The Journal of Systems & Software 171 (2021) 110841

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An automated model-based approach to repair test suites of evolving web applications

Javaria Imtiaz *, Muhammad Zohaib Iqbal, Muhammad Uzair khan

Quest Lab, Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan
UAV Dependability Lab, National Center of Robotics and Automation (NCRA), Islamabad, Pakistan

ARTICLE INFO

Article history:
Received 12 March 2020
Received in revised form 21 September 2020
Accepted 22 September 2020
Available online 28 September 2020

Keywords:
Web testing
Automated test scripts
Regression testing
Model-based
Web test repair

ABSTRACT

Capture–Replay tools are widely used for the automated testing of web applications The scripts written for these Capture–Replay tools are strongly coupled with the web elements of web applications. These test scripts are sensitive to changes in web elements and require repairs as the web pages evolve. In this paper, we propose an automated model-based approach to repair the Capture–Replay test scripts that are broken due to such changes. Our approach repairs the test scripts that may be broken due to the breakages (e.g., broken locators, missing web elements) reported in the existing test breakage taxonomy. Our approach is based on a DOM-based strategy and is independent of the underlying Capture–Replay tool. We developed a tool to demonstrate the applicability of the approach. We perform an empirical study on seven subject applications. The results show that the approach successfully repairs the broken test scripts while maintaining the same DOM coverage and fault-finding capability. We also evaluate the usefulness of the repaired test scripts according to the opinion of professional testers. We conduct an experiment to compare our approach with the state-of-the-art DOM-based test repair approach, WATER. The comparison results show that our approach repairs more test breakages than WATER.

1. Introduction

Capture–Replay tools are commonly used by testers to perform functional testing of web applications. In the web application domain, testers perform end-to-end testing of their applications by creating test scripts using Capture–Replay tools such as Selenium (The Selenium Project. http://seleniumhq.org/docs/03_webdriver.html/), TestComplete (TestComplete. https://smartbear.com/product/testcomplete/), QTP (Unified Functional Testing (UFT). http://www8.hp.com/us/en/softwaresolutions/unified-functional-automated-testing), and Watir (Watir WebDriver. http://watirwebdriver.com). These test automation tools allow the testers to record the various test scenarios as a sequence of user actions (such as mouse clicks, keyboard entries, and navigation commands) performed on the web application and later replay (re-execute) these with the same or different data to test a web application.

Similar to other systems, web applications evolve over time. The evolution of web applications can introduce several different types of changes of web elements, for example, changes in visual representation due to new style sheets being applied, changes in visual labels for various fields, or distribution of fields on one page to multiple pages. The test scripts of Capture–Replay tools are strongly coupled with the elements of web pages and are very sensitive to any change in the web elements of the web application. Even simple changes, such as slight modifications in a web page layout may break the existing test scripts, because the web elements that the scripts are referencing may not be valid in the new version. A detailed taxonomy of various types of web application test breakages is presented by Imtiaz et al. (2019) and Hammoudi et al. (2016b). Fixing these test scripts manually requires a significant effort. According to a study (Memon and Soffa, 2003), such changes can require around 75% of the test suites to be fixed. This overhead is considered as a major obstacle for organizations to move towards automated web application testing (Chen et al., 2012).

There are approaches in the literature that focus on automating the test script repair for web applications (Stocco et al., 2018; Choudhary et al., 2011; Hammoudi et al., 2016a), however, these approaches are limited and do not cater for a large number of breakages that occur in practice as defined in the web test breakage taxonomy (Hammoudi et al., 2016b). The approach presented in Stocco et al. (2018) only focuses on fixing the breakages caused due to the change of field locator. Furthermore, this approach is not applicable in case of style sheet changes

* Corresponding author at: Quest Lab, Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan.
E-mail addresses: javaria.imtiaz@questlab.pk (J. Imtiaz), zohaib.iqbal@nu.edu.pk (M.Z. Iqbal), uzair.khan@nu.edu.pk (M.U. khan).

https://doi.org/10.1016/j.jss.2020.110841
0164-1212/

or when the visual representation of page elements is changed (i.e., a button change to an icon, etc.,) and hence produces many false positives. The DOM-based web test repair approach WATER (Choudhary et al., 2011), repairs the test scripts corresponding to the locator-based changes and the broken assertions. These are only two of the five different types of breakage categories that may occur in an application (Hammoudi et al., 2016b). For example, the breakages due to change in page layouts of web applications are very common, however, such breakages are not correctly repaired by WATER. Similarly, the approach does not repair the test cases that are broken due to change in XPath. The WATERFALL (Hammoudi et al., 2016a) approach builds on top of WATER and focuses on identifying and repairing breakages that occur during intermediate commits of a web application. However, the WATERFALL approach has the same limitation as the WATER approach and both of these approaches only focus on repairing the broken locators and assertions in test scripts.

In this paper, we propose a model-based automated approach to repair the broken test scripts of Capture–Replay testing tools. The proposed approach covers the various types of changes of web elements that may result in the breakage of test scripts. The work is based on the taxonomy by Hammoudi et al. (2016b) that defines all such changes of web elements that may occur in practice. No other existing work provides such a comprehensive test repair strategy. Our approach is also not specific to any particular Capture–Replay tool. To provide a tool independent approach, we develop a UML profile that allows the capture of various concepts related to Capture–Replay test scripts for web applications. For this purpose, we extend the UML Test Profile (UTP) (UML Test Profile (UTP), 2005) and add important concepts relevant to capture and replay test scripts of web applications. Our approach uses a DOM-based approach for capturing the differences between two versions of a web application. We define a number of repair strategies corresponding to different categories of test breakages. The repaired test models can be automatically transformed into test scripts of specific capture–replay tools, such as Selenium, QTP, and Watir. We also develop an open-source tool[1] that automates the proposed test repair approach.

