WATER: Web Application TEst Repair

Shauvik Roy Choudhary[*], Dan Zhao[†*], Husayn Versee[*], Alessandro Orso[*]

[*]College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
shauvik@cc.gatech.edu
hversee3@gatech.edu
orso@cc.gatech.edu

[†]College of Information Science and Engineering
Hunan University
Changsha, Hunan, China
dzhao34@gmail.com

ABSTRACT

Web applications tend to evolve quickly, resulting in errors and failures in test automation scripts that exercise them. Repairing such scripts to work on the updated application is essential for maintaining the quality of the test suite. Updating such scripts manually is a time consuming task, which is often difficult and is prone to errors if not performed carefully. In this paper, we propose a technique to automatically suggest repairs for such web application test scripts. Our technique is based on differential testing and compares the behavior of the test case on two successive versions of the web application: first version in which the test script runs successfully and the second version in which the script results in an error or failure. By analyzing the difference between these two executions, our technique suggests repairs that can be applied to repair the scripts. To evaluate our technique, we implemented it in a tool called WATER and exercised it on real web applications with test cases. Our experiments show that WATER can suggest meaningful repairs for practical test cases, many of which correspond to those made later by developers themselves.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Web testing, test repair

1. INTRODUCTION

Web applications are popular among developers due to their instant deployment. i.e., unlike traditional desktop applications, web applications can be updated at a single point (server or cluster) and quickly delivered to all users. Due to this feature of web applications, they tend to evolve more frequently as compared to traditional desktop software. Thus, regression testing is essential to ensure that the updates do not break existing functionality of the web application. The changes made during evolution often results in errors and failures in test automation scripts that exercise such applications. One could discard such test cases from the test suite but this reduces the quality of the regression test suite. Thus, such broken scripts need to be updated to work with the latest version of the web application. The task of updating such scripts is currently performed manually, which requires a significant manual effort and time.

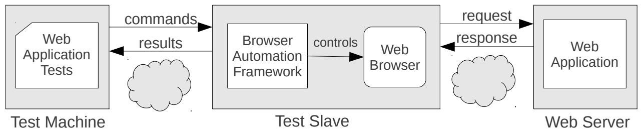
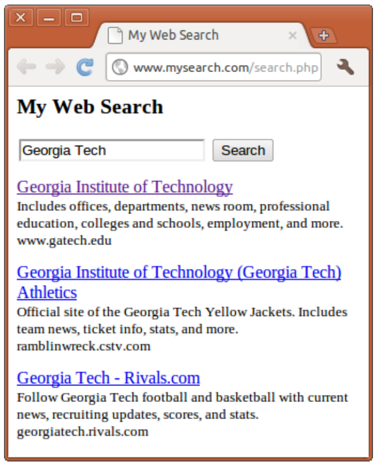


Figure 1: A typical web testing deployment.

Since web applications follow a client-server model, their testing is more sophisticated than traditional applications that run on a single machine. For illustration, we present a typical web testing deployment in Figure 1. The tests are executed on the test machine, which contacts one of many test slaves (virtual machines) to execute the tests. Web application tests consist of a set of actions that are performed on web page loaded inside the web browser. For each test case, the test input is the data entered on a form or the sequence of events (mouse clicks, selections etc.) fired on web page elements. These tests are written either in a domain specific language or in a standard programming language using an client library API, both provided by a Browser Automation Framework (BAF). The test actions are actually run on the test slave machine, which consists of a web browser and the BAF setup in a particular configuration. The BAF accepts commands for performing various actions, which it executes in the browser. These commands initially load in the browser, the web application under test, which is hosted on a remote web server. After loading the application, subsequent actions are performed on it, which results in execution of client side scripts (e.g., JavaScript) or requests posted to the web server for executing additional server side scripts. Like any test case, web application tests have assertion checks after important actions have been performed on the web application or at the end of the test case. In particular, the assertions in web tests check for the presence of certain elements or select an element and checks its contents against expected values. These checks determine the success or failure of the test case.

