, and constant change of AUT functionalities. As such, automated tools can work in parallel with traditional testing techniques.

- **Saving resources:** Client requirements are always on the change, and hence, so is the software code. Testers spend a lot of their time validating these changes either manually or through automation scripts. Assuring quality through manual interaction becomes impossible when frequent changes take place [40]. Maintaining automation scripts is no exception to this as slight changes in the web page structure/functionality may break multiple test scripts [9, 40]. Moreover, testers are expected to exercise new possible scenarios in case of updated AUT functionality. These efforts make testing labor-intensive and time-consuming, thus costly. Automated web testing can

greatly reduce the cost and human resources needed [41]. Thus, testers can engage themselves in other quality assurance tasks rather than maintaining previous scripts or manually exploring new scenarios.

- **Increased product quality:** Improved testing leads to better product quality [42]. Automated testing approaches can somewhat reduce the tester's dependency to check simple functionalities like navigation, proper form submission, input validation, form validation, etc. Otherwise, manual testing is prone to errors often due to negligence. Automation scripts can reduce the risk but the initial script development process is also manual [4]. Due to the critical role web applications play in society, a lack of testing simple functionalities can bring huge financial loss. Testers and organizations must rely on automated testing to automate redundant and simple quality assurance checklists. Moreover, automated tools can detect failures in each AUT version, missed during other forms of testing. Testers opt for failure detection as early as possible to reduce fixing costs. Such precautionary measures are taken to improve the software product as a whole.

2.3.2 Relevant Concepts for Automated Testing

Generally, web applications change state (either URL or UI) on each action performed in the current state. The same holds when an automated testing approach interacts with the application to mimic a real user, which can be modeled as a Markov Decision Process (MDP). That is, every automated testing approach can be modeled using a 4-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. Here $\mathcal{S}$ and $\mathcal{A}$ represent the unique states and actions detected by the state abstraction algorithm (discussed later on). Whereas $\mathcal{R}$ and $\mathcal{P}$ represent exploration reward function and transition probabilities. As depicted in Figure 2.5, the automated agent perceives the current state $s_t \in \mathcal{S}$ at any time t. A valid action $a_t \in \mathcal{A}$ is executed and the new state tran-

sition environment is received by the agent. The rest of this subsection describes relevant concepts and steps for automated web testing.

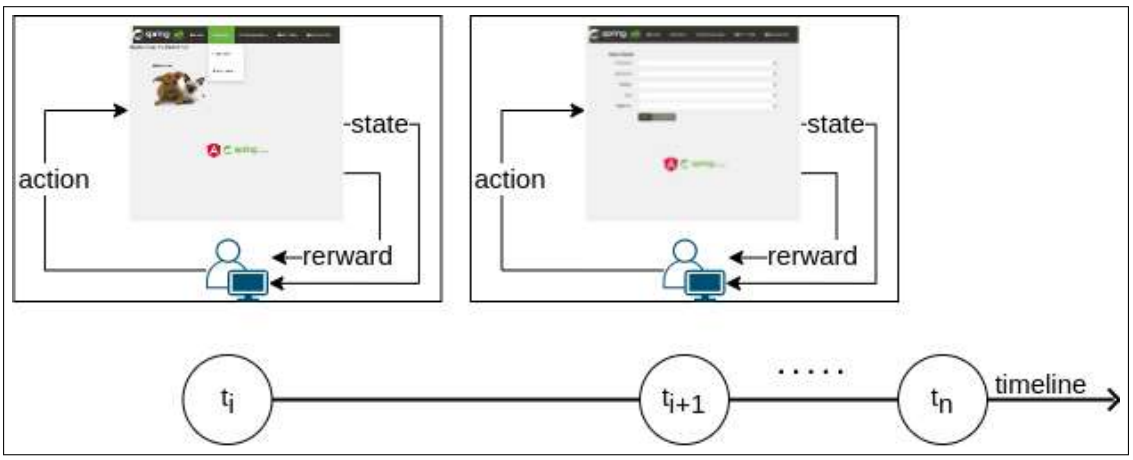


Figure 2.5: Automated Approaches as a Markov Decision Process

2.3.2.1 Navigation Model

Any web application can be conceptualized and represented as a State-Flow Graph (SFG) where the nodes are the different states of the system. The edges connecting two states, say $\langle s_i, s_j \rangle$, is the action a located in s_i. Transition to state s_j from state s_i occurs on executing a (is also represented as $\langle s_i, a, s_j \rangle$). The navigation model is the SFG constructed in this manner to represent the entire web application. Figure 2.6 illustrates a sample web scenario along with its partial navigation model of the open-source PetClinic project [1].

The initial DOM state (index/home page) of the web application is set as the root node of the navigation model. The navigation model can have multiple edges from one state to the other as in websites multiple actions can result in the same state. For example in Figure 2.6, clicking the *home button* from the navbar or clicking the "Spring" logo will both result in navigating to the same page. Moreover, the navigation model can be cyclic as the same web pages can be visited repeatedly. Here, the navigating to and from the same two set of pages can cause cyclic exploration of AUT functionalities.

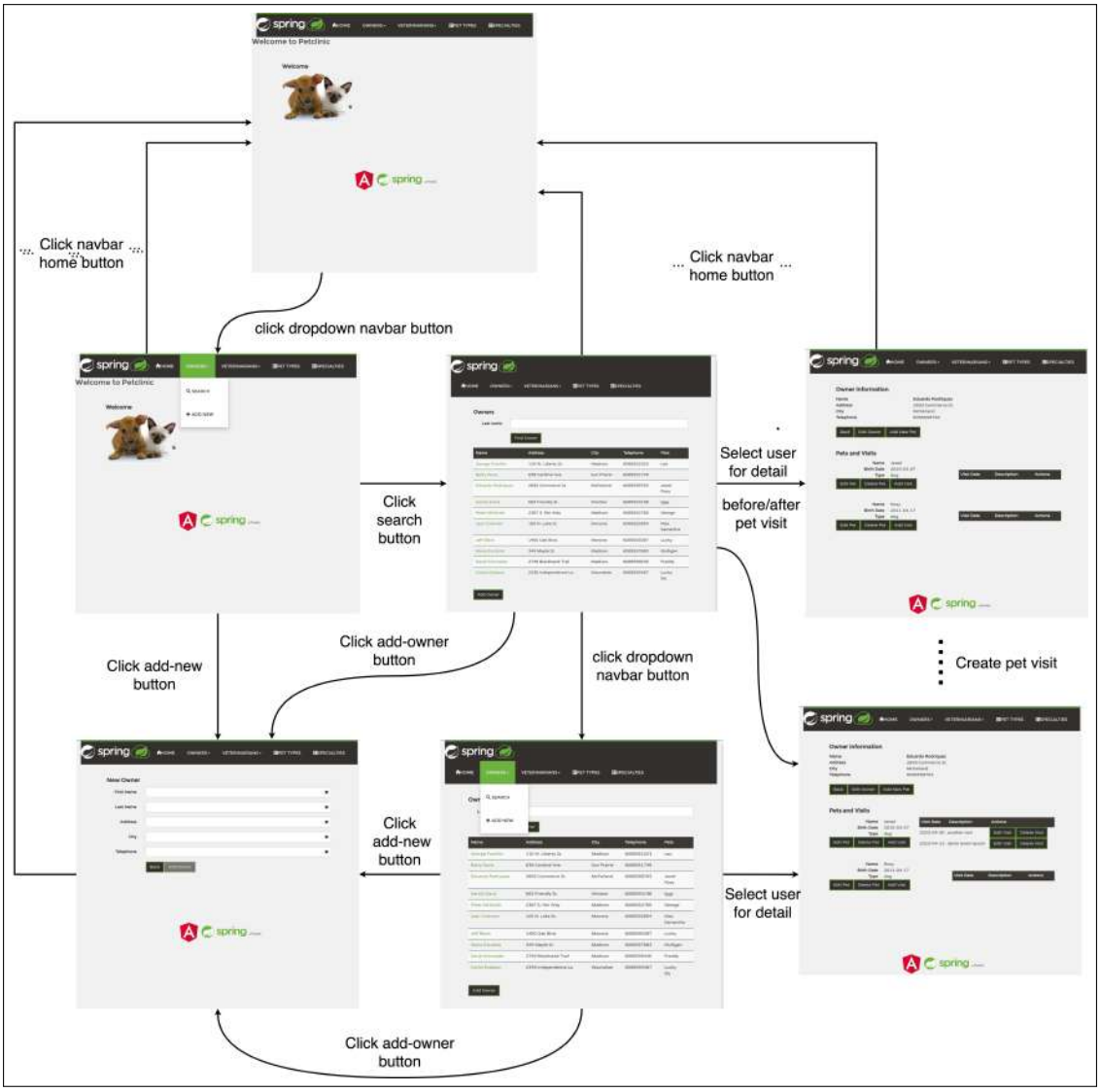


Figure 2.6: Simplified navigational model of partial functionalities from the PetClinic web application [1].

2.3.2.2 State Abstraction

State abstraction is the process to identify states that serve the same functionality. This is done as web applications contain cyclic navigation to enhance user experience. Moreover, a large number of unique web pages or elements can represent the same functionality, for example, in an admin panel, user profile pages for all users are functionally the same. Each user profile page serves the same functionality and testing them all is redundant. This exponentially increasing number of testable states in AUT is known as the state explosion problem [43]. State explosion negatively affects the efficiency of test suite generation [5]. This problem is solved by employing state abstraction to reduce redundantly testing the same behaviors. The abstraction algorithm detects and groups functionally similar states. Then, only states with non-overlapping functionality are focused for testing. Selecting the proper granularity of state separation is challenging due to a tradeoff between state explosion and detecting minor changes in the DOM [43].

2.3.2.3 Exploration Strategy

Modern AJAX-based web applications cannot be fully represented by a static navigation model as contents in a particular state can dynamically change. The navigation model also needs to be dynamically updated with the introduction of new states and actions. For example, the delete page is only exposed after proper entity creation or it can only be navigated if the user is an admin. Hence, new states can emerge or the same page can represent different sets of actions depending on state information. Taking this dynamic nature into consideration, automated testing approaches define an exploration strategy to prioritize action selection in the current state. The action selection strategies can range from simple tree-based traversals (like depth-first or breadth-first) to reinforcement learning based techniques to maximize coverage metrics [12, 5]. Usually, actions leading to new unique states are rewarded to encourage exploiting unseen AUT functionalities.

Exploration strategies revert the AUT back to the root state when unable to explore new states. This promotes testing diverse scenarios.

2.3.3 Challenges of Automated Web Testing

Manual testing allows testers to have a "gut feeling" regarding usability, bug proneness, and assertions to be made. Automated approaches fail in this case but make generic checks to find exceptions, crashes, HTTP errors, or getting the GUI into an unresponsive state. Based on prior literature [43, 10, 41], the challenges faced by automated program repair can broadly be divided into three categories. These are i) state separation granularity, ii) efficient action selection, and iii) exploration completeness. These challenges are described below:

1. **State separation granularity:** Since the state space can increase in an exponential fashion, automated testing tools need to separate functionally unique states neither too coarse nor too fine-grained. If the separation is too fine-grained then functionally duplicated states will be considered for exploration. On the contrary, making it too coarse-grained will make the algorithm incapable of detecting subtle dynamic changes representing new functionalities. Unfortunately, no such universal threshold exists that can properly separate states in every web application [43]. Moreover, modern front-end frameworks dynamically generate attribute IDs and classes on each page load to prevent caching components (allowing dynamic up-to-date data to be displayed). This makes the detection and grouping of similar states more challenging. Near duplicate states heavily impact the accuracy and applicability of automated E2E testing techniques, causing redundant test execution [43, 5]. State abstraction is an open challenge in this domain [43].

2. **Efficient action selection:** Apart from the separation of unique states, the actions inside them have dynamic properties as well. For example, the same

input form can have different input fields based on the user role. Automated approaches rely on code, branch, and state coverage metrics which may lead to sub-optimal action selection. For example, after filling out the input fields, the tool navigates to a new page without properly submitting the form. This would benefit coverage metrics but in reality unable to verify the usage scenario. This sense of "naturalness" in action selection is also an open challenge in web testing.

3. **Exploration completeness:** Automated testing tools cannot ensure exploiting every usage scenario even with ideal state abstraction. Applications may expose unseen states and actions only after being interacted with in a specific pattern. Testers use their domain knowledge and boundary value analysis to expose such scenarios [4]. Automated tools might consider functionality to be tested and give them less priority, whereas more unexpected usage scenarios can be exercised. For example, typically web forms are a prominent source of failures and on proper submission, automated tools might consider exploration completeness. However, the same form might expose new states or failures on unexpected usage scenarios like - submitting an empty form or filling partial inputs. To this end, how automated tools can achieve human-like skilled exploration is still a big challenge.

2.4 Summary

This chapter creates the knowledge base to aid readers understand automated web testing. To do so, this chapter initially elaborates on relevant terminologies and concepts in web application testing such as web application architecture, web state, etc. Later on, the current practices in manual web testing and automation are presented as readers might not be acquainted with this domain. The advantages and disadvantages of each approach are discussed to establish the im-

portance of fully scriptless automated testing approaches. Finally, automated web testing is introduced along with its importance, related concepts, and challenges in literature. Automated testing can minimize the effort needed to maintain test automation scripts after changes in AUT. Moreover, basic functionality checks such as navigation, form submission, input validation, etc. can be done without human intervention. In short, fully automated web testing can be used to verify diverse scenarios more often and find failures early in the development life-cycle.

Web applications can be modeled as a State-Flow Graph where the states are representations of web functionality at a particular time. Transitions (edges) are made to the UI or DOM based on interaction with each state. However, numerous states can behaviorally represent the same functionality and also contain cyclic links between them. Due to this automated approaches face major challenges in - i) identifying functionally similar states (state abstraction), ii) prioritizing interaction for efficient exploration, and iii) exploiting all usage scenarios (corner cases) in the AUT. Based on the concepts described in this chapter, the next chapter elaborates on the related work on automated web testing.