We also evaluate the effectiveness of our proposed approach by conducting a series of four evaluations on one industrial and six open-source case studies. Our first experiment focuses on evaluating the effectiveness of the approach in terms of the number of broken test scripts that are repaired and the comparison of achieved DOM coverage of repaired and original test suite. The second experiment uses mutation analysis to compare the fault-finding capability of the repaired test scripts with the original test suite. The third evaluation focuses on the usefulness of the repair of the broken test scripts. For this, we asked a pool of experts to inspect the test repairs done by the proposed approach. In the fourth evaluation, we performed a comparison in terms of the number of test script repairs of our approach with a state-of-the-art DOM-based web test repair approach (WATER (Choudhary et al., 2011)) on the seven case studies.

Our results indicate that the proposed approach effectively repairs the 91% of broken web test scripts and achieves a similar DOM-coverage, as by the original test suite, on the evolved versions of subject applications. A team of professional testers found the suggested repairs, for different types of test breakages, useful for the regression testing of evolving web applications. Furthermore, the DOM-based fault-finding capability of the repaired test suite is equivalent to the original test suite. Our empirical evaluation on 528 Selenium web driver test scripts of seven web applications shows that the proposed can effectively repair 83% of the overall breakages, whereas the existing technique WATER repairs 58% test breakages which only includes attribute-based locators and broken assertion values.

To summarize, the main contributions of this paper are:

(1) We develop an approach for automated repair of Capture–Replay test scripts that break due to the changes in user interfaces of evolving web applications.
(2) To make our approach generic, we develop a DOM difference meta-model for the representation of DOM changes detected between earlier and the modified versions of the same web application.
(3) We extend the UML Testing Profile (UTP) (UML Test Profile (UTP), 2005) to capturing the details of capture–replay web test scripts and develop a corresponding UML profile for modeling the action sequences encoded in automated test scripts to provide tool independent repair solutions.
(4) We devise a strategy to classify the existing test suite into reusable, retestable, and obsolete test scripts as per (Leung and White, 1989).
(5) We design a set of heuristics to repair the test scripts covering all the various types of breakage categories defined in web test breakage taxonomy (Hammoudi et al., 2016b).
(6) We develop an open-source tool[2] to automate our approach for repairing the broken web test scripts.
(7) We perform a series of empirical evaluations on an industrial and six open-source case studies to evaluate the effectiveness of our proposed approach in terms of repairing the broken test scripts, coverage of DOM elements, and fault-finding capability. We also evaluate the usefulness of the repaired test scripts according to the opinion of professional testers. We also compare our approach with the only other available DOM-based test repair approach, WATER (Choudhary et al., 2011).

The rest of the paper is organized as follows: Section 2 explains the background and running example of test breakages of evolving web applications. Section 3 defines our model-based test script repair approach for evolving web applications. Section 4 presents the tool implementation of our approach. Section 5 presents the empirical evaluation of the proposed approach. Section 6 discusses the threat to the validity of our results. Section 7 compares our work with related research and Section 8 presents the conclusion.

2. Background and running example

In this section, we provide a background of web test breakages and also discuss a running example that will be used throughout the paper.

2.1. Background on web test breakages

The most widely known approach for test automation is capture–replay testing, in which a tester performs the test steps using testing tools (e.g., Selenium (The Selenium Project. http://seleniumhq.org/docs/03_webdriver.html/), TestComplete (TestComplete. https://smartbear.com/product/testcomplete/), Watir (Watir WebDriver. http://watirwebdriver.com), and QTP (Unified Functional Testing (UFT). http://www8.hp.com/us/en/softwaresolutions/unified-functional-automated-testing)). The tools record the tester's actions and generate a corresponding test script that replays the actions automatically. The script encodes the test steps in the form of tuples *<locator, action, value>*. In Capture–Replay tools, the target web elements are identified using different *locators*. There are two types of locators used to identify

[1] https://github.com/javariaimtiaz12/Model-basedWebTestRepairTool.

[2] https://github.com/javariaimtiaz12/Model-basedWebTestRepairTool.

UI elements on the web pages: (i) attribute-based (e.g. id, name) and (ii) structure-based locators (e.g. XPaths, CSS). The *action* in a test script specifies the type of action performed on the web element (e.g. click, select, enter). The *value* refers to the input value entered by the testers or the text expected on the web page.

Capture–Replay test scripts are prone to changes when web applications evolve. It becomes a challenging situation for the testers to deal with the rapid evolution of the web application under test, because even small layout changes may lead to breakages in the test scripts. Changes like addition, deletion of web elements, or repositioning of web elements typically cause the Capture–Replay test scripts to fail on the evolved version. Hammoudi et al. develop a detailed taxonomy that discusses the various types of web test breakages (Hammoudi et al., 2016b). The taxonomy is widely used as a reference in web test repair literature (Imtiaz et al., 2019). This taxonomy is developed by considering 453 versions of eight different web applications and resulted in a catalog containing the types of test breakages. This taxonomy is broadly classified into five distinct categories of types of test breakages, as follows:

i. Breakages related to locators used in tests
ii. Breakages related to values and actions used in tests
iii. Breakages related to page reloading
iv. Breakages related to changes in user session times
v. Breakages related to popup boxes

The first category of test breakages is related to *locator-based changes* and the breakage causes in this category are related to locators. For example, the value of an attribute of a web element (such as, id or name) that was being used to locate the web element in the original web page has been changed in the new web page.