In web applications, the elements on the web page are often changed due to requirements change or get displaced due to insertion or deletion of other elements. This can lead to an error if the test is unable to locate the element at its original location. Such errors are common and currently need manual effort from the developer to understand the change and update the test case accordingly.


My Web Search

www.mysearch.com/search.php

My Web Search

Georgia Tech | Search

Georgia Institute of Technology
Includes offices, departments, news room, professional education, colleges and schools, employment, and more.
www.gatech.edu

Georgia Institute of Technology (Georgia Tech) Athletics
Official site of the Georgia Tech Yellow Jackets. Includes team news, ticket info, stats, and more.
ramblinwreck.cstv.com

Georgia Tech - Rivals.com
Follow Georgia Tech football and basketball with current news, recruiting updates, scores, and stats.
georgiatech.rivals.com

Figure 2: Screenshot of *My Web Search*.

Typically, the developer loads the old and new version of the web application in the browser and analyzes the elements in a debugger to develop the fix. We think this process can be automated and such errors can be fixed automatically.

To address the limitations of existing technology, in this paper we propose a technique to automatically suggest repairs for outdated test cases of the following two types: (1) tests that result in a failure due to mismatch between expected and actual values of web page elements. (2) tests that result in errors due to displaced or changed web page elements.

The rest of this paper is organized as follows. In the next section, we present an example web application along with a test and sample changes to motivate our technique. This is followed by the description of our technique and implemented details in Section 3. Then we present results of our empirical evaluation in Section 4 followed by related work and concluding remarks in Sections 5 and 6.

2. MOTIVATING EXAMPLE

Before we describe our approach, we introduce a simple web application along with tests to verify its functionality and use it as a motivating example.

The Web-Application. The web application *"My Web Search" (MWS)* is a simplified web search engine that accepts a search keyword and returns a list of websites matching the keyword. Figure 2 shows the screenshot of a web page generated by *MWS*. On the initial page, the user enters a search keyword in the text box and clicks on the search button. The resulting web page displays for each search result, the title of the website, a paragraph of description from the website and the website URL. Figure 3 shows the generated HTML source associated with the web page, which was sent to the browser in response to the request. This page is parsed into a DOM[1] tree by the browser and is exposed to client side scripts through the DOM API provided by the web browser. On lines 4–7, the HTML source shows the web form containing the text box with name `q` and the search button that has the identifier `search`. Lines 8–15 shows the first search result and contains result title enclosed in the `a` tag, the result description in the `p` tag and the result url in the `span` tag.

The Test Case. A sample test case for the MWS application is presented in Table 1. The test has been written using the Selenium web testing system [11] and consists of different commands that are executed in the web browser. This is how the script follows: The first command, `open`, takes the path of the server side script

[1]Document Object Model (http://www.w3.org/DOM/)

```
 1 <html>
 2 <body>
 3 <h2>My Web Search</h2>
 4 <form>
 5   <input type="text" name="q"/>
 6   <input id="search" type="submit" value="Search">
 7 </form>
 8 <div>
 9   <a href="http://www.gatech.edu/">
10     Georgia Institute of Technology
11   </a>
12   <p>Includes offices, departments, news
13      room, professional education...</p>
14   <span>www.gatech.edu</span>
15 </div>
16 <div>...</div>
17 <div>...</div>
18 </body>
19 </html>
```

Figure 3: HTML source of *My Web Search*.

as an argument page and loads it in the web browser. The next command, `type`, enters the string "Georgia Tech" into the textbox identified by name `q`. The `clickAndWait` command then performs a click on the search button identified by its id `search` and waits for the resulting page to load. After the page has loaded, the last two `assertText` commands, select a DOM node and check its text content against expected values. In particular, the command on line 4 uses the DOM API to select a hyperlink node based on its indexing on the webpage and checks if its text content match "Georgia Institute of Technology". The next command selects a node using its XPath location and checks if its text content matches "www.gatech.edu".