Chapter 3

Literature Review on Automated End-to-End Web Testing

Web application testing has been in focus in the research community due to the challenges imposed by it [44, 27, 8]. Researchers have tried to aid developers and testers in exploring bugs before release. A lot of research effort is taken to improve all approaches of testing - i) manual, ii) automation, and iii) automated. However, the research contributions concerning manual and automation scripts do not make the process fully automated (hands-free). Existing scriptless automated web testing approaches can be broadly categorized into two categories. These categories - i) model-based, and ii) model-free, are done based on the usage of static navigation model. Model-based approaches usually rely on a static navigation model that generate test cases in a random or search strategy. As they are based on navigation models, pages that are not included in the navigation model are never explored and thus not tested by the approach. On the other hand, model-free approaches can adjust to dynamic changes in AUT, and the exploration is driven by different coverage metric-based rewards. However, search completeness is not guaranteed due to sub-optimal exploration decisions. Each of these categories and their subcategories are described in detail in the subsequent sections.

3.1 Model-Based Automated Approaches

Model-based automated testing approaches are the most common type in literature as test cases can be generated without executing the application [4]. They rely on a static navigation model, requiring external assistance or the tester's domain knowledge [5]. Some prominent approaches, ATUSA [25], SUBWEB [11], and DIG [4], fall into this category, which are discussed in the following subsections.

3.1.1 ATUSA

ATUSA [25] is the first automatic test generation approach for modern asynchronous *Web 2.0* applications. Approaches prior to ATUSA are applicable for classical multi-page web applications and desktop applications which pose serious limitations for *Web 2.0*. At that time, Capture-Replay tools were the only means of end-to-end testing of client-server based asynchronous web applications. ATUSA uses client-side invariants to assert the correctness of such applications. ATUSA contributes to providing a mechanism to express these invariants and validate them dynamically on the fly.

Initially, CRAWLJAX [12], a model-free crawler to automatically generate the application State-Flow graph, is used to derive the AUT state machine. During the crawling process, random inputs are provided to discover new states. However, custom values are provided to bypass scenario logic, for example, a valid email and password for authentication. The AUT is reloaded on each trace since the browser does not register the history of changes made by dynamic DOM events.

Generic and application-specific invariants are used to solve the Oracle problem [45] (correctness assessment of the test case output). A fault model[46] is associated with each invariant that represents an AJAX-specific fault. The generic DOM-tree invariants used are - i) validated DOM; inspect HTML validity on dynamic execution to prevent browser crashes and security vulnerabilities, ii) no

error messages in DOM; error messages (either client-side or server-side) would indicate deviation from intended behavior, iii) accessibility and i18n compliant DOM; comparing dynamic state changes with Accessible Rich Internet Applications suite (ARIA) guidelines[1] and W3C internationalization and localization (i18n)13 guidelines[2], and iii) secure states; checks made to ensure no element can change DOM content or steal data from it [47]. On the other hand, ATUSA allows expressing application-specific invariants through constraints/preconditions using XPath, regular expression, or JS expressions. For example, the logout button clicked from any page should redirect users to the login page. Additionally, ATUSA checks generic state machine invariants like - i) no dead clickables; detect whether any button or link is unresponsive or unavailable, and ii) consistent back-button; whether back button functionality in the browser is registered with each page of AUT and not exit from the application.

Using these invariants paired with the obtained test paths, the subject systems are tested to detect faults and performance/coverage metrics. In a commercial product, Coachjezelf[3] ATUSA detects six types of failures among which three are from generic invariants and others from application-specific invariants. An evaluation on TUDU open-source web application [4] resulted in 75% client-side code coverage within 6 minutes of execution. However, ATUSA utilizes uses inputs on test paths traversing the static navigation model. The feasibility of test cases generated in this manner is often low as they require a large number of input generations and corresponding executions. This makes the size of test suites generated by ATUSA fairly large even for simple web applications [11, 48]. Moreover, the state separation function is based on simple DOM string comparison (Levenshtein edit distance [49]) that is responsible for a large number of functionally similar states, and thus many test paths.

[1]https://www.w3.org/WAI/standards-guidelines/aria/
[2]https://www.w3.org/International/publications
[3]https://www.coachjezelf.nl/
[4]https://www.julien-dubois.com/tudu-lists

3.1.2 SUBWEB

SUBWEB [11] is proposed by Biagiola et al. that generate test cases via path selection and input generation using genetic algorithms. Here the navigation model is constructed using manually defined page objects. Page objects encapsulate the web element access details in a web page and expose interfaces to operate with those elements. Page objects return the next page object on navigational method invocation that represents the web state after transition [50]. As such, developers latently expose the navigation model by defining the web application logic as page objects. Moreover, the authors argue that depending on page objects is not a disadvantage as they are popular among practitioners due to their reusability and maintainability aspects, even after following their defined annotation patterns.

Traditionally approaches prior to SUBWEB depend on the crawling-based mechanism of AUT to extract the navigation model. However, crawlers are prone to limit exploration of full AUT functionalities as they resort to random input generation. This causes two-fold problems - i) inputs can be stuck to local optimum regions (failing to exercise corner cases), and ii) generate infeasible test paths (that is, test inputs do not match with the actual execution scenario). SUBWEB takes advantage of the tester's domain knowledge incorporated in manually written page objects. However, preconditions need to be defined in the page object navigation methods to automatically extract guard rules (such as, the user must be logged in for this navigation). Test inputs are derived by analyzing the parsed preconditions to promote testing diverse scenarios. SUBWEB will not generate test paths for which the guards are not satisfied and throw exceptions. Later, the navigation model is derived using page objects as nodes and the navigation methods as edges. Moreover, infeasibility is further tackled in test path and input generation through employing genetic algorithms.

SUBWEB takes motivations from search-based test generation of object-oriented systems [51] that generate method sequences (test path) and parameter values (ac-

tion input) at the same time. Test generation is aimed at invoking all navigational methods defined in page objects (that is, exercising all edges in the navigation graph) to satisfy the transition coverage adequacy criterion. To promote diverse scenario execution at first, the fitness function is the sum of branch distances of uncovered paths. The test path traces are stored during each execution of navigational methods. Genetic algorithms such as crossover, mutation and are modified to take the navigation graph structure into account. In case of crossover, two test traces are broken and merged to form two new test cases if their intersection point consists of the same source-target page object pair. On the other hand, mutation operations are a combination of *change*, *delete*, and *insert* operations. The *change operator* simply finds and replaces two methods with the same source-target page object pair. The *delete operator* randomly selects a method $m_n(P_i \rightarrow P_j)$ in a test case and keeps deleting subsequent methods until another method with the source page object P_j is found. The *insert operator* starts from the end of a test case and from its target page object finds a path to a randomly selected unexplored method (navigation branch). The methods linking the existing test case and the uninvoked method are added to the end to generate a new test case. A case study on AddressBook open-source project reveals that the crawling-based technique deviates from ideally defined manual PO to a major extent. Moreover, the size of the test suite is 11 times smaller than the baseline with higher transition coverage. SUBWEB solves ATUSA's test suite size problem and test path infeasibility due to random inputs. However, SUBWEB needs manually defined page objects to derive the navigation model. Large numbers of candidate test cases need to be generated and executed to verify feasibility. Moreover, each page object needs manually specify guards and preconditions. This makes SUBWEB very much labor-intensive and time-consuming. This dissolves the benefit of automated approaches as such a manual step is not possible without domain knowledge and a good understanding of AUT business logic.

3.1.3 DIG

Biagiola et al. again later proposed an approach named DIG [4](DIversity-based Generator) that prioritizes generating candidate test cases with the most structural diversity. The candidate test case generation process relies on a diversity-based distance heuristic comparing each time against existing test candidates. The web application does not need to be executed to generate a test candidate pool. Such a heuristic is motivated by the fact that executing diverse usage scenarios can detect failures early in the process. In turn, this would improve failure detection efficiency as web in-browser test execution is slow.

The key contribution of DIG is to generate a diversity-aware distance heuristic given that diversity enhances fault detection [52]. Firstly, the navigation model of the AUT web page is generated using a crawling mechanism. The crawled web pages are iterated and their corresponding page objects are generated. This is done keeping in mind the practical popularity and maintainability. The crawling and automatic page object generation is done using the APOGEN tool [8]. Using the navigation model and DIG's novel distance heuristic (elaborate later) the set of candidate test cases is generated. This step is implemented on top of EvoSuite [51] (a test suite generation tool). Later the abstract candidate test cases are validated and actualized using the page objects. Utilizing the page objects the test cases are executed using Selenium WebDriver [53].

Model-based techniques relying on random or search-based exploration are computationally expensive and prone to infeasibility problems. This is because they need to execute the test case and then determine its validity. Even so, the generated test cases are likely to be behaviorally similar. DIG removes solves the infeasibility problem [11] by employing page objects to validate valid state actions. Moreover, the distance function ensures structurally diverse candidate tests. The heuristic starts from a random test case traversing the obtained navigation model. For each subsequent candidate test case, the difference in node-action pair from

existing generated test cases is computed. The deviation between the set of node-action pairs between two test cases is calculated based on sequence edit distance [54, 49]. Moreover, the distance between input values selection for each action is calculated. After computing the distances for all candidate test cases from already executed test cases, the one with the maximum distance is executed. This is added to the set of final test cases if it contributes to an increase in coverage of the navigation model. This process is repeated until the time budget is exhausted.

DIG results in state-of-the-art performance in fault detection and coverage among model-based approaches. The evaluation was done on six open-source benchmark projects implemented in different popular frontend frameworks. The distance calculation function is computationally expensive as the number of comparisons needed for each candidate calculation iteration increases exponentially. Although experimentation showed that the advantage in efficiency is overshadowed by this distance computation overhead. Although DIG generates the navigation model and page objects automatically, it is not free from manual intervention. Crafted inputs are required to exploit the corner cases and accomplish business logic. Even with that static navigation models often fail to fully grasp the dynamic nature of AUT. Reliance on expert domain knowledge to generate high quality test models motivate the need for model-free approaches.

3.2 Model-Free Automated Approaches

Model-free approaches dynamically exercise the state space of web applications following some exploration strategy. They are not subject to problems faced by model-based approaches due to employing a static navigation model. Model-free approaches usually do not require human intervention and rely on the exploration strategy. Whereas, model-based approaches may be mired to specific areas of AUT and cannot consider states exposed dynamically. However, model-free approaches are not superior to model-based approaches in all scenarios as domain knowledge is necessary to build high-quality test suites [4]. The following subsections elaborate on some of the most influential model-free approaches and how they work.

3.2.1 CRAWLJAX

CRAWLJAX [12] is the first technique to enable the crawling of dynamic AJAX event-driven websites. CRAWLJAX pioneers model-free testing strategies in the literature. What started out as a technique to automatically infer the state machine of a subject system, is widely used as the preliminary step for a handful of model-based automated testing approaches. Different approaches conduct operations on the inferred state machine (navigation model) to generate test cases. CRAWLJAX itself when deriving the state information, can store traces as test cases and report detected failures. CRAWLJAX adopts random action selection and input generation policy to discover new states.

At first, the web application under test is opened in the browser and the DOM tree is parsed. By default, HTML tags $\langle A \rangle$, $\langle BUTTON \rangle$, or $\langle INPUTtype = submit \rangle$are filtered from the DOM. These elements are considered as the candidate clickables responsible to change the current state. However, other rules (such as XPATH) can be defined by users to select candidates as any element can support interactions through dynamic events. The crawler initiates interaction with the

resulting set of candidate actions and waits for state transition. Now, the state comparison algorithm is employed to identify whether the resulting state is a new one or explored earlier. Simply the difference in transitions can be computed using the Levenshtein method using some similarity threshold τ. Another approach embedded in CRAWLJAX employs a series of comparators, one precedes the other to check similarity. For example, a DateTime comparator checks whether two dates are present in the DOM tree and filters those. Incrementally the comparators match elements in the DOM trees and then calculate the difference between them. If the difference is greater than the threshold, meaning the states are different, the algorithm checks whether any existing states' DOM matches with it using the same process. Finally, if the tree is new, it is added to the State-Flow Graph. In AJAX-based applications, browsers are incapable of executing the "back" functionality to restore the previous state. As such, CRAWLJAX stores the trace information to restore the application state (containing unexplored candidate actions) from the root. From this state incrementally all unexplored candidate actions are executed to generate the final navigation model.

CRAWLJAX furthermore implements a multithreaded crawling mechanism to improve execution speed. Here the global State-Flow graph is maintained and crawling tasks are managed by the main thread. Each trace is handled by threads provided with their own embedded browser to allow parallel browser execution simulation. The original work by Mesbah et al. focused on the evaluation of properly generating candidate clickables and AUT state machine. However, CRAWLJAX to date acts as a baseline for evaluation in web testing based on failure detection and code coverage in many studies citation. CRAWLJAX as a testing tool itself produces too many redundant test cases, focusing exploration of the same AUT behaviors. This is due to the simple DOM string matching-based state abstraction, expanding states having similar behavior. Alongside redundancy, random input and exploration hugely affect efficiency and effectiveness.

3.2.2 FEEDEX

FEEDEX [3] by Fard et al. is a model-free approach that prioritizes maximizing functional coverage in minimal time rather than exhaustively crawling the AUT's state space. Exploration decision is taken by combining four factors - i) code coverage impact, ii) navigational diversity, iii) page structural diversity, and iv) test model size. Based on these factors the exploration strategy knows beforehand which action to exercise to maximize coverage. Authors evaluate higher yield in coverage metric compared to exhaustive crawling using depth-first, breadth-first, and random strategies.