The second category of breakages is linked with *value-based changes* when the input value used by the original test is no longer accepted on the evolved web page, e.g., a dropdown option that was previously selected by a test is not available in the modified web page.

The third category of test breakages is related to *page-reloading activities*. For example, some test scripts fail when the application takes too much time to load the user interface. The test breakages occur when the wait time specified in the test script is not enough to properly load and display the content of a web page.

The fourth category is related to *user session timeouts*. This problem occurs when the web page becomes inactive during testing after *n* minutes of inactivity. For example, a web application has a preset session timeout of 30 min while the test suite running on this application takes almost one hour to complete its execution. During testing when the application logs out after 30 min, the test scripts fail.

The fifth category is related to the *unexpected occurrences of popup boxes* in web applications. For example, unexpected alerts or popup boxes appear to display errors or other forms of messages.

Web applications are expected to run on multiple web browsers (e.g., Chrome, Firefox, Safari, Internet Explorer, Edge) with different browser versions. It is a common problem in a web test automation that test scripts executes successfully against one browser but fails on other due to compatibility issues. This is due to the difference in rendering the HTML, CSS, and other scripts that may lead to test breakages. Similarly, different browsers may have different loading time that may cause test breakages.

2.2. Running example

We use Collabtive,[3] an open-source web-based project management system, as a running example. Collabtive is a good representative example of modern-day web applications as it is a multi-page application and consists of a variety of web elements. The application is used as a collaboration platform among team members working in an organization. It helps to plan tasks, design milestones, perform time tracking of activities to complete a project. For this paper, we are considering the earliest (0.6.5) and the modified (1.0.0) versions of Collabtive case study to highlight the HTML changes which can cause test breakages. Fig. 1 shows the "Add User" web page of the previous version (0.6.5) and its corresponding test script. This test script is recorded using Selenium IDE and consists of various test commands. The first command gets the URL of the page under test and loads it in a web browser (Line 1). The test then fills the various fields, including 'name', 'email', 'pass', 'rate', and select the 'role' of a new user. At the end (Line 8), an assertion is placed to verify that a new user has been added successfully into a database.

When the same test script is executed on an evolved version of the web page (version 1.0.0), shown in Fig. 2, the original test first stops at Line 4 while attempting to locate the "id=pass" on the evolved page. This test command failed due to the change in identifier of text field when <input "id=pass" …></input> becomes <input "id =password" …></input>. This trivial locator-based change may fail the whole test script to run on the evolved version. Another breakage occurs where radio buttons of the original version are replaced by a dropdown list in the evolved version. In this particular case, the original test script will break at Line 6 when attempting to click on the radio button with the 'role3' id (Project Manager). Similarly, if all else goes well, the test script will stop at Line 8 due to an unexpected assertion value, since the evolved version is expected to display a message "Added Successfully" rather than the original message "User Added" on the screen. This broken test script, due to locator-based changes, element type change, and assertion failures requires modifications to work on the latest version. Fig. 2 shows the corresponding repaired test scripts by making small changes in the original test script.

Manually making such modifications in the test scripts by comparing them manually with the web page elements is a laborious and time-consuming activity. Such changes are very common in web applications and frequently require modifying the test scripts. Our proposed work targets such breakages and provide an automated approach for fixing the test scripts to make them executable on the evolved versions.

3. Model-based test script repair approach

In this section, we present our automated test repair approach for evolving web applications. Fig. 3 shows an overview of the proposed approach. This approach takes the two different versions of a web application and a test suite for the earlier version of the application as an input to generate repaired test scripts for the evolved version. The approach is based on well-accepted principles of meta-modeling and is independent of the underlying Capture–Replay tools. Our approach involves five key automated phases: (1) Webpage difference detection, (2) A tool-independent representation of Capture–Replay test scripts, (3) Test suite classification 4) Test repair strategy and (5) Transforming test models to Capture–Replay tool-specific test scripts. The various phases of the approach are discussed in the following subsections.

3.1. Webpage difference detection

In this phase, the two versions of a web application (original and evolved version) are compared based on the DOM and a difference model containing the various type of modifications in the evolved web application is populated. It captures the DOM

[3] https://sourceforge.net/projects/collabtive/.

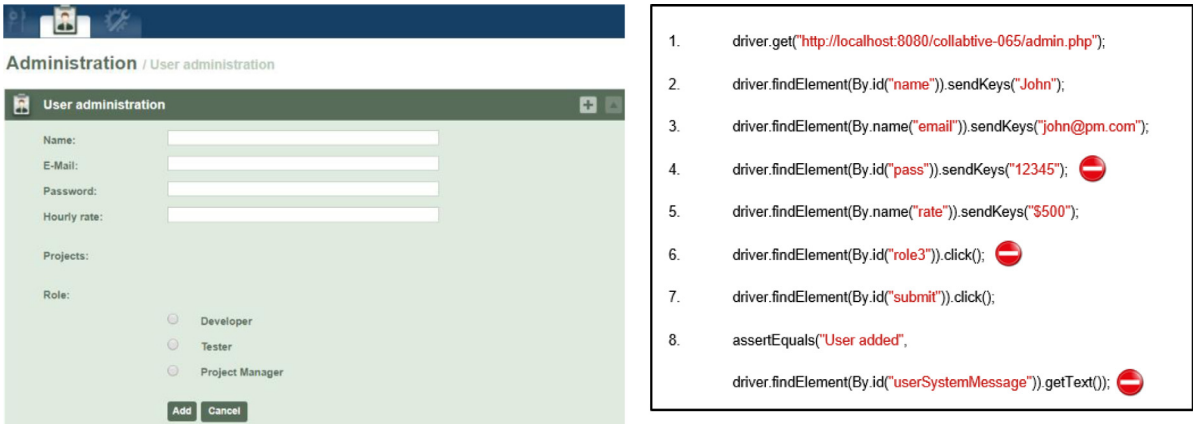

Fig. 1. Collabtive Case Study (Version 0.6.5) along with Selenium IDE tests.