Changes to the Web Application. We consider four changes to the MWS web application as shown in Figure 4. In Change A, the identifier of the search button on line 6 is modified to "searchBtn". This would lead to an *error* in the test case on line 2 because of the failure to locate the button using its original identifier "search". To repair this command, the developer modifies the argument to `id="searchBtn"`. For Change B, a link was inserted in the form between lines 6 and 7. This change results in a *failure* at line 4 of the test case because the indexing of the element's DOM node changed. To fix the failing test case command for this change, the developer would modify the DOM locator in argument 1 to `document.links[1]`. Incase of Change C, the content of the DOM node associated with the `a` tag on lines 9-11 has changed. This will lead to a *failure* at the command on line 4 of the test case due to the mismatch between actual and expected values. This failure can be fixed by updating the argument 2 of this command with the actual value of the DOM node text content (i.e., "Georgia Institute of Technology (GATech)" in this case). For Change D, a `div` is placed around the results on lines 18–17. This change leads to an *error* at the `assertText` command on line 5 of the test case because the XPath locator can no longer be resolved to any element. In order to fix this command, the developer would change the XPath in argument 1 to `//body/div/div[1]/span`.

Note that for the purpose of illustration in this example, we show a simple server-side application i.e., all the functionality is performed by the the server side script that generates the web page. However, the same can be performed using AJAX (Asynchronous JavaScript And HTML), where the request is sent and received by a JavaScript program, and the results are updated on the screen using the DOM API. Although in the motivating example, we illustrate the differences at the HTML level, our technique described in this paper operates on the dynamic DOM tree that is maintained by the browser as a result of all such client-side scripts.

Reasons for Broken Test Scripts. We identified that there are mainly the following three reasons of broken scripts on a given web page: (1) Structural changes (2) Content changes. (3) Blind

	Command	Argument 1	Argument 2
1.	open	/search.php	
2.	type	q	Georgia Tech
3.	clickAndWait	id=search	
4.	assertText	document.links[0]	Georgia Institute of Technology
5.	assertText	//body/div[1]/span	www.gatech.edu

Table 1: Selenium Web Test.

changes. We define **structural changes** in the DOM tree as the changes where any DOM node or node attribute is added, modified or deleted. Due to such changes, a locator that could select a DOM node in the older version of the web application might no longer be able select the corresponding DOM node in the newer version, thereby leading to a test case error. We call this problem a *non-selection problem* and it can be due to a deleted DOM node or changed DOM location in the DOM tree. Structural changes can also result in a different node being selected, leading to a problem that we call *mis-selection problem*. In this case, the test case command may either fail due to the observed difference in node contents of the two corresponding nodes or might result in an error due to actions not supported by the wrongly selected node (e.g., the test case command might try to type text on an element that does not support the type command). A specific type of structural changes are the addition, deletion or modification of form elements. Form elements are important because they are used to collect data from users, which is then read to perform some action. The values entered by the test script in such form elements are sometimes essential for the test script to proceed, especially if they are required by input validation checks. In particular, if a new element is added to a web form, the test case should be augmented with a step to add suitable data to the element. Whereas if an existing form element is deleted, the test case should be updated to skip the commands involving that element. Modifications of a form element can be considered as a combination of the addition of the new version of the element and deletion of the old one. We call the problem of updating test cases for all such changes as *form data problem*.

Modification of the text or HTML contained in a DOM node leads to **content changes**. Such changes affect the assert commands that check the content of that particular node and leads to an assertion failure. We call this problem as *obsolete content problem*, which is typically repaired by the developer by updating the expected value with the actual value of the test case. An engaged reader might notice that some of the problems have an overlap in the kind of changes they notice in the DOM tree. For example, a test case assertion failure can either be due to the mis-selection problem or because of the obsolete content problem. In our technique we handle such overlaps by using a relative ordering of repairs that the technique attempts to apply. For the discussed example, if the test repair technique cannot apply a repair for the mis-selection problem, it will attempt to repair for obsolete content.

Often there is a server-side change of the web application that cannot be directly observed in the DOM tree. We call such changes **blind changes** as they are not seen by the test scripts on the client side and therefore cannot be possibly repaired from the client side alone. Examples of these are business logic changes where the inputs result in an error or failure message dialog shown on the web page but a black box technique would not have any guidance on how to change the inputs for getting rid of the message dialog. This paper does not address blind changes and we plan to consider it in future work.