FEEDEX employs a greedy approach to select and expand partially expanded states (states containing one or more unexercised actions) that have a positive impact on aspects such as - functional coverage, navigational coverage, page structure coverage, and test suite size, within a limited time budget. The partially expanded state is selected for further exploration based on the -

- **Code Coverage Impact:** $CI(s)$ is the code coverage impact due to exploration of state s. The increase in code coverage from exploring s is measured in code coverage impact. Suppose, loading the owner list page executes only 3 lines of JavaScript out of a total of 30 lines. The increase in code coverage, $CI(s_{owners}) = 0.1(\frac{3}{30} - 0)$. Now the pagination button *next* is clicked to execute a total of 5 lines of code. Therefore $CI(s_{pagination}) = 0.06(\frac{5}{30} - 0.1)$, which means code coverage will not increase beyond 6% even with multiple clicks to the pagination button. If another button, say *owner search* is pressed to execute 15 lines of code, then similarly the $CI(s_{search}) = 0.4$. This in turn promotes exploration of the search functionality in further traces.

- **Path Diversity:** Diverse paths in the navigation model are more likely to cover different functionalities. Whereas functionalities on the same path are less likely to cover diverse scenarios. The path similarity between two state

s_i and s_j from the root is denoted by:

$$Pathsim(s_i, s_j) = \frac{2 * \left|P_{s_i} \bigcap P_{s_j}\right|}{|P_{s_i}| + \left|P_{s_j}\right|} \tag{3.1}$$

where $|P_s|$ is the length of path from root to state s and $\left|P_{s_i} \bigcap P_{s_j}\right|$ denotes the unique events common to both paths. In the web, each state can have multiple paths from the root. The $MaxPathSim(s_i, s_j)$ calculates the maximum similarity to finally calculate the path diversity $PD(s_i, s_j)$:

$$PD(s_i, s_j) = 1 - MaxPathSim(s_i, s_j) \tag{3.2}$$

Using this equation states with maximum path diversity from explored ones can be calculated and expanded.

- **DOM Diversity:** To minimize exploration by making small changes in the DOM, actions that increase DOM structure diversity are prioritized. A significant change in DOM structure leads to diverse functionality. The DOM diversity is calculated as follows:

$$DD(s_i, s_j) = \frac{TED(t_i, t_j)}{max(|t_i|, |t_j|)} \tag{3.3}$$

where $TED(t_i, t_j)$ represents the tree edit distance [12] and $|t|$ is the length of the state's DOM tree.

The overall state score is a combination of these metrics. The state score determines which state to expand, that is, the state with the highest score is selected. The state score equation is a linear combination defined as:

$$Score(s_i, s_j) = w_{CI} \cdot CI(s_i, s_j) + w_{PD} \cdot PD(s_i, s_j) + w_{DD} \cdot DD(s_i, s_j) \tag{3.4}$$

here the $w_{CI}, w_{PD}, and w_{DD}$ represents configurable user-defined weights to prior-

itize metric(s) on demand. These weights are not always similar as testers might prioritize one factor over the over.

FEEDEX is not purely a web test generation technique but rather creates a test model. This test model represents the AUT functionality state machine. Empirically FEEDEX is evaluated on six web applications against traditional exhaustive search strategies. Results yielded at least 10% and 23% better code coverage and diversity respectively. Moreover, a reduction of at least 38% and 42% is seen in the size of the test model and test suite respectively. This illustrates the efficacy of FEEDEX to generate test cases exercising diverse usage scenarios.

3.2.3 WebExplor

WebExplor [5] is the first model-free reinforcement learning (RL) based automated end-to-end web application testing approach proposed by Zheng et al. The exploration is driven by a curiosity-based reward function paired with a DFA (Deterministic Finite Automata) guidance mechanism. Modeling modern state-based web applications as a Markov Decision Process enables the applicability of reinforcement learning. The core algorithm of WebExplor consists of three major components - i) pre-processing; mapping HTML pages to abstracted states, ii) curiosity-based exploration and exploitation, and iii) DFA guidance; to prevent exploration from being "stuck" to local optima.

A novel state representation mechanism is devised to tackle the exponentially increasing number of states in web applications. The mechanism consolidates HTML pages into one single state that focuses on the same business logic. Even with variations in HTML code, the user interactions may be similar. WebExplor for this reason focuses on operable actions in DOM and HTML structural similarity. Given a web page, the state representation is made by tag-wise comparison of HTML code. Comparisons with all previously explored states are made to identify whether the functionality of the current state is already explored. This compari-

son is made based on a predefined threshold. Moreover, the two states are termed different if their URLs do not match. If the current state is not similar to any other previous states, then it is added to the set of unique states.

WebExplor focuses on generating valid action sequences rather than input values. The action sequences generated during exploration operate on random values. The random values are generated according to W3C standards [55]. WebExplor aims to maximize state exploration and thus its reinforcement learning rewards actions responsible for discovering new unique states. The notion of curiosity comes from the frequency count of each action executed so that the probability of selecting less executed actions increases. This frequency count table $N(s_i, a, s_j)$ is utilized to calculate the curiosity value as:

$$curiosity(s_i, a, s_j) = \frac{1}{\sqrt{N(s_i, a, s_j)}} \tag{3.5}$$

Initially, every tuple count $N(s_i, a, s_j)$ is set to 1. On each transition from s_i to s_j executing action a, the count is incremented. This curiosity values heavily impacts the reward function used in the Q-learning [56] (an RL algorithm). The Q-value in WebExplor is defined as:

$$Q(s_i, a) = curiosity(s_i, a) + \lambda max_{a'} Q(s_j, a') \tag{3.6}$$

where λ is the discount factor ranging $[0, 1]$. The Q-value propagates recursively to antecedent states. Finally, the Gumbel-Softmax method [57] is used to define the action selection policy using the Q-value.

$$\pi(a) = \frac{exp(\frac{Q(s,a)+g_a}{\tau})}{\sum_{a_i \in A} exp(\frac{Q(s,a_i)+g_i}{\tau})} \tag{3.7}$$

here A is the valid set of actions in state s, $\tau = 1$, and g_x are noise sampled from Gumbel(0,1) distribution. The higher the Q-value, the higher probability of that action being executed. This policy enables WebExplor to balance exploration and

exploitation. Actions finding new states are rewarded with high curiosity value. This in turn increases the Q-value to make that action more probable for execution, promoting the exploitation of new functionality. Operating on the same action gradually reduces the curiosity (hence, Q-value) and hence facilitates exploration of less visited actions of AUT. Even with this learned policy function, many a time, deeply nested functionalities are not executed as they require a long sequence of actions in a specific manner. Interrupts due to failure or other unexplored actions make testing such features difficult. As such, WebExplor employs a DFA mechanism. Exploration traces are stored along with their curiosity in a deterministic finite automaton. This mechanism is triggered when WebExplor fails to find a new unique state within 15 minutes of execution. The action with the highest curiosity is determined and the transition from the root to that action state is obtained from the DFA. WebExplor starts exploiting that scenario using the transition trace obtained.

WebExplor displays state-of-the-art results compared to other model-based, model-free, random testing approaches. Zheng et al. evaluate webExplor on six popular open-source projects, a commercial SaaS web application, and the world's top 50 websites ranked by Alexa [58]. WebExplor outperforms all existing baseline approaches in terms of failure detection and 4 out of 6 projects in terms of code coverage. Statistical significance is tested using the Mann-Whitney U test [59]. Moreover, WebExplor discovered 3,466 failures from real-world web applications.

WebExplor however, is not without its limitations. A large number of failures in benchmark subject systems are revealed only after a significantly longer period of interaction than usual. This is due to the combined effect of their action extraction from the state abstraction algorithm and action selection policy. Web applications usually contain repeated actions in subsequent states for improved user experience (usually for easier internal navigation) and these actions are considered unique in all states by WebExplor. For example, the layout actions (such

as links, buttons, dropdowns, etc.) in the footer, sidebar, or navbar are considered unique actions in each state. WebExplor assigns high curiosity values initially as they are considered unique. As a result, redundantly these actions are exercised without any benefit in functionality coverage. Moreover, the action selection policy introduces randomicity sampled from the Gumbell distribution to enhance exploration. This technique backfires when redundant actions are explored again, even after initial execution to reduce curiosity value. The DFA mechanism aims to guide the exploration of less visited actions every 15 minutes when new states are not found. Even so, on reaching the less visited state, the redundant actions in that state often cause ineffectiveness.

3.3 Summary

Automated end-to-end web testing approaches in literature can be broadly divided into two categories based on the usage of static navigation models, namely model-based and model-free approaches. Some of the prominent approaches of these two categories are discussed in this chapter. Model-based approaches rely on a navigation model (a State-Flow graph) constructed manually or automatically. This model is traversed to generate test cases. ATUSA [25], SUBWEB [11], and DIG [4] are examples of this category. For example, the navigation model is generated through crawling the AUT using CRAWLJAX [12] in ATUSA, ii) manually designed page objects to represent states in SUBWEB, and automatically using APOGEN [8] in DIG. As observed, the preliminary step is to model the AUT behavior using the navigation model. This static model is utilized to generate test cases. For example, ATUSA traverses the model and validates invariants, SUBWEB uses genetic operations to generate diverse test paths, whereas DIG utilizes its distance algorithm to generate candidate test cases that are the most diverse from each other. The main drawback of model-based approaches is the reliance

on static navigation models and the infeasibility of generated test cases [4]. Often test actions are not valid and models fail to capture the full essence of AUT [5].

The second category of testing approaches is model-free techniques. Such techniques remove the dependency on static navigation models. Considering the dynamic current state of AUT at a particular time, actions are executed based on an exploration strategy. CRAWLJAX [12] is the first influential contribution to this category. CRAWLJAX started with the goal to derive the state machine of any web application. This state machine can be specifically used to generate test models and for application functionality comprehension. The state machine explored the AUT in a random fashion with randomly generated inputs. FEEDEX [3] by Fard et al. focuses on functionality coverage. The exploration strategy used is a linear combination of code coverage impact, test path diversity, and DOM diversity. States with the most score were selected for expansion. This work contributed to metrics closely related to judging an efficient test model. Evaluations done on FEEDEX revealed better code coverage and diversity along with reduced size of the test model and test suite. The more recent contribution WebExplor [5] produces state-of-the-art results compared to existing model-based and model-free approaches. It relies on a curiosity-driven reinforcement learning approach paired with DFA guidance to help exploration of less visited functionalities. The failure detection capability and code coverage obtained from these approaches are satisfactory but still not adopted by practitioners [10]. This is because, model-based approaches require human intervention, whereas model-free lack human domain expertise. Besides, the test cases generated do not produce natural interaction scenarios. Even for the state-of-the-art, WebExplor, randomness introduced from the action selection policy and duplicated cyclic navigation bring a negative impact on its naturalness. The next chapter contributes to a guideline dataset that encapsulates the natural exploration decisions of a human tester.

Chapter 4

WebEV: A Dataset on the Behavior of Testers for Web Application End to End Testing

Automated End-to-End (E2E) web testing is a key component in modern rapid development to validate system functionality. However, there are no resources supporting practitioners on how diverse scenarios are tested manually. This chapter presents WebEV, a dataset containing E2E test cases from open-source popular projects. Projects are selected based on - i) Cypress-based automation, ii) popularity on GitHub, and iii) executability of test cases. The dataset contains information regarding each test command along with the incurred state change representation. Snapshots of the application are used to retrieve - i) the current URL of the application, ii) the screenshot and HTML text of the entire page, and iii) the screenshot and HTML text of an operated UI element. This process is done both before and after each command execution to capture the perception of testers on each state transition, i.e., extract their thought process during testing. This dataset can assist the research community to model user web interaction, predict the tester's perception, and improve the state of automated testing approaches.

4.1 Introduction

Modern web applications provide a rich set of interaction patterns that make the testing process harder. Testers are challenged to adopt a wide range of strategies to continuously re-validate the diverse usage scenarios [60]. Despite the availability of many automated testing approaches in literature [3, 4, 5], their industrial satisfaction is limited [10]. Such approaches do not learn the tester's thought process to effectively validate diverse functionalities. This is due to the unavailability of datasets that encapsulate the behavior of testers during manual web testing.

E2E testing is the most prominent type of testing as it simulates real user behavior to validate each diverse scenario [10, 29, 27]. Unfortunately, no dataset in literature represents how testers manually do E2E web testing. Instead, researchers have devised - automated testing techniques that reward coverage metrics [3, 4, 5], usage statistics based user behavior modeling [6], domain-specific language to generate tests [7], automatic page-object generation to encapsulate web pages [8], stable web locator generator for automation scripts [9], and so on. These efforts focus on either fully automated approaches that lack the tester's domain knowledge (maximize code metrics) or improving automation script generation that is expensive to maintain[10]. Integrating the benefits of both may improve the current practice, that is, incorporate the tester's behavior to test the subject system's diverse scenarios labor-free.

Linares-Vasque et. al. [61] addressed this gap to improve the unsatisfactory state of Android automated testing. The study surveyed 102 open-source Android contributors about their approaches taken during manual testing. The fact that the code coverage metric is not useful to 51 respondents, draws more focus on exploring the behavior of testers during manual testing. Moreover, research efforts have been made by Pecorelli et. al [62] and Christophe et. al [63] to statically analyze the quality of open-source test cases in Android and web domains respec-

tively. However, these studies focus on code quality and flakiness rather than interaction strategies, failing to represent the tester's interaction patterns.

In the Android automated testing domain, Humanoid [64] leverages UI interaction patterns to make test interactions more natural. These interactions are learned from the Rico dataset [26]. Rico is a crowdsourced dataset containing app metadata, design, and UI interactions. The WevQuery dataset [65] similarly contains general user interaction patterns for the web. However, such datasets fail to consider the behavior exhibited by testers during E2E testing. Certain scenarios of the AUT may be left unexplored as users do not interact with the intention of finding defects. To the best of acquired knowledge, no such dataset exists in the literature that encapsulates the tester's behavior for web application E2E testing.