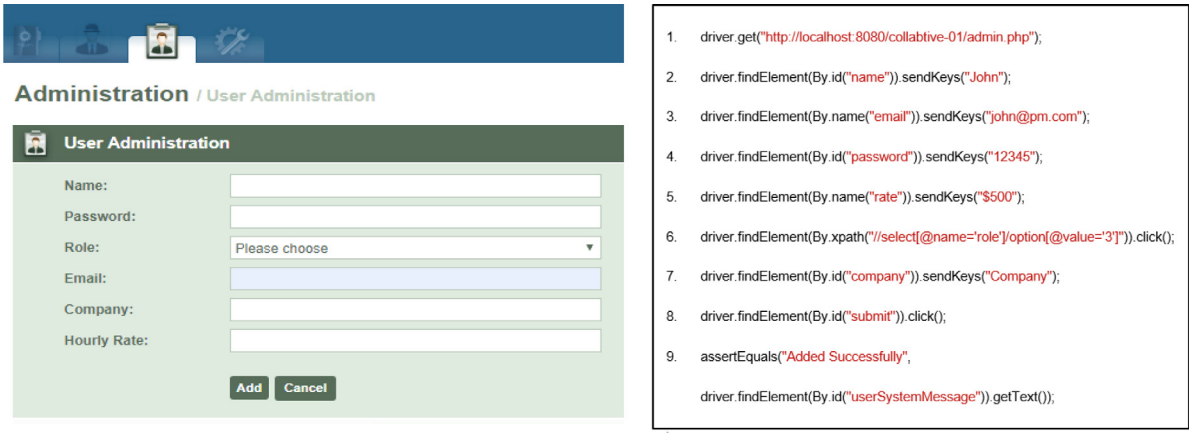

Fig. 2. Collabtive Case Study (Version 1.0.0) along with Selenium IDE tests.

of the modified page and compares it with the DOM capture of a previous version of the page. Fig. 4 depicts the DOM-based comparison between two versions of the web page. The DOM elements from the original page are compared to elements from the evolved by matching their unique attributes, positions, and similarity measures.

Our goal is to determine the differences (which have an impact on tests) between two versions of the web application. To identify differences, we extended the DiffUtil (D.D.D.L., https://github.com/java-diff-utils/java-diff-utils) difference detection algorithm. The original algorithm provides a set of difference patches between the two versions. These patches primarily highlight the position changes of HTML blocks (div element). In our context, we require a more detailed comparison of the various HTML elements and are interested in all the changes that may cause a test breakage. These changes include modifications in existing DOM nodes (such as addition, deletion, or modification of web elements), web element attribute changes (e.g., modification in an identifier of a web element), label changes (e.g., the label of a widget is updated), and function-level changes (pop-up boxes).

Fig. 5 shows our difference detection algorithm. Our implementation uses state-of-the-art Crawljax (Mesbah et al., 2012) to parse the web pages from both web versions. It accepts the URL of a web application and automatically derives a navigational model of the dynamic DOM states and transitions between them by crawling the application. For each version, we executed Crawljax to capture and record the behavior of applications in the form of navigational models. The crawling is performed in a similar fashion using the same depth and crawling time for both applications to extract the navigational model. The generated models are then compared with each other to get the similar web pages. To compute the similarity between web pages or to ensure the page-to-page mapping between the two models, we use the Cosine Similarity Measure. This criterion is used by many existing studies (Lin et al., 2017; Rau et al., 2018; De Lucia et al., 2007) and reported as a good measure for checking semantic equivalence. The cosine-similarity ($\cos(\theta)$) of two given word vectors is defined as vector V and V' in the range of $[-1, 1]$. Negative values express a high dissimilarity whereas values close to positive numbers express similarity. In our case, Cosine Similarity Measure is normalized from -1 (when the web pages are semantically inequivalent) to 1 (when both web pages are semantically equivalent). Therefore, for the given two web pages wp and wp', the similarity is defined as:

$$Similarity(V, V') = \frac{V \cdot V'}{\|V\| \times \|V'\|}$$

Here, V and V' are the vectors corresponding to the web pages wp and wp'. This step produces the sorted lists of matched web pages wp and wp' (Algorithm-1, Lines 1–2). After page-level comparison, we have to detect the elements in the first web page that have a corresponding element in the second page. We used JSoup (JSoup, 2019) for elements crawling i.e. web elements from each page in the form of D and D' (Algorithm-1, Lines 4–5). To identify differences between new and old versions of the web page, we extended the DiffUtil (D.D.D.L. https://github.com/java-diff-utils/java-diff-utils) difference detection algorithm. To extract such changes, we compare the pairs of matching widgets by comparing