3. APPROACH

The goal of our technique is to repair web application test cases. For achieving this goal, the technique detects broken parts of a test

```
CHANGE A
6   <input id="searchBtn" type="submit" value="Search">

CHANGE B
4 <form>
5   <input type="text" name="q"/>
6   <input id="search" type="submit" value="Search">
+   <a href="adv.php">Advanced Search</a>
7 </form>

CHANGE C
 9   <a href="http://www.gatech.edu/">
10     Georgia Institute of Technology (GATech)
11   </a>

CHANGE D
++ <div id="container">
 8   <div>
 9     <a href="http://www.gatech.edu/">
10       Georgia Institute of Technology
11     </a>
12     <p>Includes offices, departments, news
13        room, professional education...</p>
14     <span>www.gatech.edu</span>
15   </div>
16   <div>...</div>
17   <div>...</div>
++ </div>
```

Figure 4: Sample changes in *My Web Search*.

case, suggests repairs for them and validates if the repairs actually make the test case pass. We describe these activities in the following two phases of our technique:

3.1 Test Data Collection

The technique accepts a test case along with the old and new versions of the application, such that the test case passes in the old version of the application and fails in the new one. For collecting test data, the technique first runs the test case on the old version of the web application and collects data form the browser's DOM tree for each test case command. Our technique collects ten properties for each node in the DOM tree as shown in Table 2. The first five properties are used by test automation systems to select a particular DOM node[2]. The other properties are generated values by the browser as a result of applying client side scripts and styles. These properties are used in the next subsection for suggesting repairs.

After obtaining the DOM data for the old version, the technique obtains the same information for the new version of the application until the broken command results in test execution to abort. It also saves the details of the broken test case command along with the error/failure message. If the broken command was due to an assertion failure, the actual value of assertion is also stored. These details are then provided to the next phase which analyzes the data collected to suggests repairs.

3.2 Suggesting Repairs

The data collected in the previous step helps our technique identify the broken test case commands and obtain the error/failure information. Using this information, this step of the technique suggests repair to be applied to repair the test case. To perform this task, Algorithm 1 is used, which consists of three main functions, namely `SuggestRepairs`, `RepairLocators` and `GetSimilarityIndex`.

The function `SuggestRepairs` is the main entry point for this step and is responsible for suggesting repairs for the test cases. It accepts as inputs the test case (TC), the line number of the broken command in the test, DOM information from every command in the test case on both browsers, error/failure messages and optionally the actual value, in case of assertion failures resulting from mismatch from the expected value. This function attempts to address the four problems discussed in Section 2, namely *non-selection*

