Testing Web Applications: A Survey

Mouna Hammoudi

ABSTRACT

Web applications are widely used. The massive use of web applications imposes the need for testing them. Testing web applications is a challenging process given that it needs to account for the dynamic, asynchronous and interactive nature of web applications. Various strategies exist for testing web applications such as capture-replay and programmable web testing. However, test suites created in this manner are brittle and easily break when changes are applied to the web application under test. Furthermore, web applications continuously evolve and new versions of web applications are constantly released in order to fix bugs, respond to changing requirements, modify layouts, etc. The continuous evolution of web applications might lead to test suite obsoleteness. In this scenario, the test suite that was created for the first version of the web application would become outdated and would require repair. In this paper, we present a survey relative to testing web applications. We selected eight papers that discuss topics related to testing web applications. The topics that are discussed in this paper are: Test repair, test breakage prevention, test maintenance, capture-replay testing versus programmable web testing and faults within web applications.

1. INTRODUCTION

Web applications evolve quickly in order to fix bugs, change layouts and respond to changing requirements. Regression testing for web applications is essential in order to ensure that changes being made did not damage the existing functionalities. Manual black box testing of web applications is an expensive and laborious process. Hence, software engineers create test scripts in order to automate the testing process. Creating test scripts is difficult and expensive given that software testers need to consider all the possible input combinations capable of revealing faults [7]. Test scripts can serve regression testing purposes to ensure that no bugs were introduced in a new release of the web application under test. Nevertheless, releasing a new version of a web application might break the initial test scripts due to functionality and user interface changes within the web application. Simple changes such as deleting a web element, displacing it, adding new elements could lead to test obsoleteness [4]. Manually repairing test scripts is an expensive process that requires software engineers to analyze the changes that were applied to the web application and understand why such changes could cause test breakages [7]. Test repair is a complicated process that requires a deep understanding of the logic encoded within the test scripts. Such a process is even more complicated in situations where the software engineer attempting to repair the tests is different from the one who wrote them. Grechanik et al state that professional testers prefer discarding old test scripts and creating new ones rather than fixing them. Creating tests for web applications represents a significant investment [4]. Thus, repairing broken tests would be more cost efficient than rewriting them from scratch. Due to such considerations, researchers created new techniques and tools that aim at facilitating test script maintenance and preventing test breakages.

In this paper, we present a survey based on the analysis of eight papers that address the following topics: (1)Test repair, (2) prevention of test breakages and facilitation of test maintenance, (3) empirical studies related to testing web applications. It is essential to conduct this survey given that addressing issues related to test fragility requires a profound understanding of the state of the art. This survey would inform researchers about the limitations and the weaknesses of the existing techniques used to test web applications. Also, it would enable researchers to create superior approaches that would not suffer from such limitations and that would outperform existing approaches.

We provide brief overviews of the papers that we have taken into consideration while conducting this survey. Each paper is listed under its category.

Test Repair:

Some researchers created techniques for automatically repairing test scripts in the face of evolving web applications.

- Choudhary et al present WATER [3], a tool that suggests test repairs to software engineers. WATER suggests repairs for obsolete locators, obsolete assertions, newly added form elements and deleted form elements.
- Leotta et al suggest the use of a multi-locator approach in order to repair test cases. Whenever a test breaks due to the use of an obsolete locator within the new version of the web application, the multi-locator approach determines the web element that the test case

failed to locate within the new version of the web application under test [9].

- Alshawan et al present a novel testing technique that makes use of user session data for testing web applications. They propose an automated technique for repairing session data whenever changes are made to the web application under test [5].

Prevention of Test Breakages and Facilitation of Test Maintenance:
Some researchers created techniques for preventing test breakages and facilitating test maintenance in the face of evolving web applications.

- Leotta et al created an approach called ROBULA used for preventing test breakages. ROBULA reduces web test case aging through the generation of robust XPath locators that are less likely to break whenever the web application evolves [8].

- Yandrapally et al created a new record/replay approach that identifies web elements based on their contextual clues. Their technique identifies the web element of interest based on other elements surrounding it. This technique automates the process of test case creation and prevents future test breakages that could occur due to the web application's evolution [19].

- Stocco et al developed a technique for creating web objects automatically. This technique reduces the effort required in order to maintain tests whenever the web application evolves. This technique addresses the problem of test code duplication. Whenever changes are made to the web application under test, the tester only needs to repair the cause of the break once. Such a repair would be automatically propagated to all tests using the broken element [16].

Empirical Studies:
Some researchers conducted empirical studies related to web applications and their testing mechanisms.

- Some researchers analyzed the process of test creation. These researchers conjecture that the manner in which the initial test suite was created for the first version of the web application might impact the robustness of the test suite in the face of web application changes. Leotta et al conducted an empirical study to compare the efforts required for test creation and test evolution using capture/replay web approaches versus programmable web approaches. They present data proving that capture/replay constitutes an approach that incurs high costs relative to test maintenance and test creation [7].

- Other researchers attempted to understand the nature of faults within web applications. Marchetto et al produced a web fault taxonomy that categorizes faults encountered in web applications. Such a taxonomy can be used by researchers in order to seed realistic faults within web applications for experimental purposes. These seeded faults would be representative of real web application faults [11].

2. BACKGROUND

In this section, we present background that is required to understand the remainder of this paper.

2.1 Web Application Evolution

Developers usually apply changes to their web applications as a means for meeting new requirements, adding new functionalities, fixing bugs, etc. Leotta et al [8] define two types of changes that can be applied over web applications: (1) Structural changes and (2) Logical Changes. Structural changes refer to all the changes that modify the page layout and structure such as re-styling or changing a web element's locator. For example, a structural change could be that the id="Password" of a text field is changed into id="Pwd". Logical changes refer to all the changes modifying the logic of the web application under test such as functionality addition, functionality modification, functionality deletion, etc [8].