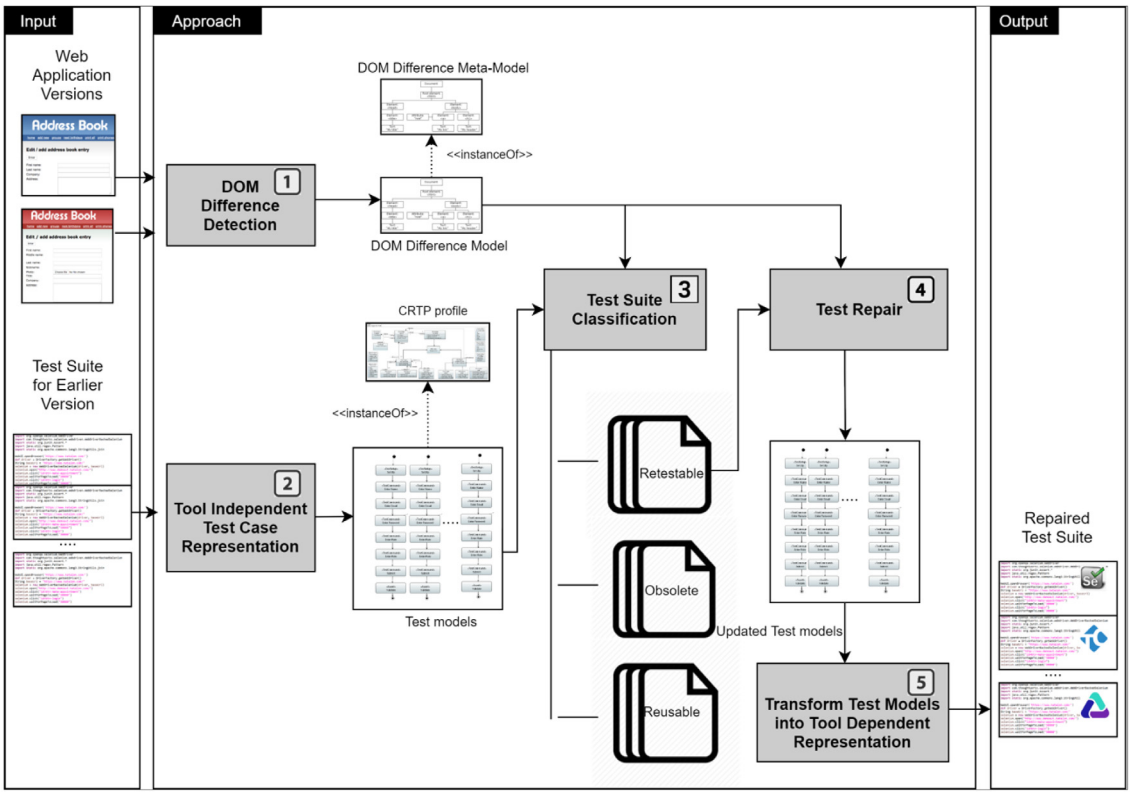

Fig. 3. Overview of our approach.

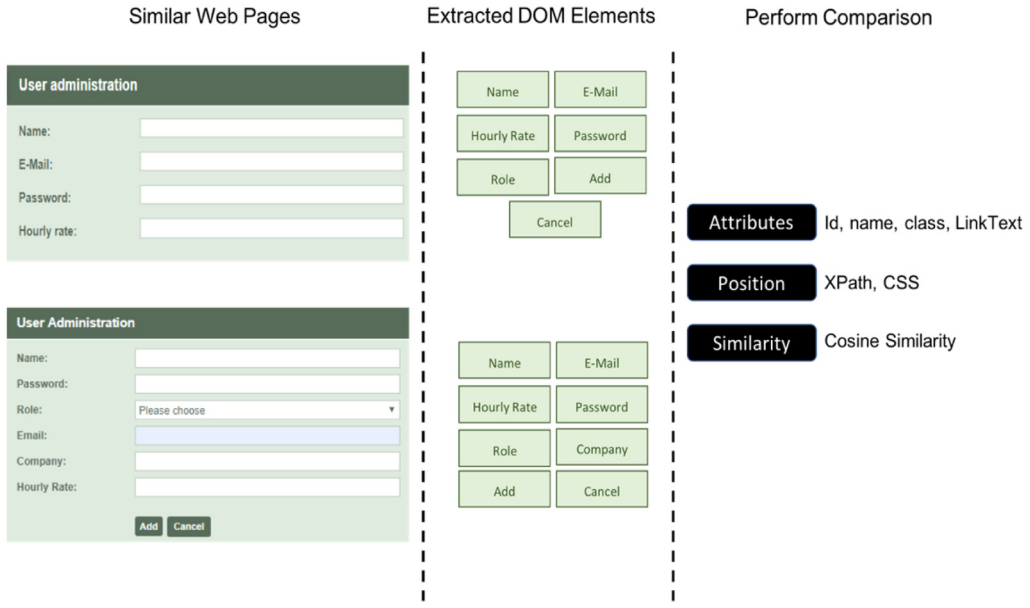

Fig. 4. DOM-based Comparison.

the respective locators (id, name, XPath, or class). Our algorithm not only relies on the locators due to their evolving nature. Cosine Similarity is used for determining the similarity of label and name attributes to compare the DOM elements in the similar web pages (Algorithm-1, Line 8) (Hartmann et al., 2008). For all the similar web elements in both web pages, it further compares attributes, text, functions, DOM hierarchy, and element options in both the versions (Algorithm-1, Lines 9–12). In the case of a newly created element, it is automatically added to the changeset, as no matching already existing element can be found.

Once the difference is detected, the resultant changeset is a difference (containing original and modifies values corresponding to changed elements) and the ChangeCategory which refers to the type of edit operation (e.g. added element, deleted element, and changed element) performed on the evolved version (Algorithm-1, Lines 13–18). At completion, this algorithm returns a DOM difference model corresponding to the modifications (Algorithm-1, Line 22).