[2]`http://seleniumhq.org/docs/02_selenium_ide.html#locating-elements`

Algorithm 1: Suggesting Repairs For Broken Tests.

```
/* SuggestRepairs */
Input  : $TC, i$ : Test case and index of broken command
         $D_o, D_n$: DOM tree collection for each command in TC for old
                and new versions of application
         $msg, val$ : Error or failure message and optional actual value
Output: $Repairs$ : Prioritized list of suggested repairs
1  begin
2      $Repairs \leftarrow$ new list initialized to $\emptyset$
3      if $(msg$ = 'Locator error'$)$ then
           /* Fix Locator */
4          $Repairs \leftarrow RepairLocators(TC, i, D_o[i], D_n[i])$
5      else if $(msg$ = 'Assertion failure'$)$ then
           /* Fix Locator */
6          $Repairs \leftarrow RepairLocators(TC, i, D_o[i], D_n[i])$
7          if $Repairs \neq \emptyset$ then
8              return $Repairs$
           /* Fix Assertion */
9          if $(TC[i]$ is a 'Value Assertion'$)$ then
10             $TC[i].expected\_value \leftarrow val$
11         else
12             $TC[i] = negateAssertion(TC[i])$
13         $Repairs.add(CheckRepair(TC, i))$
14     else
           /* Populate Form */
15         for $j \leftarrow 0$ to $i$ do
16             if $(TC[j]$ is a "FORM_REQUEST"$)$ then
17                 $new\_elements \leftarrow FormDiff(D_o[j], D_n[j])$
18                 foreach $(e$ in $new\_elements)$ do
19                     $TC.insertAt(j, selectRandom(e))$
20                 $Repairs.add(CheckRepair(TC, i))$
21     if $Repairs = \emptyset$ then
22         $TC.remove(i)$
23         $Repairs.add(CheckRepair(TC, i))$
24     return $Repairs$
```

```
/* RepairLocators */
Input  : $TC, i$ : Test case and index of broken command
         $d_o, d_n$ : DOM trees for broken command in old and new versions
               of the application
Output: $Repairs$ : Suggested Locator Repairs
1  begin
2      $Repairs, matches \leftarrow$ new list initialized to $\emptyset$
3      $node_o \leftarrow GetNodeByLocator(d_o, TC[i].locator)$
4      foreach $prop$ in {"id", "xpath", "class", "linkText", "name"} do
5          if $node_o.prop \neq$ "" then
6              $matches.add(GetNodesByProperty(d_n, prop))$
7      foreach $(node$ in $matches)$ do
8          $TC[i].locator \leftarrow node.xpath$
9          $Repairs.add(CheckRepair(TC, i))$
10     if $Repairs = \emptyset$ then
11         $similarNodes \leftarrow$ new ordered list initialized to $\emptyset$
12         foreach $node$ in $d_n$ do
13             $sI \leftarrow GetSimilarityIndex(node_o, node)$
14             if $sI > 0.5$ then
15                 $similarNodes.add(node, sI)$
16         foreach $node$ in $similarNodes$ do
17             $TC[i].locator \leftarrow node.xpath$
18             $Repairs.add(CheckRepair(TC, i))$
19     return $Repairs$
```

```
/* GetSimilarityIndex */
Input  : $a, b$ where $a \in d_o, b \in d_n$
Output: $\rho$ (Similarity Index)
1  begin
2      $\alpha \leftarrow 0.9$
3      $\rho, \rho_1, \rho_2 \leftarrow 0$
4      if $a.tagname == b.tagname$ then
5          $\rho_1 \leftarrow (1 - LevenshteinDistance(a.xpath, b.xpath)/$
                  $max(length(a.xpath), length(b.xpath)))$
6          foreach $prop$ in {"coord", "clickable", "visible", "zindex",
           "hash"} do
7              if $a.prop == b.prop$ then
8                  $\rho_2 \leftarrow \rho_2 + 1$
9          $\rho_2 \leftarrow \rho_2 / 5$
10         $\rho \leftarrow (\rho_1 * \alpha + \rho_2 * (1 - \alpha))$
11     return $\rho$
```

Property	Description
id	Unique identifier of the DOM node, if defined
xpath	X-Path of the node in the DOM structure
class	CSS style class of a DOM node
linkText	The text content contained, in case of link nodes
name	The name attribute, in case of form elements
tagname	Name of the tag associated with the DOM element
coord	Absolute screen position of a DOM node
clickable	True if the DOM element has a click handler
visible	True if the DOM element is visible
zindex	DOM element's screen stack order, if defined
hash	Checksum of the node's textual content, if any

Table 2: DOM properties collected

problem, *mis-selection problem*, *form data problem* and *obsolete content problem.* At first, the technique attempts at lines 3–4, to fix locator errors which correspond to the non-selection problem. For this, it calls the routine `RepairLocators` to update the locator. It then handles assertion failures by addressing the mis-selection problem on lines 6–8 by using the same strategy as the one to fix locator errors to find the correct node. If unable to find a locator, the technique then addresses the obsolete content problem on lines 9–13. Here, it checks whether the assertion compares actual and expected values. If so, the expected value is replaced by the actual value. Otherwise, the assertion command is negated. If the test case passes after his change, the change is added to the list of suggested repairs. In case of all other errors, the technique attempts to address the form data problem on lines 15–20. To do so, it looks at all preceding commands to find form requests. At the point where the form request is sent, the technique attempts to find any newly added form elements by calling the `FormDiff` routine. For all such elements, the technique selects random values in these elements. The intuition here is that the form request that fails due to the an empty value of the new element, might get through when a random value is selected. If no repair can be found, then on lines 21–23 the technique checks if removing the broken command makes the test case pass. This can be useful for cases where elements are no longer present on a web page. After this, the technique outputs the list of suggested repairs to developers.