2.2 Tool Fragility

Web applications can be tested using different types of tools. First generation tools rely on screen coordinates in order to locate and manipulate web elements within the web application under test. These tools produce fragile test suites that are dependent on screen resolution and window size used while testing the web application under consideration. Second generation tools counterbalance the drawbacks of first generation ones by offering simple web element selection mechanisms. The web element of interest is located according to its position within the DOM tree of the web page under test [7]. Tests that are created using second generation tools are fragile as well, given that their execution is dependent on the DOM tree of the web page under test [7]. Simple changes affecting the DOM tree of a web application could cause a test to break. Third generation tools test web applications based on image recognition of web elements within the user interface of the web application under test. Tests created using third generation tools are fragile as well, given that simple changes affecting the visual appearance of a web element could cause the test to break. For instance, changing the size and the color of the web element of interest might cause the test to break. Researchers have aimed at addressing test fragility by creating new techniques and approaches [7].

2.3 Web Application Tests

It is important to understand the components and the structure of test cases. Tests are typically populated with a set of commands driving test execution and instructing the browser to perform specific actions during the test execution process [14]. In order to illustrate a command's structure, we consider the example of Selenium IDE, which is a record/replay tool used for recording a sequence of user interactions and replaying them. Each command within a test case consists of a three tuple <action, locator, value>. The action command specifies which action (click, select, etc) needs to be applied during the test execution process. The locator component specifies the web element that needs to be selected and manipulated during test execution. Finally, the value component specifies the value that needs to be input within a certain web element during test execution [14]. For instance, "John" could be specified as a value to be input within the "First Name" text field of the web page under


```
chars = ''.join(random.sample(string.letters, 15))
type(chars)
```

Figure 1: Typing a string within a text field in Sikuli IDE

test. An example of a command is the following three tuple <type, id="firstname", "John">. Such a command would instruct the replay engine to type the value "John" within the text field with id="firstname". Although the example shown above is relative to Selenium IDE, the three components still apply to test scripts created using other tools. Among third generation tools, we list Sikuli which is a tool used for testing web applications based on image recognition. Figure 1 represents an extract from a Sikuli test case instructing the input of a randomly generated string value within the text field "City or Airport".

We notice that each component of the three tuple <action, locator, value> is present within the test extract represented in figure 1. The action "type" is specified in the third line of the test script. The locator is represented in the first line of figure 1 as a screen shot of the text field "City or Airport". The value component of the three tuple is represented by the string "chars", which is passed as an argument to the "type" command in line 3 and randomly generated in line 2 of figure 1.

2.4 Web Element Locators

Locators are used for identifying and applying actions to web elements of interest. There are two major classes of locators: Attribute-based locators and structure-based locators. Attribute-based locators identify an element based on its name/value pair within the DOM tree of the web page under test [19]. Id, Name and LinkText are examples of attribute-based locators. The locator "id=FirstName" is an attribute-based locator which identifies the text field "FirstName" based on its ID. Structure-based locators identify an element based on its position within the DOM tree of the web page under test. The path leading to the web element under test is used as a locator strategy for structure-based locators [19]. CSS selectors, XPaths, DOM locators are examples of structure based locators. For instance, the locator "//div/button[1]" is a structure-based locator and more specifically, an XPath that identifies the first button positioned after a div within the DOM tree of the web page under test.

3. STUDY METHODOLOGY

We adopted a systematic approach in order to select papers to be taken into account within this survey. We chose "Google" as a search engine and we typed the following keywords: "testing", " test fragility", "regression testing", "web applications", "automatic repair", "repair suggestion", "tests breakages", "breakage prevention", "test maintenance", "test evolution", "test robustness". We examined the results output by Google and we only retained conference papers and journal articles among those results. We examined the set of conference papers and journal articles obtained and we selected the most related ones to the object of our survey. Also, we wanted to ensure diversity in the topics covered by our survey so we chose three papers related to test repair, three papers related to the prevention of test breakages and the facilitation of test maintenance. Finally, we chose two papers presenting empirical studies that investigate different research questions in the context of web application testing.

4. TOPICS RELATED TO WEB APPLICATION TESTING

4.1 Test Repair

4.1.1 WATER: Web Application Test Repair

Choudhary et al address the issue of test breakages by creating a technique that automatically suggests test repairs [3]. They create a novel repair strategy that suggests repairs related to locators, assertions, newly added form elements and/or deleted form elements within the new release of the web application under test. The technique presented by Choudhary et al is innovative given that no research has suggested repairs for obsolete values and deleted/added form elements in the new version of the web application under test. Also, no research has attempted to repair locators based on comparing web element properties in the two versions of the web application under test [3]. Choudhary et al propose a novel categorization of the changes that can be applied over a web application:

-**Structural Changes** involve the addition, the deletion or the modification of a DOM node or a DOM attribute [3].

-**Content changes** involve the modification of a DOM node's text or HTML content. Such changes cause failures at the level of the assertions that verify the content of the modified DOM node [3].

-**Blind changes** are defined as changes applied to the server side of a web application. Such changes are not noticeable at the level of the client side of the web application. Therefore, the authors do not attempt to repair test breakages caused by blind changes in web applications [3].