We capture the difference information as a model to provide an automated strategy. For this purpose, we define a DOM

Algorithm-1 DifferenceAnalyzer

Input: W_o: original web application
W_m: modified web application
Output: *DDM:* DOM Difference Meta-model

```
Begin: DifferenceAnalyzer
1.  wp [] ← parseHTMLPages(Wo)
2.  wp'[]←parseHTMLPages (Wm) //sort pages according to similarity index
3.  for i ←1 to wp.size() do
4.      D ← extractDomElements(wp[i])
5.      D' ← extractDomElements(wp'[i])
6.    foreach e in D do
7.      foreach e' in D' do
8.        if (e.widget.isSameNode(e'.widget')) then //match widgets using cosine similarity index
9.          foreach locator in {"id", "name", "XPath", "class", "linkText"} do
10.           if (widget.locator == widget'.locator) then
11.             compare value and actions
12.             compare structure-based locators
13.             DDM.add(difference, ChangeCategory)
14.           else
15.             DDM.add(difference, ChangeCategory)
16.         end for
17.       else
18.           DDM.add(difference, ChangeCategory)
19.     end for
20.   end for
21.  end for
22.  return DDM
end DifferenceAnalyzer
```

Fig. 5. DifferenceAnalyzer Algorithm.

Difference Meta-model (see Fig. 6). The difference meta-model includes all the necessary concepts of DOM which can affect the automated test scripts. The difference meta-model can describe additions, deletions, and modifications of different types of DOM elements. Our proposed meta-model captures the various changes in the two versions of webpages that may cause test breakages, including, *addition, deletion, and change* in web elements, their attributes, or functions. The differences detected are populated as an instance of the DOM Difference Meta-model, referred to as DOM Difference Model (DDM). The DDM is composed of various concepts such as WebPage, WebElement, ElementType, Attribute, Function, Option, and Text. A DOM object is represented by the web page. A web page is composed of various web elements such as input fields, buttons, and links, etc.

Web elements may have specific different types such as text boxes, password fields, checkboxes, radio buttons, dropdowns, file inputs, date picker, calendar, etc. Web elements can have options/menu to allow the user to select one value from the predefined list. A web element is uniquely identified by its attributes within a web page. Attributes define the DOM properties of the web elements like ID, Name, Class, XPath, TagName, CSS Selectors, link Text, etc. Functions represent the DOM events that occur due to user action or state change of elements in the DOM tree.

Fig. 7 shows the difference identified as a DDM between two versions of the running example, Collabtive (0.6.5) and Collabtive (1.0.0) represented as an instance model of the DOM difference meta-model. All webpages that are changed have corresponding objects of the class 'ChangedWebPage'. All the various elements that were present in the original webpage but were deleted from the evolved webpage are populated as instances of the 'DeletedElement' class. For example, Role radio buttons were present in the original version, but are not present in the evolved Admin webpage of Collabtive (version 1.0.0). All the elements in the original webpage for which any of the attributes are changed are populated as instances of 'ChangedElement' class and the exact changes are populated as instances of 'ChangedAttribute' class. For example, some structural features (attributes) of the Password have been modified in the evolved version Collabtive (0.6.5) web page. Similarly, all the elements that are present in the evolved version, but were not included in the original version are populated as instances of 'AddedElement' class. For example, the Company text field and Role Dropdown list which were not present in the original version of the Collabtive (0.6.5) Admin web page.

3.2. A tool-independent representation for Capture–Replay test scripts

An important phase of our approach is to transform the test scripts written as Capture–Replay Test Scripts to a tool-independent representation. Such a representation allows an approach to apply on a large variety of capture–replay tools for web applications, including, Selenium (The Selenium Project. http://seleniumhq.org/docs/03_webdriver.html/), TestComplete (TestComplete. https://smartbear.com/product/testcomplete/), Watir (Watir WebDriver. http://watirwebdriver.com), and QTP (Unified Functional Testing (UFT). http://www8.hp.com/us/en/softwaresolutions/unified-functional-automated-testing). For this purpose, we extend the UML Testing Profile (UTP) 2.0 (UML Test Profile (UTP), 2005) and UML (Unified Modeling language (UML), v. 30th - Nov. Available: http://www.omg.org/spec/UML/2.4.1/) to include the concepts relevant to automated web application testing. Our proposed profile reuses a subset of UML and UTP concepts to create extensions as necessary to support the specific requirements of capture–replay testing tools. Fig. 8 shows the use of reusable packages from UML and UTP to add extensions to fully specify automated test scripts.

UTP is a modeling language based on UML to support the specification of test automation architecture. UTP is developed to capture the tests as a model and offers a wide range of concepts related to test design activities, test execution, and evaluation. This is developed by a working group at OMG and recently the