The logic to repair locator errors is presented in the `RepairLocators` function. It takes as input the test case, line number of the broken command and DOM tree from the old and new versions of the applications. The output is a list of locator updates that replace the current locator to make the test pass on the new version of the application. On line 3, the technique uses the locator to extract the DOM node ($node_o$) from the old version's DOM tree. It then extracts the five properties, which used for locating elements, from $node_o$ and searches for them in the DOM tree of the new version of the web application (lines 4–6). For all the DOM tree nodes which match the locating properties, the technique updates the locator to match these nodes and verifies if that makes the test pass (lines 7–9). If no repair strategy is found in the previous steps, the technique finds nodes in the new DOM tree that are similar to $node_o$ by using `GetSimilarityIndex`, which is described in the next paragraph. Then on lines 16–18, the technique updates the locator in the broken command with that of the similar nodes and checks if this makes the test pass. All locator updates in this phase resulting in a test case pass are returned as suggestions.

The function `GetSimilarityIndex` computes the similarity of two nodes from different DOM trees. It takes the two nodes as input and outputs a similarity index, which is a value in range [0,1]. For computing the similarity index, the technique first checks if the nodes have the same tagname (line 4). Although different nodes can have the same tagname, we found that it was unusual for a node to change its tagname during evolution. So, if the tagnames do not match, the default value of ρ, zero, is returned and means that the nodes are not similar at all. If the tagnames match, the similarity index is calculated based on the xpath and other remaining node properties. The first component of the the similarity index uses the normalized Levenshtein distance between the xpaths of the two nodes and assigns its ones complement to ρ_1 (line 5). The second component is the fraction computed using the matching node properties (coord, clickable, visible, zindex, and hash), which is assigned to ρ_2 (lines 6–8). The algorithm combines these two components by a multiplying them with suitable weights. The first component is assigned a higher weight because the xpath of the node should be very similar across versions and because many nodes can have similar properties. Intuitively, the second component is only used to break a "tie" resulting from two different nodes with a similar xpath.

Apart from these three main functions, there are some utility functions used in the algorithm. Function `GetNodeByLocator` searches the DOM tree for a locator and returns the first node that matches it. The routine `GetNodesByProperty` searches against the property and returns all the nodes that match the property.The function `CheckRepair` confirms if a suggested change results in the test case pass completely or at least for the broken command. If the test case passes completely, the change indicates success and is returned. If the test fails at the broken command, the suggestion is discarded by returning *null*. In case the broken command passes after the change but the test breaks at a later command, the technique repeats again to repair that command.

4. EVALUATION

To assess the usefulness of our technique, we implemented it in a tool called WATER and used it to answer the following research questions:

RQ1: Can WATER suggest repairs for a significant number of the broken test scripts in a web application?

RQ2: Does WATER assist repair without providing too many wrong suggestions?

We describe three case studies for which WATER was used to repair outdated test scripts for newer versions of the evolving web application that they interact with. The subjects chosen for the case studies are all real world web applications with a public software configuration repository having test scripts developed by developers themselves.

4.1 Joomla CMS

Joomla [10] is an open source Content Management System (CMS), which is written in PHP and is used to publish online content. It is a fairly mature piece of software with 11 years of development and is very popular – It is used to power 2.7% of the entire web[3] and the latest stable version (1.6.3) has been downloaded over 78,000 times within a week from its release. Joomla has 42 test cases written using the selenium testing framework, which we considered for our study. We chose 4 successive version pairs for this study. Our choice was influenced by source code changes resulting in a broken test case and existence of patches to fix broken test cases. The top section of Table 3 shows the details of the joomla case study. Column 1 shows the old and new version numbers of the web application considered. The column T_{broken} shows the number of test cases broken in the new version and C_{fail} shows the number of commands in each of these that are broken. For these broken commands, WATER suggested some repairs, the average of which has been shown in as shown in the column S_{avg}. The number of test case repairs that actually corresponded to fixes later made by the developers have been listed down in column T_{fix} and the test case version where the fix was found are presented in the V_{fix} column.