The authors also define four problems that could be caused by the changes that were previously enumerated. Structural changes could cause (1) non-selection, (2) mis-selection and (3) form data problems. The **non-selection problem** refers to the inability of a test case to select a DOM node that was successfully identified in the older version of the web application under test [3]. The **mis-selection problem** refers to the selection of an unintended element in the newer version of the web application [3]. The **form data problem** refers to the use of inappropriate input values within form elements. This problem is caused by the deletion, the addition or the modification of form elements [3]. The addition of form elements imposes the use of specific input values for such elements. Hence, the test case needs to be augmented with appropriate commands applying values to the newly added form elements. In contrast, the deletion of one or more form elements requires the deletion of all test script commands involving the deleted form elements. Content changes could lead to the **obsolete content problem**. This problem is encountered when the value of the assertion does not match the actual value of the DOM node under consideration [3].

Choudhary et al created a technique to automatically suggest test repairs for the four problems previously identified. The approach is composed of two major steps namely (1) collecting test data and (2) suggesting repairs. The data collection phase consists of gathering information relative to the old version and the new version of the web application under test. More specifically, ten properties (id, xpath, class, linkText, name, tagname, coordinates, clickable behavior, visible property, zindex and hash value) are collected for all the web elements in the old version and the new version of the web application [3]. Also, the error/failure message reported by the testing tool and the position (command) at which the test script breaks are collected as well. Such information is input to the second phase of the approach which deals with repair suggestion [3]. The second phase of the approach aims at suggesting repairs for the four problems previously enumerated. The **non-selection problem** and the **mis-selection problem** are addressed by comparing web element properties in the old version and the new version of the web application under test. If any of these properties match, an updated locator is suggested as a repair strategy for the broken one [3]. The **obsolete content problem** is addressed by replacing the expected value of the assertion with the actual value of the web element under test in the new version of the web application [3]. The **form data problem** is addressed through the identification of newly added web elements in the new version of the web application using a FormDiff that compares the old version and the new version of the web application under test. Random values are generated for all of the added web elements in the new version of the web application. If no repair can be found, the technique deletes the command at which the test case breaks and checks if the test case passes. Following all these steps, the approach suggests a list of repairs to the software engineer [3]. Choudhary et al implemented their approach by creating a tool called WATER, they evaluated their approach by considering 11 versions of 3 different web applications. The results of the evaluation indicate that WATER is capable of suggesting correct repairs 81% of the time [3].

Limitations: WATER is a tool that automatically suggests repairs for broken tests. The authors assume that the position at which the breakage is manifested within the test case matches the position at which the cause of the breakage occurs. In practice, the cause of a breakage could be located at a statement preceding its manifestation. Choudhary et al do not suggest any method for root cause analysis.

In practice, some test cases could present "silent breakages", which are situations in which test cases do not explicitly break and keep performing unintended behavior till the end of execution. The authors do not propose any solution in order to address such breakages.

Choudhary et al suggest a repair technique based on random generation of input values for newly added input fields within the web application under test [3]. Random generation of input values may not be effective given that newly added input fields may only accept restricted types of input values. For instance, assuming that a new "date of birth" text field is added within the web page under test, such a text field would only expect dates as input values. The test case would break if the user inputs a random string. Furthermore, the authors do not propose any technique in order to identify newly added input fields and differentiate them from pre-existing ones within the first release of the web application under test.

4.1.2 *Using Multi-Locators to Increase the Robustness of Web Test Cases*

Leotta et al develop an approach for automatically repairing broken XPath locators in a new version of the web application under test [8]. They only consider locator breakages caused by structural changes in the web application under test. They present a new type of locator called multi-locator, which outputs the best locator among a set of locators generated by five different algorithms. The selection procedure relies on a voting mechanism that assigns different weights to each locator creation algorithm [8]. The main motivation behind the use of a multi-locator as opposed to an individual locator is that web element locators are fragile individually, and non-fragile collectively. No research has attempted to address problems related to locator fragility by using a multi-locator approach. Leotta et al address the fragilities of a single locator by creating a novel type of locator called multi-locator, which combines the results produced by different locator generation algorithms and outputs one single locator that corresponds to the most voted one [8]. The multi-locator algorithm is populated with five different XPath generation algorithms, which are used by the following tools (1) FirePath absolute (2) FirePath Relative ID-based (3) Selenium IDE locators (4) Montoto and (5) ROBULA+ [8]. Given that certain locator generation algorithms tend to produce more robust locators than others, the multi-locator approach assigns different voting weights to each locator generation algorithm depending on the latter's robustness. More specifically, FirePath absolute locators are assigned the weight 0.25, FirePath Relative ID-based locators are assigned the weight 0.50, Montoto, Selenium IDE and ROBULA+ are assigned the weight 0.90 [8]. The multi-locator approach only considers candidate web elements that are uniquely identified by all XPath locators in the new version of the web application. Leotta et al use a mathematical formula to compute a voting score for each web element selected by a locator. The element with the highest vote is returned as the correct target web element in the new version of the web application [8]. Once the correct target web element is returned by the multi-locator, the incorrect locators are automatically repaired by re-executing the corresponding generation algorithms on the DOM tree of the new web application version and generating new locators for all the broken element locators. These repaired locators replace the broken ones and can be used as a basis for suggesting new repairs when the web application evolves [8]. The multi-locator approach is evaluated by considering six web applications, each having two versions and comparing the percentages of broken locators using different strategies (the multi-locator approach, ROBULA+, Selenium IDE, Montoto, FirePath Absolute and FirePath Relative ID-based). Experimental results show that the number of broken locators is reduced by 30% on average when using the multi-locator approach as opposed to ROBULA+, Selenium IDE, Montoto, FirePath Absolute and FirePath Relative ID-based [8].

Limitations: The authors only consider situations in which all the locators point to one single web element in the next release of the web application under test. The authors do not address situations in which one locator does not

identify any web element in the next release of the web application. Also, they do not address scenarios in which one single locator identifies two or more elements in the next release of the web application under test.