The above discussion indicates that current model-free automated techniques fail to satisfy practitioner expectations due to the lack of human-like interactions. A dataset needs to be constructed keeping in mind the real-world practice, diversity of the web ecosystem, and extendibility. Thereafter, interaction patterns learned from the dataset can be used to understand the tester's usage scenarios. These patterns can improve the current state of - i) user interaction modeling, ii) tester perception guidance, and iii) automated testing.

4.2 Dataset Construction

This paper presents WebEV, a dataset representing the way testers perform manual web automated testing. Such a dataset can be utilized to make efficient test cases and automated testing more acceptable. The WebEV dataset is generated by extracting the tester's interaction sequences during End-to-End (E2E) web testing, as depicted in Figure 5.1

4.2.1 Objectives

WebEV aims to mine the tester's behavior, which replicates real user scenarios to validate the AUT. The design of WebEV is driven by the following broad concerns:

Real-world practice: Practitioners do not always follow the theoretical best practices. WebEV aims to collect the tester's behavior from real-world large systems with rich collaborated development history. This ensures discovering the generalized trends and practices of testers at scale.

Diversity: Web applications are usually an amalgam of diverse frontend and backend technologies. Such technological and domain-specific complexities should not interfere with dataset construction. This in turn would make the dataset generalized and allow a richer diversity of the tester's behavior.

Extendibility: The dataset construction process should be such that it can incorporate new projects in an automated way. The test case extraction process should not be affected by newer technology or application domain.

Automatability: Manual extraction of test cases and test command from large-scale projects are labor-intensive and erroneous. Keeping automatability in mind, projects are selected that have well-maintained test cases and clear documentation regarding dependencies and test cases.

4.2.2 Source of Tester's Behavior

Identifying web testing artifacts that contain interaction sequences is the initial step to constructing the dataset. The goal is to extract interactions such as clicks, inputs, hovers, etc. along with the element being operated on. These interactions can be extracted in a similar fashion irrespective of the underlying technology used. This is because all client interactions are plain JavaScript to the browser. Keeping this fundamental web application property in mind, the most straightforward way is to gather professional testers and assign the task of hunting faults in sample web

apps. Afterward, their interactions can be recorded for further analysis. However, this approach would be very time intensive, non-replicable, and prone to the tester or organization-specific practice. On the other hand, open-source projects can be selected to avoid such bias. E2E tests can be mined from projects with a high degree of collaborative and maintenance efforts. These test cases have a similar life-cycle as code changes, from being requested, developed, and then maintained in a peer-reviewed process. GitHub is selected to search web projects with E2E tests due to its dominance in the open-source development community.

4.2.3 Selection of Testing Tool

Web projects using Cypress [66] for E2E testing are considered for this study. Although web testing literature heavily focuses on Selenium-based test cases, Cypress has become the de facto for automated E2E web testing. This is reflected by Cypress's 3,823,184 weekly npm [67] downloads as of 2023-02-02, whereas the alternatives selenium-webdriver and playwright have 1,962,681 and 929,853 respectively. This indicates the current popularity of Cypress based test automation.

Cypress runner starts a mock browser instance and tests execute in the same run loop as the subject application. Consecutively, it takes snapshots during execution, enabling time traveling back to the application state on each test command. Therefore, dynamic analysis of the project's test commands and their states can be examined irrespective of their underlying technology.

4.2.4 Subject System Filtering

The goal is to narrow down peer-reviewed GitHub repositories using Cypress that have traction in the community, i.e., stars, forks, and contributors. For selection purposes, repositories using *cypress-io/cypress* or eslint-plugin-cypress as dependencies are listed using ghtopdep [68]. Merging projects having these dependencies and removing duplicates yielded 3345 projects. Similar to prior work [4, 69], using

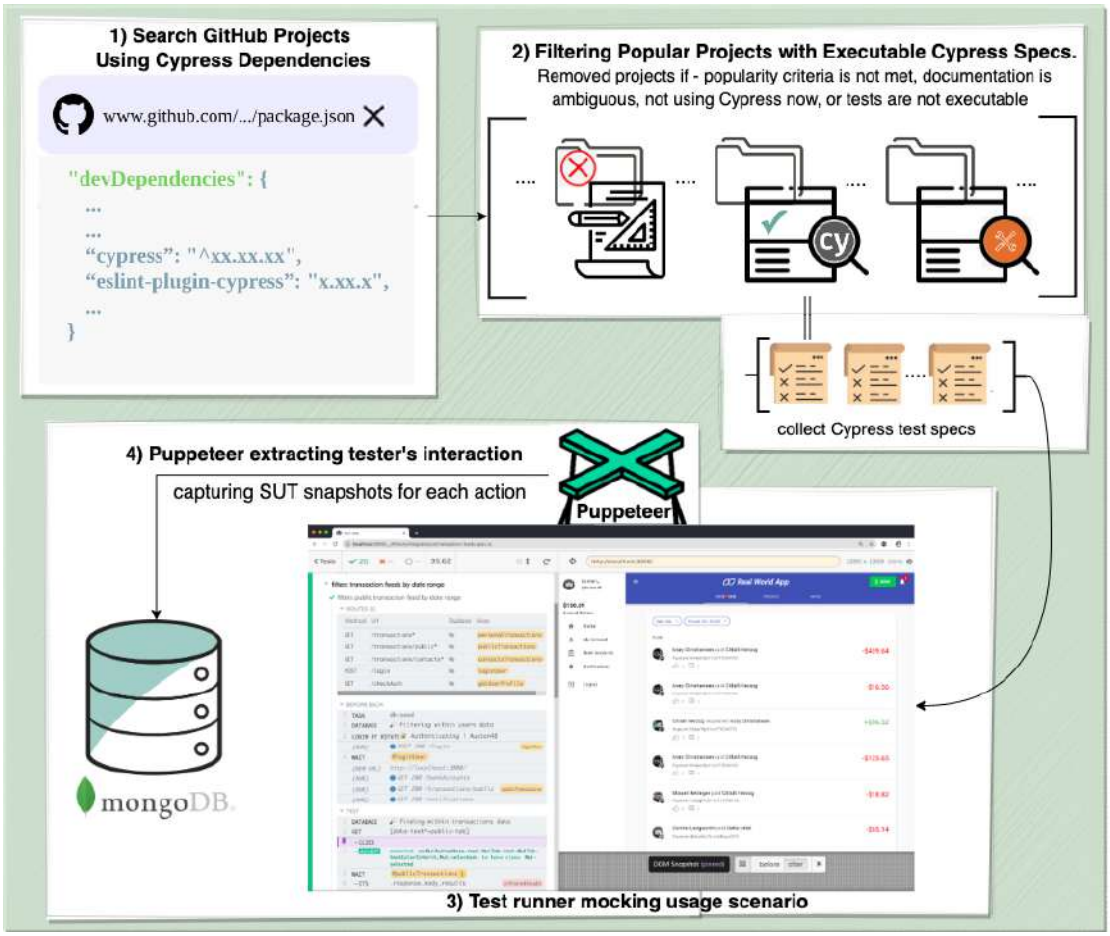


Figure 4.1: Overview of approach to collect tester's usage scenarios

the GitHub API, projects are shortlisted if - i) #stars $\geq$ 50 to measure popularity, ii) #commits $\geq$ 50 for maturity, and iii) #contributors $\geq$ 3 to ensure project received outside collaboration. This filtering led to 1420 projects. A manual investigation was done to further filter repositories not being web applications or not actively using Cypress for E2E testing. Dependencies to run each of the projects are resolved and tests are executed using the Cypress runner. Repositories failing to execute the test cases are neglected as those did not maintain the test specifications along with code changes. Similar to literature [70], the top 100 web-based projects (sorted based on GitHub popularity) are considered due to time constraints, which should be a good representation of current practices.

4.2.5 Extracting Test Cases

Cypress testing specs contain a set of test cases, which are further broken down into individual commands. The commands and snapshots are obtained by gaining control of the mock execution browser using Google's Puppeteer[71]. Dynamic execution is used rather than static analysis to avoid errors in resolving dependencies among files and modules. The state changes due to each command could not have been captured through static analysis. Alternatively, the commands and snapshots can be extracted from the Cypress mock browser from the commands list panel and the AUT iframe respectively. For more details on the Cypress test runner, the readers may explore the official documentation. For convenience, all projects are translated to use a specific Cypress version automatically using the Cypress runner, 12.10.0 to be specific in this study.

First of all, Puppeteer needs the port associated with the running mock browser instance to gain control. The port is known by querying the *launchOptions* parameter in *before:browser:launch* event, defined inside the Cypress config file (e.g., cypress.conFigurejs). After establishing a connection, the targeted E2E spec files are run automatically by executing a click from the list of specs. The puppeteer script collects the tests after waiting until the spec has finished execution. Each test case is expanded by clicking to get the list of commands. Then, using appropriate CSS selectors, Puppeteer extracts all the commands. Additionally, Cypress keeps snapshots of AUT before and after the command is executed. Clicking on each command time travels back to that snapshot. Also, it highlights the UI element being operated on, located by the selector in *get* command. The snapshots before and after each command are used to capture - i) the current URL, ii) the screenshot and HTML text of the entire page, and iii) the non-highlighted screenshot and HTML text of the selected element.

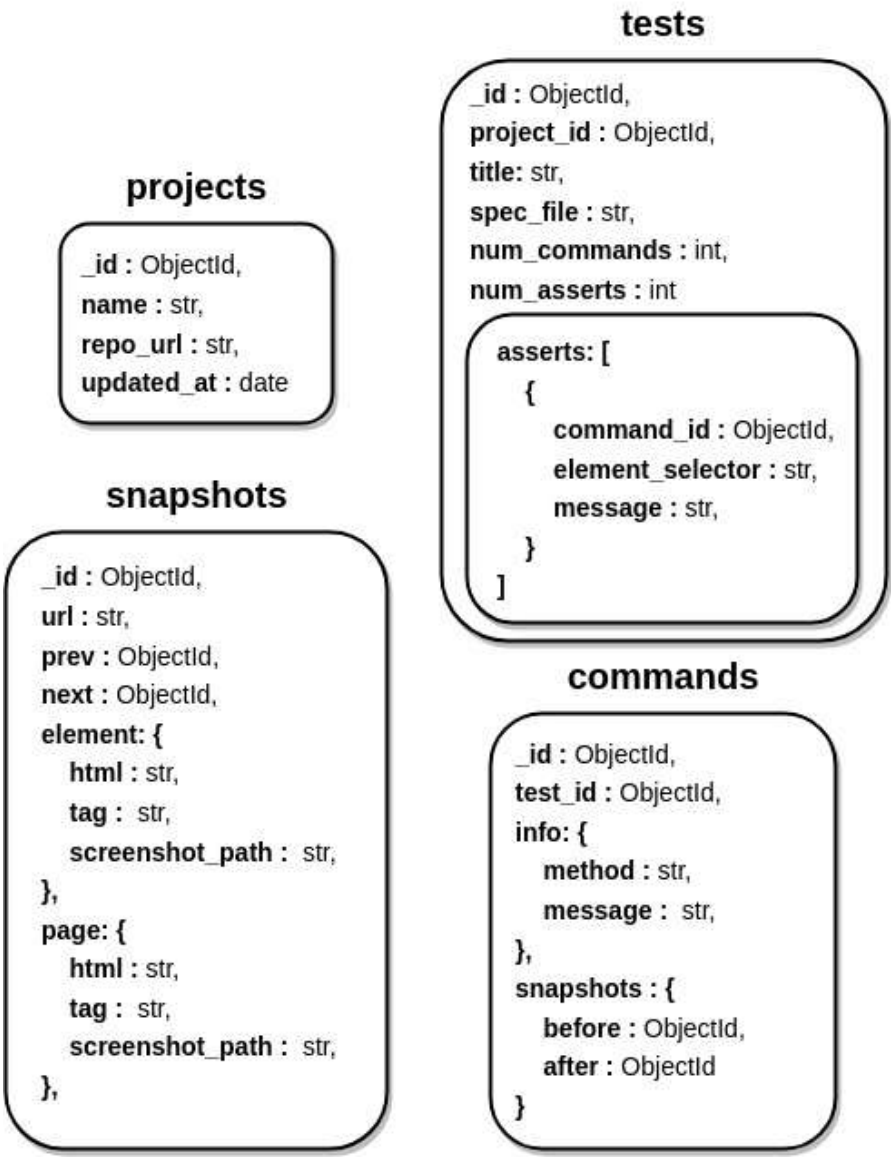


Figure 4.2: WebEV dataset representation

4.2.6 Dataset Representation

WebEV and the Puppeteer script used to generate it are made publicly available [72]. The dataset is represented using mongoDB [73] due to its flexibility and wide usage in software mining research [74, 75]. It stores the textual information and files references for the screenshots, as shown in the schema design in Figure 5.1. Whereas, the actual files are located in separate project-wise folders. WebEV is easily extendable as extracting usage scenarios from a new project just adds the project test command information and screenshots in the corresponding mongo collection and project folder.

4.3 Use Cases

The representation of test cases in WebEV is designed to support a wide range of research applications. Focusing only on Cypress-based test cases may limit the number of projects and tests, but the same methodology can easily be replicated to extend the dataset for other web testing tools. The only difference would be to monitor JS events using Puppeteer rather than extracting command logs provided by Cypress. Extending the dataset will increase the scope for more deep learning-based use cases. The generic data and representation of WebEV motivate three major categories of use cases, as discussed in subsequent subsections.