For the first version pair, there were two test cases that were broken. Each test had four commands that were broken, for each of

[3]http://www.joomla.org/about-joomla.html

$Versions$	T_{broken}	C_{fail}	S_{avg}	T_{fix}	V_{fix}
19478 – 19480	2	(4,4)	(3,3)	2	19484
20430 – 20431	1	(1)	(1)	1	20448
20739 – 20740	1	(1)	(0)	0	20776
20769 – 20770	2	(1,1)	(1,1)	2	20777

$Versions$	T_{broken}	C_{fail}	S_{avg}	T_{fix}	V_{fix}
963271 – 963410	2	(1,1)	(1,1)	1	v1.7
997469 – 997470	1	(5)	(1.4)	1	v1.29

Table 3: Joomla and Ofbiz Case Study

which WATER provided around 2 to 4 (average 3) suggestions. For these tests, the failure was in the oracle which is an `assertElementPresent` command to check if an HTML element identified by a locator was present on the page or not. All the suggestions were similar HTML elements present on the test. Thus, the commands with the suggested changes passed and all of them were selected as valid suggestions. Both these broken test cases were fixed in version 19484 of the tests. The third pair of applications had one command in a test failing in the new version. The change was in the option of a HTML drop down (select) element and a particular value needed to be selected for the test case command to proceed.

4.2 Apache Ofbiz

Apache Ofbiz (Open for Business) [2] is an open source suite of enterprise applications to integrate and automate business processes. It is written using the Java programming language and has common architectural components that are shared across the different applications. We obtained 16 selenium tests and chose two pair of versions for our case study as shown in the lower section of Table 3.

For the first version pair of the applications, there were two broken tests, each of which contained one broken command. The first broken test case was due to business logic error which resulted in a error message being displayed and the test used `assertElementNotPresent` to check for the non-existance of the error. The suggestion generated for this by WATER was to negate the operation and to check for the existence. Although this change would make the test pass, we believe that the developer would not repair the test case and will instead make the desired change in the source code to fix the bug. For all other broken test cases, the tool could suggest repairs that actually corresponded to the ones made by the developers.

4.3 CoScripter User Scripts

CoScripter [7, 8] is a browser based macro capture and replay tool developed by IBM Research. It is mainly targeted towards end users who can create and share such web macros with other users through the tool's website. The scripts created by this tool are similar to web application tests, except that they don't have assertion checks or oracles. This rules out assertion failures for these automation scripts. For our study, we extracted all the 5123 scripts from their website and ran them on a regular basis to find errors due to changes in the websites. Out of these scripts, 180 ran successfully on day one and five of these were found to be broken after 4 months. These scripts have been shown in Table 4 and were translated in a semi- automated manner to selenium test scripts.

WATER was able to fix three out of these five tests. The first web page contained a form that was completely changed, with three broken commands. WATER suggested repairs for the first two commands but could not make any suggestions for the third one where the form submit button was changed completely. For the second web page, the form was relocated on a different web page and required one to click on a link to go to that page. Unlike the previous two studies, the scripts considered in this study had no assertion checks, leading to more suggestions in the case of script 13389.

Script	Domain	Changed	Fixed?	Suggestions
10043	careers.yahoo.com	Form	No	(15,15, 0)
10754	www.icade.fr	Form, Reloc.	No	0
11525	www.terra.com.br	Link text	Yes	285
13389	www.terra.com.br	Link text	Yes	285
18164	www.google.com.br	Link text	Yes	1

Table 4: CoScripter Case Study

4.4 Discussion

As presented in both case studies, WATER was able to suggest repairs to broken test cases (RQ1). For the first two, the suggestions corresponded to real repairs made by developers later. As far as RQ2 is concerned, WATER suggested 1-3 repairs for each broken command in the first two case studies and 1-285 repairs for the third one. The wrong suggestions were due to the lack of a test oracle (assert) or due to a less strict one (as in the joomla case study) that resulted in the selection of many similar HTML elements for repairs.