Leotta et al only focus on the use of XPaths as web element locator strategies. In practice, software engineers use a variety of other locator strategies such as ID, linktext, name, CSS selectors and DOM locators. The authors did not conduct any comparative empirical evaluation justifying their choice to focus on XPaths and exclude other types of locators. Non XPath-based locator strategies might be less susceptible to break when testing new releases of a web application. It might be more cost-effective to repair other locator strategies if they represent lower breakage frequencies compared to XPaths. Locator strategies with lower breakage frequencies would necessitate a lower number of repairs and would thus reduce the repair effort.

4.1.3 Automated Session Data Repair for Web Application Regression Testing

User session data constitutes an effective method for testing web applications. Session data represents realistic test scripts given that they are created by the web application's users [5]. Maintaining and evolving web applications could cause session data to be obsolete given it might no longer represent a valid session in the new version of the web application under test. Alshawan et al propose an automated technique for session data repair in the context of web application regression testing [5]. No researchers have investigated regression testing for web applications using user session data. The contributions of this paper are the following: (1) The creation of an algorithm for regression testing web applications through the use of session data repair (2) The presentation of a controlled experiment's results that evaluate the effectiveness of the technique when applied to 10 versions of an open source web application (3) The presentation of results related to the scalability of the approach and its applicability in situations in which daily regression testing is applied over a web application. The repair algorithm presented in this paper is composed of two major phases: The first phase is a pre-processing white box analysis step that outputs the structure of the new version of the web application under test. The second phase extends the first phase by using its output in order to search for obsolete parameter sets and invalid sequences of URL requests within the web application [5]. The new release of the web application and the original set of user session data are both input to the algorithm. The output of the algorithm is a repaired set of user session data. Two types of repairs are applied over the session data (1) individual URL repair and (2) Sequence repair. We present each one of these:
(1)Individual URL Repair: Given that an individual URL might not use the same parameter, all individual URL requests need to be updated in order to specify new values for newly introduced parameters [5].
(2)Sequence Repair: A session may not correspond to a valid sequence of URL requests in the new version of the web application. Therefore, the sequences of URLs from the existing session database need to be updated in order exclude any nodes and edges that no longer exist in the new version of the web application under test. The authors conduct two controlled experiments in order to evaluate the effectiveness of the approach and its scalability. The results of the first controlled experiment prove that the approach is effective. Also, the results of the second controlled experiment prove that the approach is scalable to more demanding scenarios, in which daily regression testing is applied to the web application under test [5].
Limitations: The authors only consider ten versions of one single web application in order to evaluate their approach. The results of the empirical evaluation may be completely different when considering other types of web applications. It is necessary to apply the approach to a variety of web applications in order to make generalizable conclusions regarding the performance of the algorithm.

4.2 Prevention of Test Breakages and Facilitation of Test Maintenance

4.2.1 Reducing Web Test Cases Aging by Means of Robust XPath Locators

Leotta et al state that the effort required to manually repair web element locators constitutes one of the most important costs required for test repair [8]. They address the issue of test aging, which refers to the obsolescence of test scripts due to the evolution of the web application under test. They address the issue of locator obsoleteness caused by structural changes. Structural changes refer to changes affecting the structure/layout of the web application under test and leaving the application logic unaffected [8]. No research has attempted to prevent locator breakages by creating more resilient locators for the web application under test. Leotta et al create a novel algorithm called ROBULA that generates robust XPath locators, which are more resilient to changes made to a web application. Locators generated by ROBULA are less likely to break when changes are made to the web application under test [8].

ROBULA takes two inputs, namely (1) the absolute XPath of the element to be selected and (2) the HTML page under consideration. ROBULA initially selects all of the web elements in the web page under consideration and iteratively refines the element until we reach the point in which the element of interest is the only one being returned by ROBULA [8]. ROBULA generates initially an XPath for all the elements within the web page. Then, it refines the XPath of the web element under consideration by successively augmenting it and applying the following transformations: (1) Each XPath is augmented with the tag name of the parent node (2) An attribute-value pair is added to refine the parent node that was added in the first step (3) An index position is added to refine the parent node that was added in the first step [8]. This process is repeated until we reach the point in which the XPath locator is able to uniquely locate the web element of interest. ROBULA outputs a robust relative XPath expression that is capable of uniquely selecting the web element under consideration. If the web element of interest does not have a relative XPath expression that is capable of uniquely identifying it, ROBULA outputs the absolute XPath locator of the target web element [8]. ROBULA is evaluated by considering six open source web applications with two versions for each. Leotta et al evaluate the robustness of ROBULA by counting the numbers of broken absolute XPath locators, broken relative XPath locators and broken ROBULA locators in the second version of the web application under test. Experimental results demonstrate that a 56% reduction in fragility is achieved for

absolute XPath locators and a 41% reduction in fragility is achieved for relative XPath locators [8].

Limitations: Leotta et al assume that shorter locators are less susceptible to breakages when the web application evolves. They did not conduct any empirical evaluation comparing short locators with long ones to verify the validity of their assumptions. In some situations, the extra information provided by a longer locator might be necessary to refine the web element selection mechanism and effectively identify the web element of interest.

While conducting the empirical evaluation of ROBULA, the authors only consider situations in which the same web element is present in two successive releases of the web application under test. Leotta et al manually infer the presence of the same web element across two successive versions of the web application under test. They do not present any technique for automatically performing such an inference. Furthermore, inferring the persistence of a web element across two successive releases of a web application is extremely challenging, given that the developer could change all of the attribute-based and hierarchy-based characteristics of the web element of interest. Changing all these characteristics complicates the identification of persisting web elements across successive releases of the web application under test.