4.3.1 User Interaction Modeling

In modern web applications, users interact with the interface in a plethora of ways. Practitioners face difficulty designing how users can interact with web elements, despite catalogs of design patterns ([76]) being available [77]. Therefore, user interaction modeling techniques can be used to train on WebEV's UI interactions. They simulate a real user functionality scenario from start to finish including the type of UI interaction (such as click, input, scroll, etc.) and the web element being

operated on. The code and visual representation of the element and the entire page can be used to predict the next user interaction. Moreover, predicting the user's expected actions can optimize systems to pre-fetch resulting resources.

4.3.2 Tester Perception Guidance

Even with the existence of automated test scripts, practitioners frequently need to verify manually whether code changes reflect the evolving requirements. These manual checks can be converted to automation scripts for future use. However, the reuse of such scripts is hindered due to the modern development style of continuous integration. Moreover, preserving domain knowledge to design test cases that verify all possible usage scenarios is very difficult for evolving systems. UI interaction patterns in WebEV can help testers regarding when, which, and how diverse scenarios can be exploited, i.e., provide perceptual guidance. The traces in test cases can be mined to guide practitioners in the following ways:

Prioritize actions: Training the test traces from the large-scale projects in WebEV can model the actionable elements which should be prioritized. Such a tool can aid testers to identify elements that usually need more attention (such as web forms) or should be avoided (e.g., redundant cyclic navigation). Testers can benefit from dynamically adjusted action priority based on learned naturalness in the testing sequence. For example, the action sequence responsible for the creation of an entity can be given initially more priority rather than deletion. Afterward, the priority can be dynamically adjusted to remind exploring alternative actions.

Scenario coverage: E2E testing validates a sequence of actions in which the user follows - i) an expected usage scenario based on requirements, and ii) an unexpected path of execution that should be handled properly or blocked. Based on test scenarios taken from WebEV, testers simulate the expected user behavior of filling out the required information before submitting a form. Alongside, the alternate scenarios are also checked where the user forgot any required field or puts

invalid input such as putting characters in a number field, mismatching password constraint, boundary value analysis, etc. For such scenarios, assertions are made to ensure the user must be unable to submit the form. Testers often fail to test such paths of execution due to the lack of domain knowledge, experience, or time. WebEV solves this gap as it contains the testing behaviors exhibited by testers in large-scale collaborated projects. Furthermore, WebEV can be used to devise a design-based test suggestion tool that captures the subject UI component to derive interaction patterns, reducing the risk of untested and duplicate scenarios.

Assertability: Web automation scripts ensure functional correctness through assertions, such as, asserting whether a particular button is present on a page, an error message is shown on blank input, an entity created/deleted is reflected in the table, and so on. With the change in design and requirements, these assertions also have to be modified. The study has been done to learn meaningful assertions in Java-based unit testing [78], but there is no such work on E2E tests for web applications. The assertions crawled in WebEV can help practitioners understand the current practice in peer-reviewed systems. Practitioners can make queries based on AUT design to suggest related possible assertions. For example, by detecting that the AUT is on the signup page, the tool can suggest assertions like - the signup button's existence, ensure password confirmation match, redirect to the home page after registration, etc.

4.3.3 Automated Testing

The tester's behavior can be learned from WebEV to devise an automated testing tool based on action prioritization, scenario coverage, and assertability. State-of-the-art automated testing tools rely on metrics such as state coverage [5], branch coverage [4], etc. but fail to meet industrial satisfaction. For example, automated tools prefer navigating to diverse pages to increase coverage rather than submitting a partially filled web form, i.e., fulfill a complete usage scenario. Learning

the interaction patterns from real-world testing scenarios can improve automated testing. Moreover, the state explosion problem can be reduced as testers avoid test scenarios representing duplicated functionality. For example, if a table represents a collection of completely similar entities, testing the delete functionality of just one of them is sufficient.

4.4 Summary

This paper introduces WebEV (publicly available at [72]), an E2E testing dataset encapsulating interactions done to validate diverse usage scenarios of web applications. Although WebEV is in the initial stage, it can be easily extended for any web application using the same methodology. It requires a one-time effort of running the E2E test cases to extract the interaction scenarios. Cypress E2E tests from popular 100 open-source projects are executed to replicate how testers manually test diverse scenarios of AUT. Later using the Puppeteer library, test commands are extracted along with representations capturing state changes on each interaction. Practitioners can utilize this dataset to understand how large-scale practitioners exploit diverse scenarios. Also, this dataset will help researchers improve automated testing approaches taking into account the tester's perception.

Chapter 5

eBAT: An Efficient Automated Web Application Testing Approach Based on Tester's Behavior

Web application failure detection relies mostly on the tester's creativity, leaving test automation to only ease executing repetitive tasks. Existing automated testing techniques opt for test path diversity or input generation but not the tester's behavioral patterns. For example, testing deeply nested business logic, proper form submission, or non-redundant navigation are not considered. This paper proposes eBAT, an automated testing approach that considers those testers' interaction patterns from observation. A behavior-driven action selection strategy is derived from these patterns to interact with the system. Actionable elements (buttons, links, inputs, etc.) obtained through state abstraction and interaction pattern-wise grouping are operated in a tree-based manner. The effectiveness and efficiency of eBAT are evaluated as the unique number of failures detected and the detection rate respectively. Results compared against the state-of-the-art indicate

2.2 times improvement in failure detection on average with similar code coverage. Moreover, eBAT outperforms the baseline approach within 5 minutes of execution in 5 out of 6 benchmark open-source projects.

5.1 Introduction

The previous chapter proposed the WebEV dataset containing interaction patterns of testers from automation scripts. These analyzed patterns are targeted to improve the state of automated web testing. In manual web application testing, a tester always tries to test a complete functionality at a time. They design test cases in such a way that fulfills each of the unique functionality. Their goal is to detect bugs with minimum redundant action execution [2] (such as clicking the same link multiple times is redundant). These common practices can be defined as the tester's behavior. Automated web testing approaches [3, 4] focus mostly on testing metrics like code, branch coverage or path diversity, etc., which may fail to explore the complete functionality (for example, filling the inputs without submitting the form). As a result, these approaches may lead to partial functionality execution that are ineffective to exploit complete usage scenarios.

An automated testing approach based on the tester's behavior can efficiently test a complete functional action sequence, combining the benefits of both manual and automated testing. However, determining such human-level behavioral patterns is challenging as manual testing practices may vary based on the tester or organization. A generic set of behavior needs to be designed such that it is applicable to any web application. These patterns might be obtained from WebEV by examining manually written automation test scripts. Moreover, a mechanism needs to be devised to apply these behaviors in an automated approach. The resulting set of behaviors incorporated into an automated testing approach may make the failure detection process efficient.

Redundancy reduction strategies and widget-specific behavior is applied to automated Android GUI testing [79, 80, 81]. However, similar behavioral strategies have not been adopted in the case of automated web application testing. Since web application interaction patterns are different, similar policies cannot be adopted for web-based systems. To gain practical reliability and reduce the need for domain knowledge, automated testing tools need to take into account the behavioral characteristics exhibited by a tester. The test cases automatically generated should be behaviorally similar to steps written in testing scripts.

This chapter proposes an **e**fficient **B**ehavior based **A**utomated **T**esting approach (**eBAT**), incorporating strategies employed in manual testing such as non-redundant exploration and effective action interaction (for example, non-cyclic navigation, form submission, etc.). The behavioral patterns considered in this approach do not represent the entire set of tester's interaction patterns, but rather a subset observed in manually written test cases. These test cases were extracted from WebEV, that contain test interactions from popular open-source projects representing the most popular frontend web frameworks. These patterns were used to devise an action selection strategy, focusing on - i) non-redundant actions in **E**xecution **T**race (**ET**) and ii) interdependent action grouping.

5.2 Methodology

This paper proposes eBAT, incorporating the tester's behavioral patterns in model-free testing. By doing so, it minimizes execution having no impact on failure detection. The entire approach is divided into four components, namely - State Abstraction, Behavioral Grouping, Redundancy Reduction, and Exploration Completeness, as indicated in Figure 5.1.

State Abstraction. The underlying representation (such as HTML page, screenshot, etc.) of a web application at a particular time is its state. To achieve

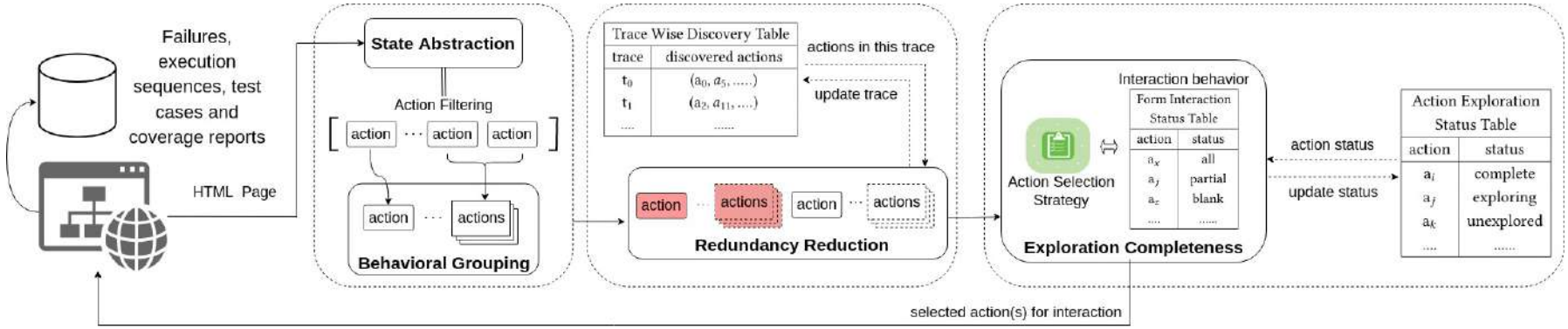


Figure 5.1: Overview of proposed behavior-based automated testing approach

the required functionality, states contain a set of operable elements such as links, buttons, input fields, dropdowns, etc. The interaction with such elements is called actions in web testing. These interactions may lead to a change in the URL or functionality of the current DOM. An exponentially increasing number of states affects the efficiency of test suite generation [43]. This problem is solved by employing state abstraction to detect functionally similar states, reducing redundant testing of the same behavior. This promotes exploring diverse scenarios.

Motivation for the abstracted state calculation process has been taken from [5]. On each ET, the approach starts by loading the home page of AUT (URL provided as a parameter) and extracts the HTML representation. The sequence of HTML tag names along with corresponding nested attributes is listed recursively. Digits within the attributes are removed for consistency as many modern front-end frameworks dynamically generate attribute IDs on each page load. The Gestalt pattern matching algorithm is used to calculate the similarity of the listed sequence after each change in the application representation. However, to differentiate the abstracted states a threshold of 0.8 is used based on prior work [5]. The resulting state is considered unique if - i) the difference between the two states is greater than the predefined threshold, or ii) the URLs of the two states are different. The operable HTML tags such as $\langle button \rangle$, $\langle a \rangle$, $\langle input \rangle$, etc. are parsed and mapped as the state actions. The visible actions are filtered using the area bounded by the element's box model, as testers only interact with visible actions.

Behavior Identification. A set of most commonly employed testers' behavior is needed for an efficient action section strategy. Since no prior work lists such behaviors, manually written front-end automation test suites captured in WebEV are analyzed to identify them. The open source community is preferred for baseline project selection due to public availability and to avoid organization or developer-specific behaviors. The open-source projects in WebEV have - i) GitHub star count ≥ 50, and ii) automation test scripts mocking the tester's interactions

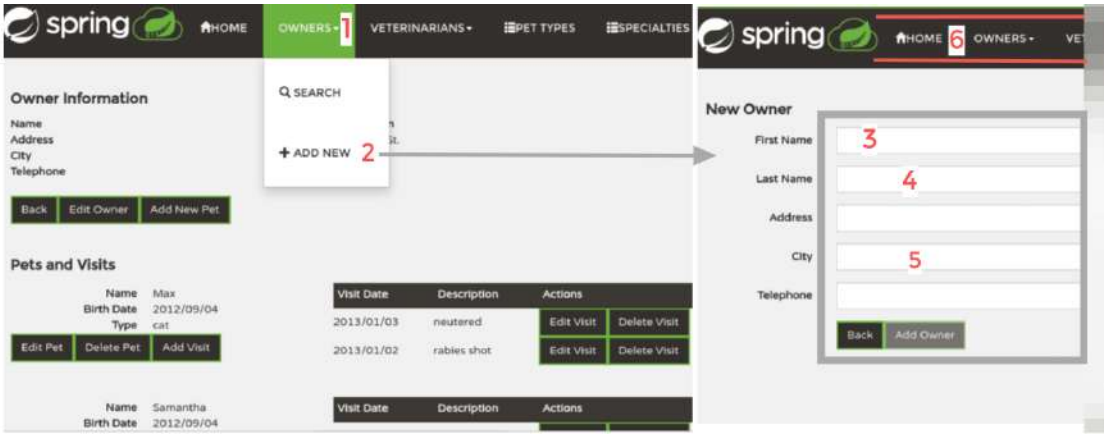


Figure 5.2: Petclinic sample project (ineffective execution scenario *owner-detail* -> *add-owner* illustrated by actions 1-6)

available. An entire list of patterns is not identified and is left for future work. For now, the patterns exhibited in all projects are identified from the hypothesis made by eyeing the interactions. The subset of the tester's behaviors identified is - i) executing non-redundant actions in a particular trace with non-cyclic navigation, ii) proper form submission, and iii) form submission with complete or partial input. These behaviors are queries in the WebEV dataset to confirm their existence in all scenarios. For example, querying test commands where form interactions are done revealed proper submission of forms to ensure functionality. Moreover, the script to check the absence of duplicated action in a single test trace confirms the identified patterns. These behaviors identified upon querying the WebEV dataset are incorporated in eBAT to devise the exploration strategy.