We see two ways in our technique could be improved. First, additional information from the server- side script execution and across web pages can be used to make better suggestions for test repair and possibly prioritizing the suggestions. Secondly, further experimentation and feedback from users could help us confirm the usefulness of our technique over a wider spectrum of web applications and can pose additional research challenges for repair.

5. RELATED WORK

Daniel and colleagues have developed ReAssert [4] to repair unit tests by updating the declaration of the expected values with the actual values obtained from running the failing test. They later extended their technique [3] by using symbolic execution to handle control flow and cases involving operations on the variable defining the expected value. Our approach differs from theirs in two ways. First, we target the different domain of web application tests. Secondly, our technique can also repair tests that result in an error due to changes in the web page.

In the context of web applications, Alshahwan and Harman [1] have proposed a technique to repair session data during regression testing. Their technique repairs the user session data that becomes outdated due to changes to the web application. Their analysis considers changes to the structure, exposed interfaces and files contained in the application. For newly introduced parameters in the interfaces, their technique attempts to randomly pick values for such parameters. In case a web page in the test sequence has been deleted, their technique attempts to find alternate pages to repair the sequence. If it is unable to do so, it splits the test session into two. Our technique is focused on repairing real test cases that fail and focuses on in-page changes as mentioned in Section 2.

GUI Testing is another area with similar challenges due to their event driven nature. Unlike web applications, GUIs generally do not provide a hierarchy of elements and such information needs to be extracted from the GUIs. Memon and Soffa [9] have developed a technique to generate graph representations of GUIs to find unusable tests and then to repair them. Grechanik and colleagues [6] propose a technique to find differences across GUI version models. The test script is then analyzed using static analysis for assessing the impact of the differences and providing suggestions for changes to avoid possible failures in the GUI application.

In the absence of test oracles, detecting if a test case failed is an important problem. Dobolyi and Weimer [5] have developed a technique to detect errors in regression testing by using a precise comparator derived from other similar web applications. Their results show that their precise comparator is significantly better than diff based comparators currently used in practice. Roest and colleagues [12] use a similar approach on AJAX web applications by using a pipelined oracle comparator to ignore irrelevant differences for regression testing. In contrast to these technique, our approach knows when a test results in a failure or an error by executing the test case and suggests repairs to make it pass.

6. CONCLUSION

Web applications tend to evolve quickly resulting in broken test scripts that need to be updated to work with the newer version of the application. In this paper, we presented our technique and implementation WATER to automatically suggest repairs for such broken tests. Our experiments show that WATER can suggest repairs for broken web application tests effectively and does not provide too many wrong suggestions. We believe that this is a step in the right direction and that further experimentation and feedback from users can help us improve the technique.

7. REFERENCES

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 298–307, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Apache Foundation. The apache open for business project. `http://ofbiz.apache.org/`, Apr 2011.

[3] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 207–218, New York, NY, USA, 2010. ACM.

[4] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 433–444, Washington, DC, USA, 2009. IEEE Computer Society.

[5] K. Dobolyi and W. Weimer. Harnessing web-based application similarities to aid in regression testing. In *Proceedings of the 20th IEEE international conference on software reliability engineering*, ISSRE'09, pages 71–80, Piscataway, NJ, USA, 2009. IEEE Press.

[6] 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.

[7] IBM Research. Coscripter. `http://coscripter.researchlabs.ibm.com/coscripter/`, Apr 2011.

[8] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[9] A. M. Memon and M. L. Soffa. Regression testing of guis. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 118–127, New York, NY, USA, 2003. ACM.

[10] Open Source Matters, Inc. Joomla! `http://joomla.org/`, Apr 2011.

[11] OpenQA. Selenium web application testing system. `http://seleniumhq.org/`, May 2010.

[12] D. Roest, A. Mesbah, and A. v. Deursen. Regression testing ajax applications: Coping with dynamism. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 127 –136, 6-10 2010.