4.2.2 Robust test automation using contextual clues

Test scripts of web applications are brittle, small changes in the page layout are capable of breaking tests. The main reason why test cases break is related to the inclusion of metadata within test cases. Such meta-data reflects the internal representation of the web application under test and causes test breakages [19]. Yandrapally et al create a technique for locating web elements in a web page by completely discarding metadata [19]. They create a technique that automatically infers contextual clues in order to uniquely locate user interface elements within the web page. Contextual clues are defined as labels or images that are located in the surroundings of the web element of interest. The use of such contextual clues uniquely selects the element of interest within the web page. The main contribution of this paper consists of the creation of an innovative technique that identifies elements based on their contextual clues. No research has considered the identification of an element based on its vicinity. The authors implemented their approach in a record/replay tool called ATA-QV [19]. The technique consists of two phases: the automation phase and the playback phase [19]. The automation phase creates a test script that records contextual clues for all user interface elements.The playback phase replays the test suite by analyzing the recorded contextual clues and executing the required action(s) on the user interface element of interest. Yandrapally et al evaluate their technique by performing three empirical studies on five open source web applications. The first empirical study aims at verifying the accuracy of the technique and verifying whether ATA-QV generates the same contextual clues that would be produced by a human being [19]. Experimental results demonstrate that 73% of the contextual clues produced by ATA-QV match the ones that could be produced by a human being. The second empirical study aims at evaluating ATA-QV's resilience to changes in web applications compared to other testing tools (Sikuli, QTP, ATA). The authors considered a second version for each of the five web applications under test, ATA-QV was able to correctly identify all of the web elements of interest within the new version of the web application. Also, experimental data shows that ATA-QV outperforms Sikuli, QTP and ATA. The third empirical study aims at measuring ATA-QV's resilience to web browser changes compared to Sikuli, ATA and QTP [19]. Again, experimental results showed that ATA-QV is significantly more resilient to web browser changes and outperforms Sikuli, QTP and ATA.

Limitations: The test scripts produced by ATA-QV could be fragile in situations in which labels are changed. Such changes could involve modifying a label's name, replacing it with an image or completely deleting it. Also, the playback mechanism could fail in situations in which the DOM of the web page under test is modified.

4.2.3 Why Creating Web Page Objects Manually if It Can Be Done Automatically?

Web application evolution poses the problem of maintaining test cases. Stocco et al facilitate test maintenance by suggesting the page object design pattern [16]. The page object design pattern aims at facilitating a test suite's maintainability by limiting the duplication of code across test cases. This approach relies on the use of objects to represent the web page elements as a series of objects. Also, it groups the functionalities of the web application into methods, which facilitates test script reusability, maintenance and readability [16]. Not using design patterns complicates test script maintainability and evolution due to duplicated code across test cases. For instance, if developers change a certain functionality in the new version of the web application under test, all the test cases that exercise such a functionality would need to be updated, which is an expensive process [16]. The use of page objects facilitates test case maintainability in a way that the tester only needs to update one single code fragment if one functionality is modified. Such an update would be automatically propagated to all the test cases that make use of the changed functionality. The use of page objects separates test case specifications from their implementation. Page objects contain all of the implementation details while the content of test cases is only limited to testing logic [16]. There are many page object creation tools such as OHMAP, SWD page recorder, WTF PageObject Utility Chrome Extension [16]. Nevertheless these tools suffer from many limitations: First, they only consider one page at a time and they ignore the structure and the dynamism of web applications. Second, they only consider a subset of web elements within the web page under test. Finally, the page objects being generated by these tools only constitute skeletons; this contradicts the motivation behind page objects, which is to encapsulate web application functionalities into methods. Stocco et al address these limitations by creating a novel technique for automatically generating page objects for web applications. They create a tool called Apogen (Automatic Page Object Generator), which reverse engineers a testing model through static analysis and dynamic analysis of the web application under test [16]. The input of APOGEN is the web application under test and its output is a set of Java files making use of page objects and separating testing implementation details from testing logic [16].

The approach developed is composed of three different phases: (1) a crawler, (2) a static analyzer and (3) a code generator.

The **Crawler phase** makes use of the open source tool Crawljax in order to create a state-based graph representing the dynamic DOM states within the web application under test along with the transitions among DOM states. For each web page within the web application under test, the crawler outputs its URL, the set of clickable items within the web page, links to other states within the state-based graph, the DOM tree for the web page and a screen shot of the web page [16].

The **Static Analysis** phase analyzes the output of the crawler phase and generates all of the necessary information in order to create page objects. It uses some parsing mechanisms in order to generate meaningful class names for page object classes and meaningful method names. Furthermore, transitions to other states within the state-based graph are saved within the classes. The output of this phase is a model representing each web page within the web application [16].

The **Code Generation phase** consists of generating appropriate code for each state within the model output by the previous static analysis phase. More specifically, this phase creates Java classes and populates them with web elements, constructors and methods using the information gathered in the static analysis phase. Stocco et al evaluate APOGEN by comparing a manual test suite created by an external tester with the methods automatically generated by APOGEN. Experimental results show that 75% of the methods automatically generated by APOGEN are equivalent to the functionalities covered by the manual test suite and 25% of the methods automatically generated by APOGEN only require minor modifications [16].

Limitations: The crawling phase of APOGEN is conditioned by the effectiveness of Crawljax in generating a state-based graph. Page object creation would fail in situations in which Crawljax is unable to generate a state space for one of the web pages under test.

4.3 Empirical Studies

4.3.1 Capture-replay vs. programmable web testing: An empirical assessment during test case evolution

Two major types of trends are used for testing web applications: (1) Capture/replay of web application interactions and (2) Programmable web testing. No attempts were made to compare the costs relative to test creation and test maintenance using programmable web testing versus programmable web testing. Leotta et al conducted an innovative empirical study that compares record/replay testing and programmable web testing [7].