The Petclinic open-source project, as shown in Figure 5.2, is used to demonstrate the impact of the identified behaviors during testing. Petclinic contains most of the failures in functionalities accessible only from the deeply nested *owner-detail* page [5]. Suppose a tester reaches this page and instead of focusing on its functionalities, follows the action sequence as shown in Figure 5.2. This would result in a redundant ineffective execution. Employing the identified behaviors in this scenario would - i) prevent cyclic navigation (Actions 1-2) since navbar actions are present on all pages, ii) prevent ineffective form interaction (Actions 3-6), and iii)

test form submissions with missing data (e.g., exploiting forms in *add-visit* page)

Behavioral Grouping. The process of grouping actions that act interdependently is termed behavioral grouping in eBAT. Actions are grouped in eBAT that collaborate in a specific sequence to achieve a business objective. Testing these functionalities, requiring the selection of such interdependent actions would otherwise rely on "luck". Web forms are considered as the most naturally encapsulate interdependent nested actions (input, dropdown, radio button, submit button, etc.[55]) to complete the intended functionality. The form encapsulated inputs and buttons are combined to always interact together as a compound action during on-the-fly testing. This grouping is done by determining the actions residing within a $\langle form \rangle$ element in the current state. These composite actions are added to the set of valid actions in the state, making actions inside the form no longer defined or executed individually. The detected submit button (using type="submit" attribute) is always executed at the end of different possible input action interactions in the case of composite form actions. This pattern of interaction ensures proper form submissions and avoids unnecessary interaction with input actions separately. The behavioral grouping of table rows and list items are not considered as the diversity in the content may exploit different failures, for example, the *owner-detail* page with and without pets need to be tested in Petclinic open-source project. Better state abstraction can classify unique rows and list items, which is out of scope from this current study.

Redundancy Reduction. Employing state abstraction and action grouping algorithms is not sufficient for efficiency as the actions are often duplicated [44]. Tracking redundant actions within a single trace enables the exploration of the system through diverse paths. For instance, the actions in the layout components (navbar, sidebar, and footer) are present in multiple states. Such actions, although usually unchanged in their functionality, are marked as valid actions in each state. The comparison of actions is done based on the action's tag and attribute rep-

resentation used to calculate the state similarity. An action is called redundant if it is encountered previously in the trace [2]. On encountering a new state in a particular trace, actions discovered in the current ET are tracked and redundant actions are filtered (as depicted in Figure 5.1). The resulting set of actions within the new state is considered for interaction in that particular trace.

Exploration Completeness. Web applications rely on a State-Flow Graph-based navigation model [34, 4] where the nodes and edges represent various states of AUT and transitions between states due to action events respectively. This model can be updated dynamically based on business logic [5], for example, the same URL containing a different set of actions based on authorization. The discovery of diverse states in AUT is devised using tree-based traversal of the dynamic navigation model [5]. This dynamic model is constructed using the states as nodes discovered through exploration. Edges indicate transitions $\langle s, a, s' \rangle$ between states using action a from state s. Depth-first traversal is used to prioritize deeply nested states that require a specific sequence of interaction [82]. Moreover, this strategy ensures incremental discovery of all states and tests the completeness of a particular path. Each unique functionality is fully exploited as long as the business logic do not expand over multiple diverse paths, which is not possible in literature without domain knowledge.

Instead of backtracking to immediate parent states, the traversal is done starting from the root state for every trace. This is because browsers cannot store the history information of previous states in AJAX based dynamic web applications. On each change, the state abstraction mechanism is employed to match similarity and extract executable actions. Afterward, the actions are grouped and redundant actions are removed through the behavioral grouping and redundancy reduction steps respectively. Actions are executed automatically based on the element's type (using Puppeteer [83]), such as clicking links or buttons, filling up input fields with random values based on the type attribute, etc. Actions leading

to external resources are blacklisted from further interaction.

Initially, starting from the root page, all actions are marked as "unexplored". An unexplored unique action is selected at random from the current web state and marked "active". This action exploration state mapping is maintained across ETs. For a different trace starting from the root, the exploration strategy needs to continue from the previous active action sequence. Partially explored actions are marked "complete" recursively when no actions in the resulting state are left incomplete. On the other hand, grouped form actions are marked as complete after considering possible interaction patterns - i) all inputs left blank, ii) interact with all inputs, and iii) interact with a subset of inputs. Finally, the entire process is repeated when all actions from the root state attain exploration completeness. The repetition promotes the exploration of new actions in nested pages that might be unavailable before, for example, the delete functionality is exposed only after successful creation of any entity.

5.3 Evaluation and Results

To evaluate the effectiveness and efficiency in terms of failure detection, eBAT is compared against the state-of-the-art WebExplor [5], as it outperforms other model-based, random, and model-free automated testing approaches. Upon encountering a failure during trace execution (tracked via browser console errors), it is added to the set of failures unique to each state. The sequence of actions taken to exploit the failure is stored as a failing test case. Both approaches were compared using the same environment, parameters, failure definition, initial login script, and AUT version. Code coverage is measured in both approaches using a Python wrapper of Google's Puppeteer library [83]. The source codes and results of both eBAT and WebExplor are made publicly available [84].

Table 5.1: Benchmark open-source project statistics

Subject Systems	**Framework**	**Commit ID**	**#Stars**	**#Commits**	**#Forks**	**LOC (JS/TS)**
dimeshift	Backbone.js	440898e	185	204	46	9365
pagekit	Vue.js	6fc7aa1	5507	4933	670	4374
splittypie	Embere.js	da052e6	136	350	19	4961
phoenix-trello	React-Pheonix	60c874d	2496	422	424	2339
retroboard	React	d79047c	709	931	189	39469
PetClinic	AngularJS	b2e12be	183	224	450	4973

Table 5.1 lists the benchmark projects used in literature [4, 5], sampled from the most popular web frameworks based on GitHub stars. To align with the baseline approach [5] and remove any statistical bias, both approaches were run 15 times with a defined time budget of 30 minutes and a state abstraction threshold of 0.8. Functionally duplicate failures were filtered manually to get the unique number of failures, for example, in Splittypie, the same errors caused by separate transaction entities were ignored. The evaluation was carried out with the following questions:

Effectiveness: *How effective is eBAT in terms of failure detection in web applications?* The effectiveness of automated web testing is measured by the unique number of failures detected and code coverage [4, 5]. Table 5.2 shows the comparison in which eBAT significantly outperforms WebExplor in 4 out of 6 projects separately using the Mann-Whitney U test at 0.05 confidence level. The experimentation choices are made similar to [4, 5].

Failures located in deeply nested pages in these projects are exploited by eBAT through non-redundant exploration and proper form submission. For instance, in Petclinic, exploiting the forms (add/edit pets and add/edit visits) in the deeply nested *owner-detail* page. Detecting such failures is difficult for WebExplor due to the randomness in the action selection policy, even with high-level DFA guidance. An exceptional case is observed in Dimeshift where WebExplor is able to detect "waking up" script errors due to redundant navigation to *wallet-detail* page. As shown in Table 5.2, WebExplor attains more code coverage in Phoenix whereas eBAT fails to explore them focusing on depth-first traversal. However, eBAT can detect more failures due to its behavior-driven action selection strategy as failure detection is not dependent on code coverage [5]. Therefore, eBAT is evaluated to be more effective in terms of failure detection compared to the baseline.

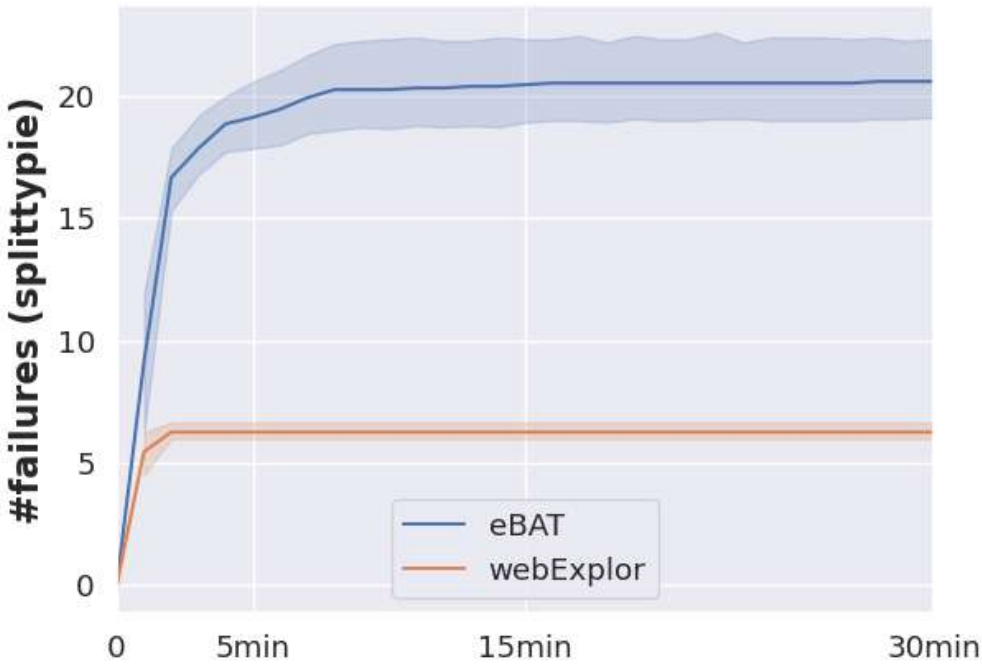
Efficiency: *How efficient is eBAT in exploiting web applications?* The efficiency is measured in terms of failure detection rate similar to [5]. The plots in Figure 5.3 demonstrate the comparison in terms of efficiency where the x-axis

Table 5.2: Comparison of failure detection effectiveness. (Values in bold and star indicate best average results along with standard deviations over 15 trials and statistically significant differences respectively)

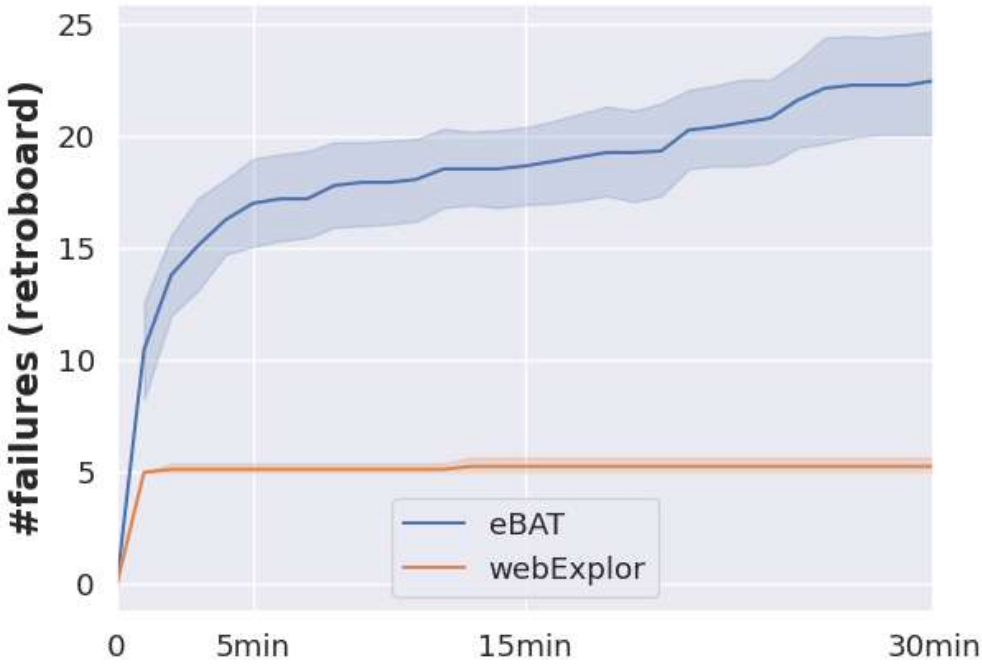
Projects	Unique Failures (#)		Code Coverage (%)	
	eBAT	WebExplor	eBAT	WebExplor
Dimeshift	6.9 (1.3)	**8.3** (1.4)	**0.9*** (0.6)	0.3 (1.4)
Pagekit	**10.2*** (4.6)	7.5 (1.2)	**90.2** (1.1)	89 (8.8)
Splittypie	**20.6*** (3.4)	6.3 (0.7)	**65.1*** (0.7)	57 (0.04)
Phoenix	**3.9** (2.5)	3.2 (0.4)	50.4 (3.3)	**50.9** (0.04)
Retroboard	**22.5*** (4.4)	5.3 (0.7)	**51.7** (7.1)	51.3 (9.1)
Petclinic	**43*** (9.9)	18.2 (1.8)	**45.8*** (0.3)	41.2 (1.2)
Average	17.8	8.1	50.7	48.3

indicates the time elapsed and the y-axis indicates the number of unique failures detected. Based on the plots, eBAT outperforms WebExplor in 5 out of 6 (exception for Figure 5.3f) projects within 5 minutes of execution. Exploring deeply nested pages results in an early jump in discovery rate just within 5 minutes in Pagekit (Figure 5.3e), Petclinic (Figure 5.3c), Splittypie (Figure 5.3a), and Retroboard (Figure 5.3b). Proper form submission saves time by avoiding unnecessary execution resulting in the faster discovery of nested update functionalities in Trello (Figure 5.3d). Errors in Splittypie are discovered only after the creation of a transaction event and navigating to its details.

WebExplor fails to exploit these scenarios in most cases because - i) proper form submission gets interrupted by other actions, and ii) selects actions that cause redundant exploration, even after the transition to less-visited areas using DFA guidance. After DFA transition to less visited states, the action policy results in ineffective exploration. For example, in Splittypie [85] DFA transition to deeply nested pages and then focusing on navbar actions make this transition ineffective. Therefore, eBAT is more efficient to discover failures, employing the design decision of redundancy reduction, behavioral grouping, and tree-based exploration.