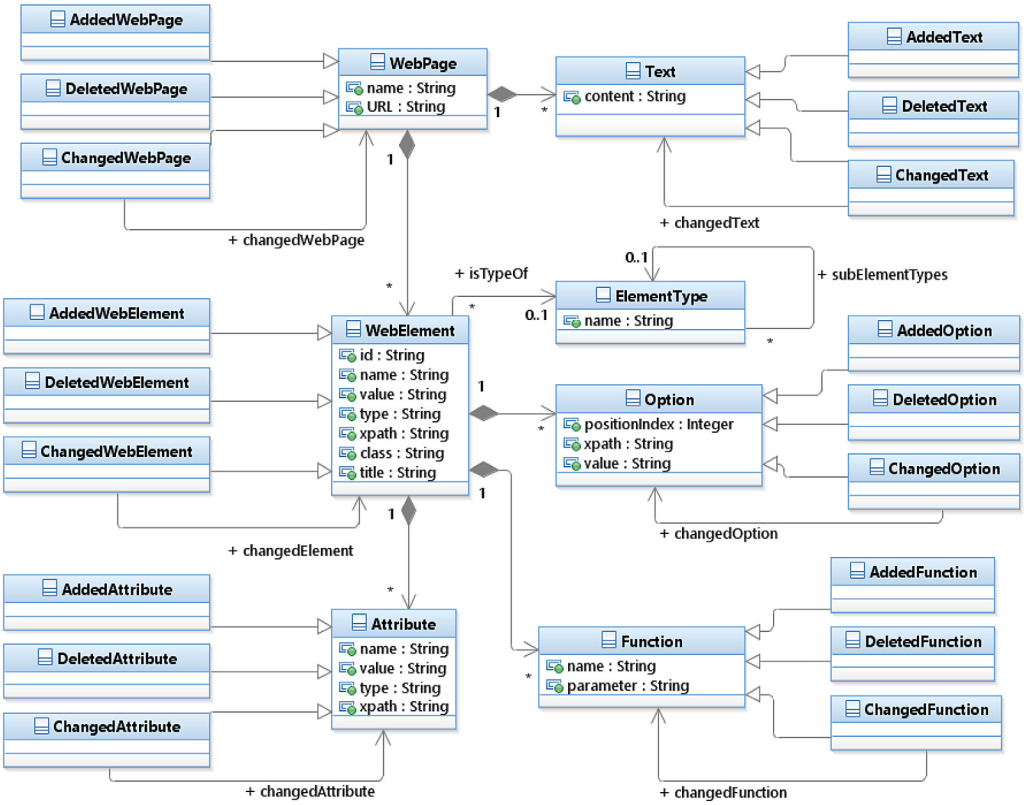

Fig. 6. DOM Difference Meta-model.

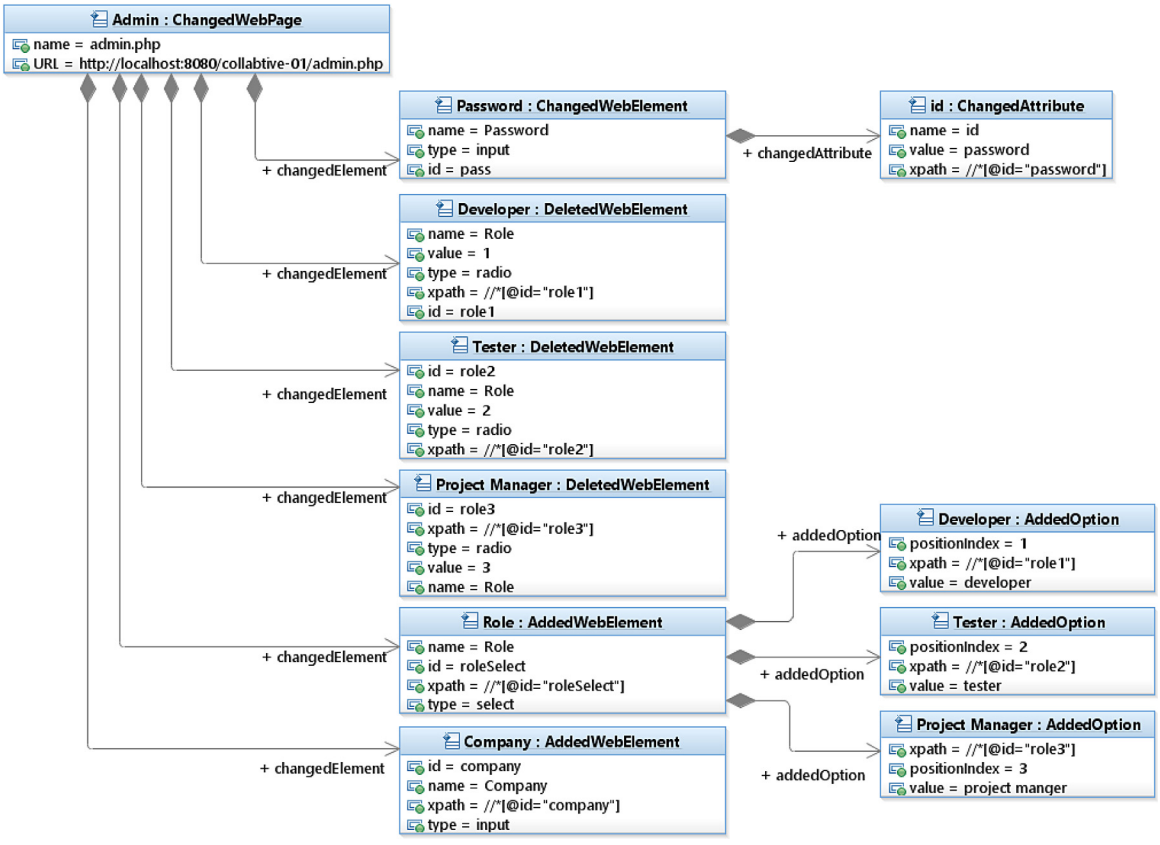

Fig. 7. DOM Difference model.