Capture/Replay web testing consists of recording a sequence of user interactions at the level of the web application under test and saving that recording into a test script that can be replayed any number of times. Test scripts created using capture/replay tools do not require any advanced testing skills and the creation of such test scripts incurs relatively low costs. However, test scripts created using these tools are fragile and tend to break due to the evolution of the web application under test [7]. Such breakages are caused by a strong coupling between the test cases and the web pages. Test scripts could break due to minor changes applied over the web application under test. Furthermore, test scripts created using record/replay tools make use of hard-coded values that need to be updated if any changes are applied to the web application under test. These test scripts would need to be repaired manually or re-recorded from scratch in order to repair the break [7].

Programmable web testing requires advanced testing skills and the programmatic creation of test scripts incurs high costs. Programmable web testing consists of writing tests as opposed to recording them. Such test scripts can make use of more sophisticated programming constructs comparing to tests created using a capture/replay approach. Also, tests created using programmable web testing can take advantage of all the benefits of programming paradigms such as modular programming, conditional execution, test logic reuse, page objects, etc. These tests scripts can be more easily modified than capture/replay ones [7]. Leotta et al compare test scripts created using a record/replay approach and test scripts created using programmable web testing. They conduct a study that aims at: (1) Comparing the test suite development effort while using programmable web testing approaches versus record/replay ones [7]. (2) Comparing the effort required for maintaining programmable test suites versus capture/replay ones when a new version of the web application under test is released [7]. (3) Determining the number of releases after which the use of programmable test suites becomes more convenient and more cost-effective in comparison with record/replay test suites [7].

Answers to these research questions would inform project managers about more suitable testing approaches given time constraints, number of versions to be released, developers' expertise, etc [7].

In order to respond to these questions, Leotta et al considered Selenium IDE as a record/replay approach and Selenium WebDriver as a programmable web testing approach. They considered six web applications, with each web application having two versions. They created two equivalent test scripts using Selenium IDE and Selenium WebDriver for each web application under test. Also, they used the page object design pattern for all the test cases that were created using Selenium WebDriver [7].

Experimental results demonstrate that for all the web applications under test, the test suite creation effort using Selenium IDE is significantly lower than the test suite development effort using Selenium WebDriver. The development effort for Selenium IDE test suites required amounts of time ranging from 68 to 291 minutes. The development effort for Selenium WebDriver test suites required amounts of time ranging from 98 to 383 minutes [7].

The repair effort required for maintaining Selenium IDE test suites is significantly greater than the repair effort required for maintaining Selenium WebDriver test suites. The repair effort for Selenium IDE test suites required amounts of time ranging from 46 to 95 minutes. The repair effort for Selenium WebDriver test suites required amounts of time ranging from 71 minutes to 120 minutes [7].

Experimental results show that the cumulative costs relative to the development and the evolution of programmable test suites become lower than the same costs relative to record/replay test suites after a small number of releases (2 releases) [7].

Limitations: The authors did not discuss the types of locators used in their test suites. The repair effort may vary according to the nature of the web element locators used in the test suite.

4.3.2 *Empirical Validation of a Web Fault Taxonomy and its usage for Fault Seeding*

Web applications are widely used, existing research aims at developing new testing techniques for web applications. No attempts were made to characterize the types of faults encountered within web applications. Marchetto et al produce a web fault taxonomy that enumerates faults that could only be encountered in web applications as opposed to generic faults that could be encountered in any program [11]. Marchetto et al construct an initial taxonomy in a top down manner and refine it through four iterations of empirical validation. Numerous groups of real bugs were extracted from real bug tracking systems in each iteration [11]. These groups enabled the refinement of the taxonomy by splitting classes, merging them, deleting them, etc. The refinement process of the web taxonomy aimed at creating a "good taxonomy", which is defined as being general, complete and exhaustive and not ambiguous. Such a fault taxonomy can be used by web testers in order to create test cases that target certain types of faults [11]. Furthermore, this taxonomy can also be used by researchers for fault injection within web applications. The reliance on the taxonomy for fault seeding guarantees the injection of realistic faults encountered within real web applications [11]. Marchetto et al identified six major classes of web faults, namely (1)multi-tier architecture faults (faults related to the interactions between the client, the server and the database), (2)GUI faults (faults relative to HTML, JavaScript and Flash), (3)Session management faults, (4)Hyperlinked structure faults, (5)Protocol based faults (HTTP, HTTPS, etc) and (6)Authentication faults [11].

Limitations: The authors created a preliminary version of the taxonomy based on their assumptions. Then, they confirmed it considering online bug reports for real web applications. The process followed for creating the taxonomy may have been more objective and accurate if the authors built their preliminary version starting from bug reports for real web applications. Furthermore, the authors only considered 376 bugs in total, which may not be sufficient to generalize the applicability of the web fault taxonomy to all web applications.

5. DISCUSSION

Root Cause Analysis. The time at which a breakage is manifested and the position at which such a breakage is caused within the test case may be different.

Direct breakages are manifested at the same position where the breakage cause occurs.

Propagated breakages are not manifested immediately, but are manifested later on subsequent test actions.

Silent breakages never manifest themselves. The test case continues executing without explicitly breaking. In this situation, the test would no longer be testing what it was designed to test. These problems suggest that we need better techniques to map the manifestation of a break to its root cause. Tracing back the manifestation of a break to its root cause would facilitate automated test repair given that root cause analysis is a pre-requisite for test repair.