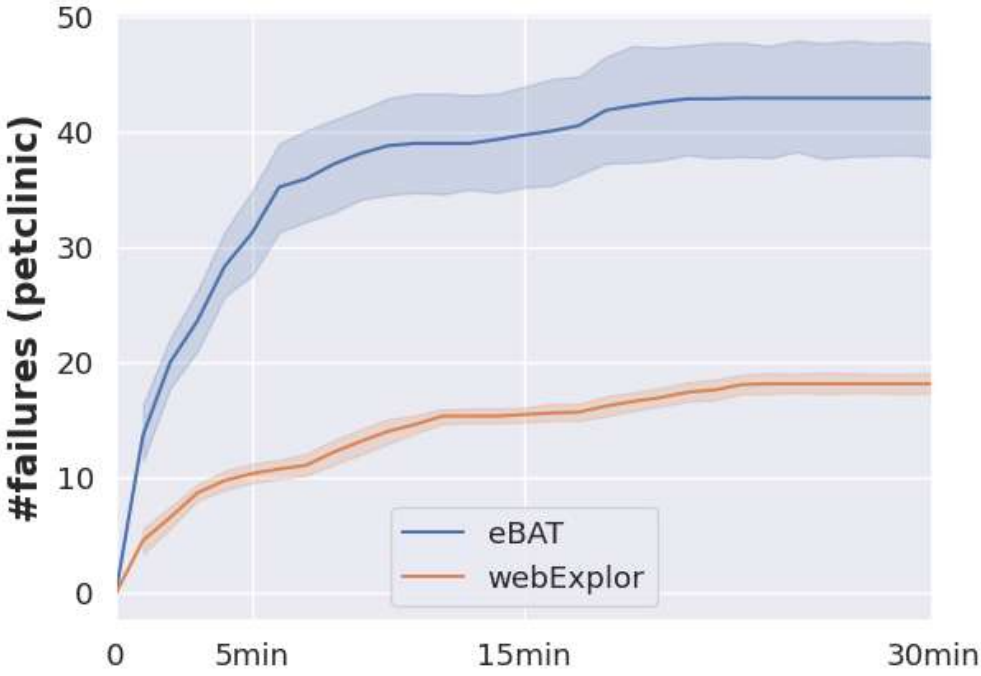


(a) Failure detection rate for Splittypie [85]

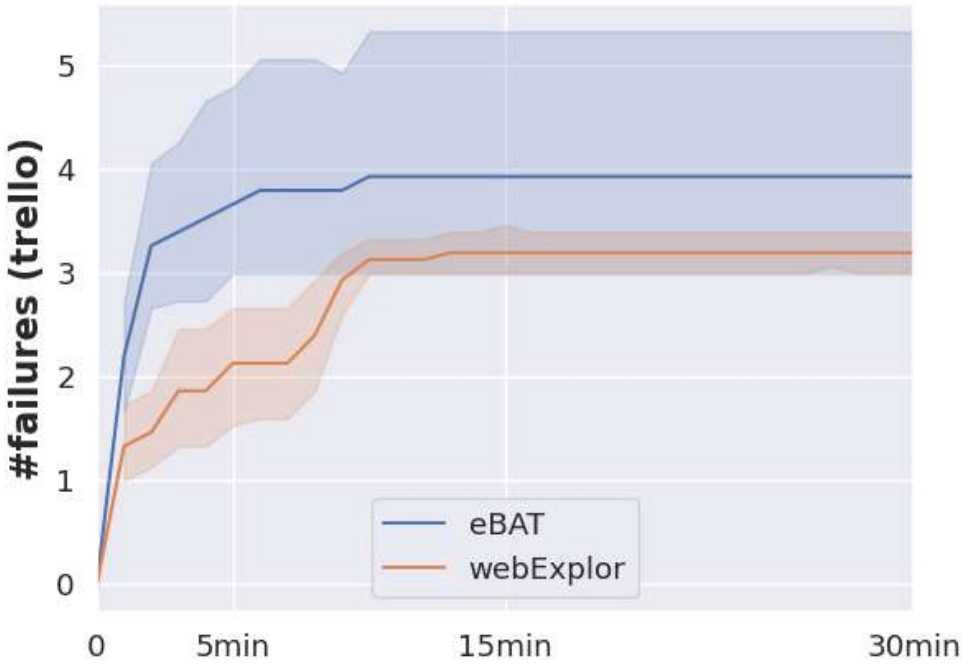


(b) Failure detection rate for Retroboard [86]

Figure 5.3: Comparison of failure detection rate. Shaded areas represent lower and upper bounds along with their average

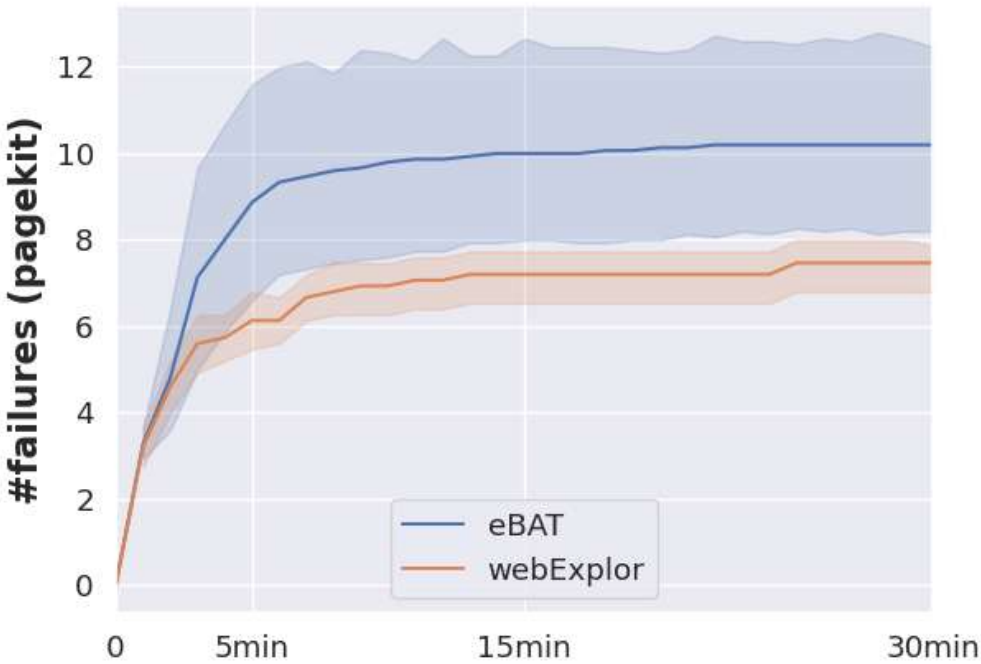


(c) Failure detection rate for Petclinic [1]

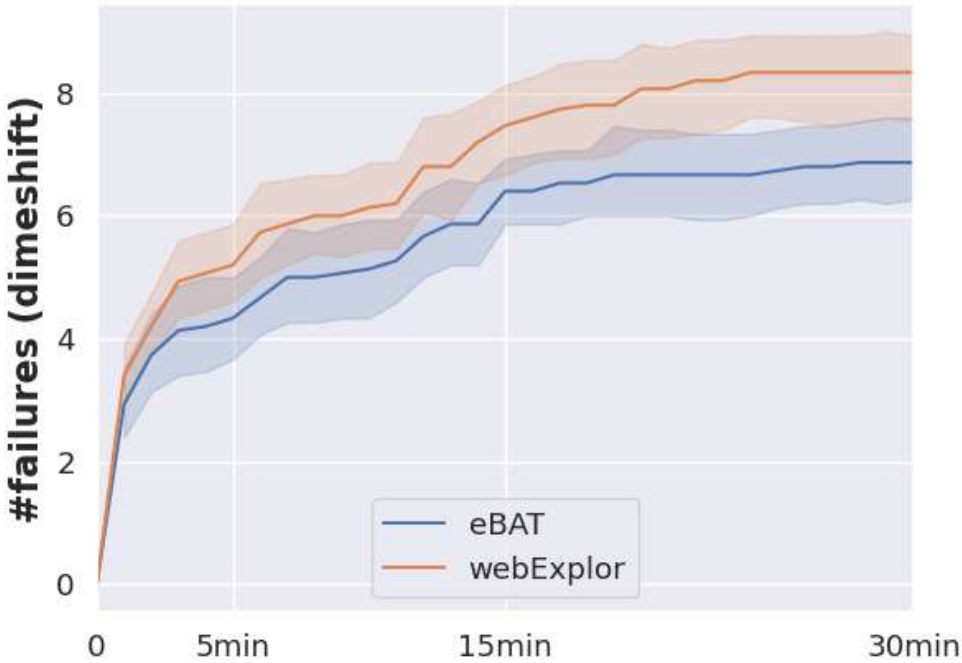


(d) Failure detection rate for Trello [87]

Figure 5.3: Comparison of failure detection rate. Shaded areas represent lower and upper bounds along with their average (cont.)



(e) Failure detection rate for Pagekit [88]



(f) Failure detection rate for Dimeshift [89]

Figure 5.3: Comparison of failure detection rate. Shaded areas represent lower and upper bounds along with their average (cont.)

5.4 Threats to Validity

Evaluation on a limited number of projects (taken from prior work [4]) poses an *external validity* threat as results may vary based on projects. To mitigate this threat, open-source projects from the top 6 popular frontend frameworks are taken for generalizability. The selected behaviors may have some confounding variables that are not considered in this preliminary study, affecting the *internal validity*. However, these patterns are identified from popular open-source projects to remove developer-specific practices. The most common behaviors are considered in this paper but there is scope to incorporate other behaviors that may change the results. The replication package [84] is provided for evaluation reproducibility.

5.5 Summary

The approach presented in this paper, eBAT, bridges the gap in automated testing by incorporating the tester's behavioral decisions taken during manual testing, such as non-redundant exploration and grouped action interaction. This devised action selection strategy executes non-redundant actions to generate the execution trace. Redundancy reduction and behavioral grouping of interdependent actions significantly improve failure detection against the state-of-the-art approach. Results show that even simple behavior-driven decisions significantly outperform the current state-of-the-art in the effectiveness and efficiency of failure detection. Thus, focusing on enriching the set of tester's behavior rather than metric-based strategies can make failure detection more efficient. Moreover, employing such strategy in exploration promotes not only failure detection but coverage metric as well. These results motivate our future plan to include the investigation of behavioral aspects other than non-redundant exploration and random form interaction, such as avoiding similar entities and form validation.

Chapter 6

Conclusion

Web applications have always drawn the researcher's attention due to the challenges imposed by them. Along with the business-specific challenges, the web also enables a rich set of user interactions and visuals. This makes the task of automated web testing approaches difficult to completely validate system functionalities. Although much research effort has been given into it, still these techniques cannot deliver human-like interactions. This is because fully automated techniques rely on coverage metrics such as code coverage, branch coverage, state coverage, etc. to evaluate the effectiveness of testing. Furthermore, action selection strategies result in redundant and ineffective execution. On the other hand, techniques that incorporate the tester's domain knowledge (model-based) are able to exercise the functionalities, given that the tester has good knowledge of the subject system. However, these techniques prove to be time-consuming, error-prone, and may generate infeasible test cases. In this context, this thesis emphasizes the impact of incorporating the manual tester's behavioral aspects in a fully automated approach. At first, the WebEV dataset is proposed that contains interaction sequences extracted from Cypress-based end-to-end test scripts. Next, based on the hypothesis, the most predominant behavioral patterns are examined. An automated approach eBAT is then proposed that considers these interaction patterns

to exercise the web application under test. This chapter draws the conclusion of this thesis, summarizing the contributions and achievements. Lastly, the onward research directions are listed to improve the state of automated web application testing.

6.1 WebEV: A Dataset on the Behavior of Testers for Web Application End-to-End Testing

A dataset encapsulating the tester's behavioral patterns is devised at first. This dataset, WebEV, works as the seed to derive the tester's patterns during manual testing and propose a behavior-driven automated web testing approach. Cypress-based test automation scripts are used to extract interaction sequences from test execution. This is because recording testers during manual testing can be subject to subjectivity in selected applications and the tester's skills. In addition, such a method is neither reproducible nor automated. GitHub repositories are filtered based on their popularity (counting GitHub stars and commits to ensuring maturity) and their usage of Cypress-based end-to-end test scripts. From this, the top 100 repositories are selected similar to prior work [70]. A manual step is needed to compile these projects after resolving system dependencies. Running their respective Cypress tests exposes the list of commands and assertions representing the way human testing would commence testing. Using the Puppeteer library these commands and assertions are crawled. Additionally, the changes due to each command in AUT are stored through snapshots saved by Cypress. Finally, the interaction logs are saved in a MongoDB database along with snapshots of AUT changes and the target element of interaction.

The WebEV dataset is not only structured to aid automated testing, rather a generic representation that establishes future research direction in fields such as user interaction modeling and tester's perception guidance. User interaction

modeling techniques can be used to train on WebEV's UI interactions as they simulate a real user functionality scenario from start to finish including the type of UI interaction (such as click, input, scroll, etc.) and the web element being operated on. Moreover, these techniques can also be used to predict the user's expected actions that further optimize systems to pre-fetch resulting resources. From a tester's perspective, preserving domain knowledge to design test cases for all possible usage scenarios is very difficult, especially for rapidly evolving systems. The interactions in WebEV can be mined to guide testers in prioritizing actions, exercise diverse scenarios, and determine assertion best practices for behaviorally similar functionality. These techniques can improve the current state of automated approaches and increase their adoption among practitioners.