IDE enhancements. IDEs could be enhanced in order to automatically update test cases and reflect changes made by a web developer over a web application. Also, such IDEs could allow breakage avoidance by alerting programmers regarding test breakage risks. The IDE would approve or disapprove developer changes made over a web application depending on the risks incurred by these changes. These IDEs would approve "safe" changes that would not lead to test breakages and would disapprove "harmful" changes that might cause test breakages. In the latter case, the IDE could propose alternative changes that are "safe" and that would not cause test breakages. Furthermore, IDEs could allow breakage prevention by forbidding developers from applying "harmful" types of changes and only allowing them to apply "safe" changes.

Bad Smells in Tests. Numerous approaches have investigated bad smells in code [18], [20], [10]. However, no approaches have been created to investigate the occurrence of bad smells in tests made for web applications. Future research could identify breakage threats in web application tests. A technique could signal "bad smells" in test cases by comparing two successive releases of the web application and analyzing its test cases. In this case, "bad smells" would represent statements that are likely to break while executing the test case for the new version of the web application. By signaling these "bad smells" within tests, the developer could rectify some of the code changes that he/she applied over the web application, as a means for avoiding risks related to test breakages.

Detecting Changes in Web Application Versions. Test breakages are caused by changes applied over a web application. Identifying changes made to a web application could be a useful way for repairing tests. For instance, identifying that a new input field was added within a web page would signify that the test case should be augmented with a statement populating the newly added input field. Conversely, detecting an input field deletion would signify that the statement populating the deleted input field should be removed from the test case. Identifying persisting input fields across multiple versions of a web application would let us examine their locator strategies and repair them if necessary. Also, the identification of any added web pages would let us create new test cases for exercising functionalities within the newly added web pages. Similarly, identifying deleted web pages would let us remove obsolete test cases from our test suite.

Automatic generation of input values. Numerous changes that are made to web applications consist of the addition of new input fields. Choudhary et al propose a technique based on random generation of input values in order to populate any newly added input fields in the next release of the web application under test [3]. This technique might not be effective in situations where newly added input fields only accept restricted types of values. New techniques could be devised in order to automatically generate values for the newly added input fields. It would be more strategic to identify the restrictions corresponding to the newly added input fields and then generate values that satisfy those restrictions.

Locator Repair. Leotta et al suggest a multi-locator repair technique but only consider situations in which locators identify one single element in the next release of the web application under test [9]. In practice, there could be situations in which a locator is not capable of identifying any web element in the next release of the web application under test. Also, there could be situations in which locators identify two or more elements in the next release of the web application. Some repair techniques could be created to ad-

dress these scenarios. In particular, some techniques could specialize in repairing locators that are not capable of identifying any element in the next release of the web application under test. Other techniques could specialize in repairing scenarios where locators identify two or more elements in the next release of the web application under test. Leotta et al present ROBULA, which is a prevention approach generating robust XPath locators [8]. Also, they suggest a repair approach based on the use of multi-locators [9]. However, both of these approaches focus on the use of XPath locators. Future work could focus on the creation of similar prevention and repair approaches for other classes of locators such as ID, CSS selectors, DOM locators, name, linktext, etc.

Empirical Studies on Locators. Locators cause a considerable source of breakages within web application test cases. There are multiple types of locators: ID, name, CSS selectors, XPath, DOM locators, etc. Some locator types may be more fragile than others. No empirical studies have investigated locator fragility based on the types of changes made to the web application under test. More specifically, we conjecture that a high number of changes applied to the structure of the DOM would tend to break structure-based locators. Similarly, a high number of changes applied to web element attributes would tend to break attribute-based locators. Future research could investigate the correlation between locator fragility and changes made to the web application under test. This could let developers know which types of locators are more susceptible to break than others considering the nature of changes made to the web application under test. Also, this would inform developers about locator robustness depending on the nature of changes applied to the web application under test.

Leotta et al suggest that short locators are more robust than long ones [8]. Such an assumption motivates the creation of ROBULA as a technique that creates the shortest possible web element locators capable of correctly identifying the web element of interest. The authors only assume that short element locators are less fragile than long ones. They do not conduct any empirical study to validate their assumptions. We could conjecture that in some scenarios, the extra information provided by a longer locator might be necessary to refine the web element selection mechanism and be able to identify the correct web element. A short locator might omit important information that is necessary to correctly identify the web element of interest. Future research could investigate the correlation between locator length and locator fragility. Such an investigation would let researchers know about situations in which long locators are more robust than short ones. Similarly, this empirical study would inform developers about the situations in which short locators are more robust than long ones.

6. RELATED WORK

Considerable research has been devoted to the development of record/replay tools for testing web applications. These tools allow software engineers to record user interactions and replay them. Such recordings can be used for regression testing new versions of a web application. Also, these recordings can serve as bug reports demonstrating the occurrence of the failure during the replay process. Warr [1] is a tool that was developed for recording and replaying any sequence of user interactions with high fidelity within a web application. Timelapse [2] is a capture/replay tool that deterministically records and replays a sequence of user interactions within the web application under test. Timelapse records not only user inputs but also network callbacks. Timelapse replays user interactions by suppressing "live" variables and only re-delivering recorded inputs. It memoizes persistent JavaScript states (browser cookies, local storage, etc.) and environmental data(current time, screen size, etc.). Mugshot [12] is a capture/replay tool that allows developers to deterministically record and replay user interactions in a web application. This tool facilitates root-cause analysis given a specific failure and offers insight into the web application's execution.

Other research has focused on regression testing for web applications. Hirzel [6] proposes a selective regression testing approach for testing web applications created using the Google Web Toolkit. The technique specifies which tests need to be rerun by comparing Java code for two versions of the web application and tracing it back to JavaScript code. Raina et al [15] developed an automated tool for performing regression testing at the level of web applications. The tool achieves effective regression testing by comparing the changes that were applied to different versions of a web application. Zhou et al [21] developed a pattern-based automated testing framework, called UTF (User-oriented Testing Framework) for regression testing web applications. UTF offers an easy creation and maintenance process for web application test scripts by identifying invariant structural patterns in the DOM trees of the web pages under test.

Other researchers have developed techniques for automated test generation at the level of web applications. Torsel [17] developed a model-based testing technique for performing black box testing in web applications. The approach produces fully automated test scripts for a given web application. Milani Fard et al [13] propose an approach based on mining human knowledge from existing test suites created by humans. Milani Fard et al couple the inferred knowledge from these test suites with the capabilities of an automated crawler in order to generate new test cases for uncovered/unchecked portions of the web application.

7. CONCLUSIONS

We presented a survey discussing different topics related to web application testing. We considered eight different conference papers and we were able to cluster them into three distinct categories, namely (1) test repair, (2) prevention of test breakages and facilitation of test maintenance and (3) empirical studies. This survey provides an overview of the state of the art with respect to testing web applications. Hence, this survey could guide researchers and inform them about the problems that have not been addressed yet by the state of the art. This survey could orient researchers towards problems that are worth considering as future research directions in the context of testing web applications.

8. REFERENCES

[1] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410, June 2011.

[2] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application

debugging. In *UIST 2013: Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, pages 473–484, St. Andrews, UK, October 8–11, 2013.

[3] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 24–29, New York, NY, USA, 2011. ACM.

[4] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[5] M. Harman and N. Alshahwan. Automated session data repair for web application regression testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 298–307, April 2008.

[6] M. Hirzel. Selective regression testing for web applications created with google web toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 110–121, New York, NY, USA, 2014. ACM.

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

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

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

[10] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 31–34, New York, NY, USA, 2007. ACM.

[11] A. Marchetto, F. Ricca, and P. Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 31–38, Oct 2007.

[12] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[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*, ASE '14, pages 67–78, New York, NY, USA, 2014. ACM.

[14] H. Pirzadeh and S. Shanian. Resilient user interface level tests. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 683–688, New York, NY, USA, 2014. ACM.

[15] S. Raina and A. P. Agarwal. An automated tool for regression testing in web applications. *SIGSOFT Softw. Eng. Notes*, 38(4):1–4, July 2013.

[16] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Why creating web page objects manually if it can be done automatically? In *Proceedings of the 10th International Workshop on Automation of Software Test*, AST '15, pages 70–74, Piscataway, NJ, USA, 2015. IEEE Press.

[17] A.-M. Torsel. Automated test case generation for web applications from a domain specific model. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 137–142, July 2011.

[18] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA, 2015. IEEE Press.

[19] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 304–314, New York, NY, USA, 2014. ACM.

[20] M. Zhang, T. Hall, N. Baddoo, and P. Wernick. Do bad smells indicate "trouble" in code? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 43–44, New York, NY, USA, 2008. ACM.

[21] J. Zhou and K. Yin. Automated web testing based on textual-visual ui patterns: The utf approach. *SIGSOFT Softw. Eng. Notes*, 39(5):1–6, Sept. 2014.

Table 1: Summary of the Empirical Studies

Empirical Study	**4.3.1- Capture-replay vs. programmable web testing: An empirical assessment during test case evolution**	**4.3.2- Empirical validation of a fault taxonomy and its usage for fault seeding**
Purposes	-Compare test suite creation time, repair time for record/replay and programmable test suites -Determine the number of releases after which programmable test suites become more cost effective than record/replay test suites	-Create a web fault taxonomy that is specific to faults encountered within web apps
Approach	-Install six web apps(with two releases for each) -Create Selenium WebDriver test suites -Create equivalent Selenium IDE test suites -Measure the amount of time required for test suite creation at the level of Selenium IDE test suites and Selenium WebDriver test suites -Measure the amount of time required for test suite repair at the level of Selenium IDE test suites and Selenium WebDriver test suites -Count the number of releases after which the cumulative time for the creation and the repair for Selenium WebDriver test suites becomes lower than the same cost for Selenium IDE test suites	-Create an initial top down taxonomy based on authors' assumptions -Apply four iterations in order to refine the taxonomy by splitting classes, merging classes, deleting classes, etc

Table 2: Summary of the Approaches

Approach	**4.1.1- Water**	**4.1.2- Multi-Locator**	**4.1.3- Session Data**	**4.2.1- ROBULA**	**4.2.2- Contextual Clues**	**4.2.3- Page Objects: APOGEN**
Approach Type	Suggests repairs for broken locators, assertions and values	Repairs broken XPath locators by using a voting procedure	Repairs user session data by performing individual URL repairs and sequence repairs	Prevents XPath locator breakages	Prevents locator breakages by identifying elements based on their contextual clues	Facilitates test script maintenance by automatically generating page objects for the web app under test
Web App Changes under consideration	Structural and logical changes	Structural changes	Structural and Logical changes	Structural changes	Structural and Logical changes	Structural and Logical changes
Solution	-Repairs for all types of locators -Repairs assertions -Repairs values	-Repaired XPath locator (The most voted XPath locator)	-Updated user session data that is applicable to the new version of the web app under test	-More robust XPath locators, which are less likely to break in a new release of the web app	-Web element localization strategies relying on the use of contextual clues	-Page objects encapsulating the implementation details within the test case -Test scripts that are limited to testing logic

APPENDIX

Table 1 summarizes the purposes and the approaches that were followed for conducting the two empirical studies presented in sections 4.3.1 and 4.3.2.

Table 2 summarizes the approaches presented in this paper. For all of the approaches presented in sections 4.1 and 4.2, table 2 presents the approach type, the web application changes taken into consideration and the solution offered by the approach.