6.2 eBAT: An Efficient Automated Web Application Testing Approach Based on Tester's Behavior

Based on the resulting dataset, this thesis proposes a fully automated web application testing approach. This technique contributes to the empirical evaluation that integrating the tester's actual interaction patterns leverage better failure detection against current coverage metric-based approaches. The approach takes the URL of the running web application in the browser and automates UI interactions to detect failures. The failures can be detected from the browser console either due to frontend or backend malfunction. The entire approach is divided into four components - i) state abstraction, ii) redundancy reduction, iii) behavioral grouping, and iv) exploration completeness. The approach starting from the root page of AUT employs state abstraction to group behaviorally similar states employing a state abstraction algorithm. This in turn minimizes the state explosion problem

in modern AJAX-based web applications. Then the DOM tree is parsed to filter actionable elements from their corresponding HTML tags (actions such as $\langle A \rangle$, $\langle INPUT \rangle$, $\langle BUTTON \rangle$, etc.). Next, actions, repeated from predecessor states, are filtered and treated as unique actions that induce a high change of cyclic navigation or ineffective execution. Actions that co-interact to perform a specific functionality are grouped (web form action in our case) and the resulting set of actions are exercised in a depth-first manner. This process is done until the time budget (set as a parameter for testing) expires.

Given an infinite amount of time, all automated techniques are able to exercise the full AUT functionality theoretically. However, practitioners have time and resources for this purpose. To evaluate the effectiveness and efficiency of eBAT, empirical evaluation is done on six open-source benchmark web applications. These projects have been selected from different popular frontend frameworks to avoid subjectivity. Results are compared with the state-of-the-art in the literature using the same parameters such as - time budget, failure definition, login scripts, etc. Results show that even simple behavior-driven decisions significantly outperform the current state-of-the-art in the effectiveness and efficiency of failure detection. The proposed approach can successfully outperform five out of 6 benchmark projects within just $1/6^{th}$ of the total execution time. Furthermore, the evaluation is statistically tested significantly using the Mann–Whitney U test. Thus, focusing on enriching the set of tester's behavior rather than metric-based strategies can make failure detection more efficient.

6.3 Future Work

This thesis paves the way to incorporate the tester's behavioral patterns in an automated testing approach. Much more work can be done to validate this area of knowledge and further extend it. The future plans of this work are listed below:

- **Extend scenario collection in WebEV:** Cypress-based end-to-end test cases are extracted from open-source projects as Cypress is the most popular automation tool based on GitHub stars. However, a more generic method needs to be developed to extract test cases from alternate automation tools like Selenium, Playwright, and other future tools. Expanding the dataset will either uncover more fine-grained patterns or confirm the saturation of patterns (that is, no other interaction patterns are commonly used). Besides, an extensive dataset can enable the application of deep learning algorithms to simulate the tester's interaction. Deep learning approaches based on large-scale tester usage traces are still unexplored. Humanoid [64] leverages crowdsourced user interactions but these interactions cannot represent the tester's goals. This gap can be bridged using WebEV that takes into account the tester's perspective during manual testing interactions.

- **Extract testers behavior automatically:** In this study the listed tester's behaviors - i) non-redundant execution, ii) non-cyclic navigation, and iii) proper form submission were based on prior hypothesis after running Web-Explor [5]. Besides, these hypotheses were confirmed by informal interview sessions with practitioners. Later the existence of these prominent patterns is inspected from the WebEV dataset. A more comprehensive list of the tester's behavior patterns can be obtained through an automated pattern-mining process, not done in this thesis.

- **Incorporating the derived behaviors in eBAT:** Incorporating all the derived behavioral patterns in eBAT is challenging. Domain-specific functionalities, for example, checking transactions/notifications between multiple parties with proper login are very difficult. Automated approaches are known to suffer from a lack of domain knowledge. Pattern derived can be checked via devising invariants at best, as in ATUSA [25]. This thesis claims that

naturalness in interactions makes automated approaches for efficient than focusing on code metrics. This can be further clarified by considering other patterns to remove the possibility of confounding effects.

- **Investigate factors causing deviation from human-like testing:** Model-free behavior-driven testing brings a new paradigm in automated web testing. Therefore it is necessary to understand specific scenarios in which eBAT and other approaches differ. This evaluation can be done without eBAT just considering the interaction patterns derived from WebEV. This in turn can improve eBAT by identifying exploration/decision factors taken during manual testing. Besides, concrete scenarios can be derived to claim efficiency through behavior-driven decisions rather than code metric improvement.

Although automated web testing is an established area of research, incorporating the tester's behavioral aspects is in its primary stage. Techniques in literature try to prove functional exploration completeness by manually defining the application state space or opting for coverage maximization. Domain knowledge is required for functional exploration and the way testers interact in certain scenarios. This work tries to minimize the gap in literature by proposing WebEV, a dataset containing such interaction patterns. Moreover, some of the identified patterns are incorporated into an automated approach, eBAT. By no means eBAT exhibits all the patterns during manual testing, rather shows that considering behavioral patterns yields better effective and efficient in terms of failure detection compared to metric-based exploration. In addition, this approach can be much improved by detecting the entire set of tester's interaction patterns utilizing the WebEV dataset. Further research can also enable deep learning algorithms to model the tester's behavior by extending and employing WebEV.

Bibliography

[1] Petclinic, "Angular petclinic web application," 2017.

[2] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 291–305, Springer, 2007.

[3] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models.," in *ISSRE*, vol. 13, pp. 278–287, 2013.

[4] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "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*, pp. 142–153, 2019.

[5] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 423–435, IEEE, 2021.

[6] S. E. Sprenkle, L. L. Pollock, and L. M. Simko, "Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour," *Software Testing, Verification and Reliability*, vol. 23, no. 6, pp. 439–464, 2013.

[7] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 881–891, IEEE, 2012.

[8] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Apogen: automatic page object generator for web testing," *Software Quality Journal*, vol. 25, no. 3, pp. 1007–1039, 2017.

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

[10] F. Ricca, M. Leotta, and A. Stocco, "Three open problems in the context of e2e web testing and a vision: Neonate," in *Advances in Computers*, vol. 113, pp. 89–133, Elsevier, 2019.

[11] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*, pp. 18–32, Springer, 2017.

[12] "Crawljax," 2017.

[13] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 67–78, 2014.

[14] A. Arora and M. Sinha, "Web application testing: A review on techniques, tools and state of art," *International Journal of Scientific & Engineering Research*, vol. 3, no. 2, p. 1, 2012.

[15] Q. Wang, L. Quan, and F. Ying, "Online testing of web-based applications," in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, vol. 3, pp. 166–169, IEEE Computer Society, 2004.

[16] S. Kundu, "Web testing: tool, challenges and methods," *IJCSI International Journal of Computer Science Issues*, vol. 9, no. 2, pp. 1694–0814, 2012.

[17] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5, pp. 78–90, IEEE, 2016.

[18] C. Rossi, "Moving to mobile: The challenges of moving from web to mobile releases," *Keynote at RELENG*, 2014.

[19] T. Fitz, "Continuous deployment at imvu: Doing the impossible fifty times a day," *Blog Post, February*, vol. 10, 2009.

[20] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, 2002.

[21] J. Dominguez, "The curious case of the chaos report 2009," *Project Smart*, pp. 1–16, 2009.

[22] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[23] BIDA, "Bangladesh it ites industry." `https://bida.gov.bd/information-technology`, 2023. [Accessed 11-May-2023].

[24] Wikipedia, "Information technology in Bangladesh - Wikipedia — en.wikipedia.org." `https://en.wikipedia.org/wiki/Information`$_{\text{t}}$`echnology`$_{\text{i}}$`n`$_{\text{B}}$`angladesh`, 2023. [Accessed 11-May-2023].

[25] A. Mesbah, A. Van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2011.

[26] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pp. 845–854, 2017.

[27] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," in *Advances in Computers*, vol. 101, pp. 193–237, Elsevier, 2016.

[28] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimaraes, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019.

[29] F. Ricca and A. Stocco, "Web test automation: Insights from the grey literature," in *International Conference on Current Trends in Theory and Practice of Informatics*, pp. 472–485, Springer, 2021.

[30] A. Berson, *Client/server architecture.* McGraw-Hill, Inc., 1996.

[31] "Iso/iec/ieee international standard - systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[32] N. Chauhan, *Software Testing: Principles and Practices.* Oxford university press, 2010.

[33] A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi, "Conceptual modeling and code generation for rich internet applications," in *Proceedings of the 6th international conference on Web engineering*, pp. 353–360, 2006.

[34] A. Mesbah and A. Van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 210–220, IEEE, 2009.

[35] L. Nagowah and K. Kora-Ramiah, "Automated complete test case coverage for web based applications," in *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions)(ICTUS)*, pp. 383–390, IEEE, 2017.

[36] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Comparing the maintainability of selenium webdriver test suites employing different locators: A case study," in *Proceedings of the 2013 international workshop on joining academia and industry contributions to testing automation*, pp. 53–58, 2013.

[37] A. Van Deursen, "Beyond page objects: Testing web applications with state objects: Use states to drive your tests," *Queue*, vol. 13, no. 6, pp. 20–37, 2015.

[38] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 272–281, IEEE, 2013.

[39] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *2013 ieee sixth international conference on software testing, verification and validation workshops*, pp. 108–113, IEEE, 2013.

[40] R. K. Yandrapally and A. Mesbah, "Fragment-based test generation for web apps," *IEEE Transactions on Software Engineering*, 2022.

[41] V. Garousi, A. Mesbah, A. Betin-Can, and S. Mirshokraie, "A systematic mapping study of web application testing," *Information and Software Technology*, vol. 55, no. 8, pp. 1374–1396, 2013.

[42] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*, pp. 36–42, IEEE, 2012.

[43] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 186–197, 2020.

[44] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th international conference on Software engineering*, pp. 571–579, 2005.

[45] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[46] R. Binder, *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[47] C.-P. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 81–90, 2009.

[48] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 121–130, IEEE, 2008.

[49] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[50] A. Van Deursen, "Testing web applications with state objects," *Communications of the ACM*, vol. 58, no. 8, pp. 36–43, 2015.

[51] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[52] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[53] S. 2023, "Seleniumhq web browser automation." `http://www.seleniumhq.org`. (Accessed on 02/07/2023).

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[55] mozilla.org, "The input (form input) element," 1998.

[56] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[57] E. Jang, S. Gu, and B. Poole, "Categorical reparametrization with gumble-softmax," in *International Conference on Learning Representations (ICLR 2017)*, OpenReview. net, 2017.

[58] A. Inc., "Top web sites rank list." `https://www.alexa`, 2020. (Accessed on 02/05/2023).

[59] O. Korosteleva, *Nonparametric methods in statistics with SAS applications*. Crc Press, 2013.

[60] J. Itkonen, M. V. Mantyla, and C. Lassenius, "How do testers do it? an exploratory study on manual testing practices," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 494–497, IEEE, 2009.

[61] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 613–622, IEEE, 2017.

[62] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, "Testing of mobile applications in the wild: A large-scale empirical study on android apps," in *Proceedings of the 28th international conference on program comprehension*, pp. 296–307, 2020.

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

[64] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1070–1073, IEEE, 2019.

[65] A. Apaolaza and M. Vigo, "Wevquery: testing hypotheses about web interaction patterns," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. EICS, pp. 1–17, 2017.

[66] "cypress-io/cypress: Fast, easy and reliable testing for anything that runs in a browser.." `https://github.com/cypress-io/cypress`. (Accessed on 02/07/2023).

[67] "npm." `https://www.npmjs.com/`. (Accessed on 02/07/2023).

[68] "github-tooling/ghtopdep: Cli tool for sorting dependents repo by stars." `https://github.com/github-tooling/ghtopdep`. (Accessed on 02/07/2023).

[69] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th international conference on Software engineering*, pp. 356–366, 2014.

[70] G. Kudrjavets, N. Nagappan, and A. Rastogi, "Do small code changes merge faster? a multi-language empirical investigation," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 537–548, 2022.

[71] "Puppeteer — puppeteer." `https://pptr.dev/`. (Accessed on 02/07/2023).

[72] "Webev-dataset - github." `https://github.com/fuadmmnf/WebEV-Dataset.git`. (Accessed on 03/08/2023).

[73] "The mongodb database - github." `https://github.com/mongodb/mongo`. (Accessed on 02/07/2023).

[74] K. Lyu, K. Noda, and T. Kobayashia, "Sdexplorer: a generic toolkit for smoothly exploring massive-scale sequence diagram," in *Proceedings of the 26th Conference on Program Comprehension*, pp. 380–384, 2018.

[75] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 45–54, IEEE, 2017.

[76] "Interaction design pattern library - welie.com." `http://www.welie.com/patterns/`. (Accessed on 02/05/2023).

[77] V. Gkantouna, A. Tsakalidis, and G. Tzimas, "Mining interaction patterns in the design of web applications for improving user experience," in *Proceedings of the 27th ACM conference on hypertext and social media*, pp. 219–224, 2016.

[78] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1398–1409, 2020.

[79] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 153–164, 2020.

[80] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pp. 2–8, 2018.

[81] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pp. 31–37, 2018.

[82] A. V. Singh and A. M. Vikas, "A review of web crawler algorithms," *International Journal of Computer Science & Information Technologies*, vol. 5, no. 5, pp. 6689–6691, 2014.

[83] Google, "Puppeteer — tools for web developers," 2010.

[84] ebat, "ebat replication package," 2022.

[85] splittypie, "Splittypie: easy expense splitting," 2016.

[86] retroboard, "Retrospective board," 2017.

[87] phoenix, "Phoenix: Trello tribute done in elixir, phoenix framework, react and redux," 2016.

[88] pagekit, "Pagekit: modular and lightweight cms," 2015.

[89] dimeshift, "Dimeshift: easiest way to track your expenses," 2016.