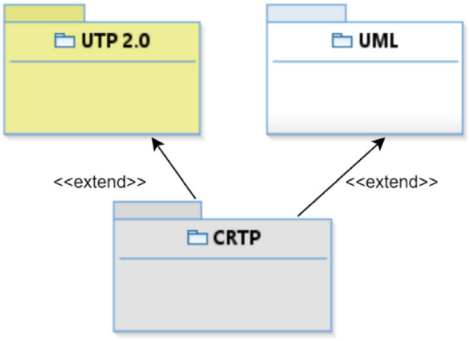

Fig. 8. CRTP extension of UML Activity Diagram and UML Testing Profile.

beta version of UTP 2.0 is available to the public. The core concepts for modeling the details of automated test scripts such as test platform, test behavior, and test evaluation are not elaborated in UTP. The provision of concepts that facilitates automated test activities is the primary intention of this version of UTP. Our extended profile includes the core concepts of UTP and several concepts that are required to capture the details of capture–replay test scripts. Fig. 9 shows an excerpt of the conceptual model of the proposed Capture–Replay Test Profile (CRTP) for modeling capture–replay tests for automated web application testing. It highlights the various concepts of UTP that are being extended by the CRTP. CRTP is organized into three packages Test Platform, Test Behavior, and Test Evaluation.

The *Test Platform* represents the test environment containing elements that are necessary to conduct test scripts. For capture–replay testing, the information of the web browser and operating system is substantial to set up the environment for test execution. In CRTP, these are defined as TestSetup which extends UTP's TestContext (allows to group test scripts, describe test configuration, and define execution order of test scripts), Test Component (define test objects communicate with System Under Test (SUT)), and Test Items (describe SUT).

The *Test Behavior* package describes the concepts required to specify test scripts, test data, and associated behaviors. For capture–replay testing, the behavior of test scripts is captured using test commands. The test command utilizes a tuple of locator, action, and value. For instance, "*driver.findElement(By.cssSelector ("input[name='username']")).sendKeys("user123");*" is a Selenium test command where CSS locator is used for locating the username textbox field, "sendkeys()" command is used for submitting the text into the textbox field and "user123" is the specified value to be entered into the text box. Therefore we introduced a new concept TestCommand for representing the test actions of capture–replay test scripts. This concept extends the Create Stimulus Action (defines test actions to submit a stimulus to the test item) and Test Procedure (defines operations consisting of a sequence of statements) to capture the action sequences of automated test scripts. The TestData extends the Property (defines input used by test scripts) to facilitate data-driven testing.

The *Test Evaluation* package defines the concepts used to evaluate the test results. For capture–replay testing, the expected results of the test scripts are usually verified through assertions. The VerificationStep extends ExpectResponseAction (describes the expected response of the SUT at a certain point in time during test execution, including the expected data) and ArbitrationSpecification (defines set of rules that calculate the eventual verdict of an executed test script). This action can be mapped to the verification step of automated test scripts such as assert, verify, and wait conditions.

UTP 2.0 does not allow the modeling of actions in the form of an activity diagram. Activity Diagrams are used to describe the workflow behavior of the system. In the case of a web application, an activity diagram shows the behavior and sequence of activities and actions. Thus, we extend the standard UML 2.0 to adequately express the concepts of capture–replay test scripts. We define a UML profile for capture–replay test scripts domain-specific concepts. We applied the profile definition methodology as suggested by Selic (Selic, 2007). The profile allows modeling the information contained in automated test scripts. Fig. 10 shows the excerpt of UML CRTP that we developed for modeling automated test web test scripts. The core three stereotypes of the profile are TestSetUp, TestCommand, and VerificationSteps that can be applied to the UML meta-class, Action.

The TestSetUp stereotype includes concepts to create the testing environment, for example, pageURL(path of the web application to navigate in the web browser), browser(to test the web application in multiple browsers), and driverWaitTime (indicate the amount of time for the WebDriver to wait for DOM elements to appear on the web page). The TestCommand stereotype contains the details of test actions performed on the web application. The set of test commands are used to run the automated test scripts. Each test command is composed of Locator, TestAction, and TestData. The Locator identifies the web element on the web page using a unique identifier (e.g. id, name, XPath, CSS). The TestAction specifies the action performed on that specific web elements. There are different types of test actions, for example, such as navigation actions (e.g. navigate, refresh), browser actions (e.g. getCurrentUrl, getTitle), web element actions (e.g. get, click, set). The TestData is the value entered by the user or through a data-driven mechanism in that specific widget. The VerificationStep stereotype validates the state of the web application to check whether it behaves the same as expected. This validation can be done through three ways Assert, Verify, and Wait. When an "Assert" fails, the test is aborted. When a "verify" fails, the test will continue execution, logging the failure. A "waitFor" command waits for some condition to become true. These three validation steps Assert, Verify, and Wait further contain many functions for validation of web application states. The important concepts of UML CRTP are also mentioned as stereotypes in Table 1. Testers can apply these stereotypes to the activity diagram to capture the specific details of Capture–Replay test scripts. The complete profile is downloadable from an open-source repository.[4]

3.2.1. Transformation from test scripts to CRTP models

The CRTP profile allows the information in capture–replay test scripts to be available as models. We parse the test scripts, written in a tool-specific language, and generated the corresponding CRTP profile models as UML Activity Diagrams. This step is performed using the Eclipse Modeling Framework (EMF) (Budinsky et al., 2004) tools that provide a modeling and code generation framework for the applications based on structured data models. To achieve this step, we implemented a component called ModelGenerator, whose task is to load all the test scripts and parse the necessary information in the mapping file. This file contains information about test objects (i.e. web elements), test actions, and verification commands. With the help of a mapping file, automatically generated corresponding activity diagrams using the EMF tools. The profile reader module uses CRTP to load the instance of the profile by applying necessary stereotypes. Fig. 11 shows the instance of CRTP for the original test script of the running example. Each test command of the Selenium test script is modeled using the extended action of an activity diagram. This activity diagram captures the details

[4] https://github.com/javariaimtiaz12/Capture-ReplayTestProfile.

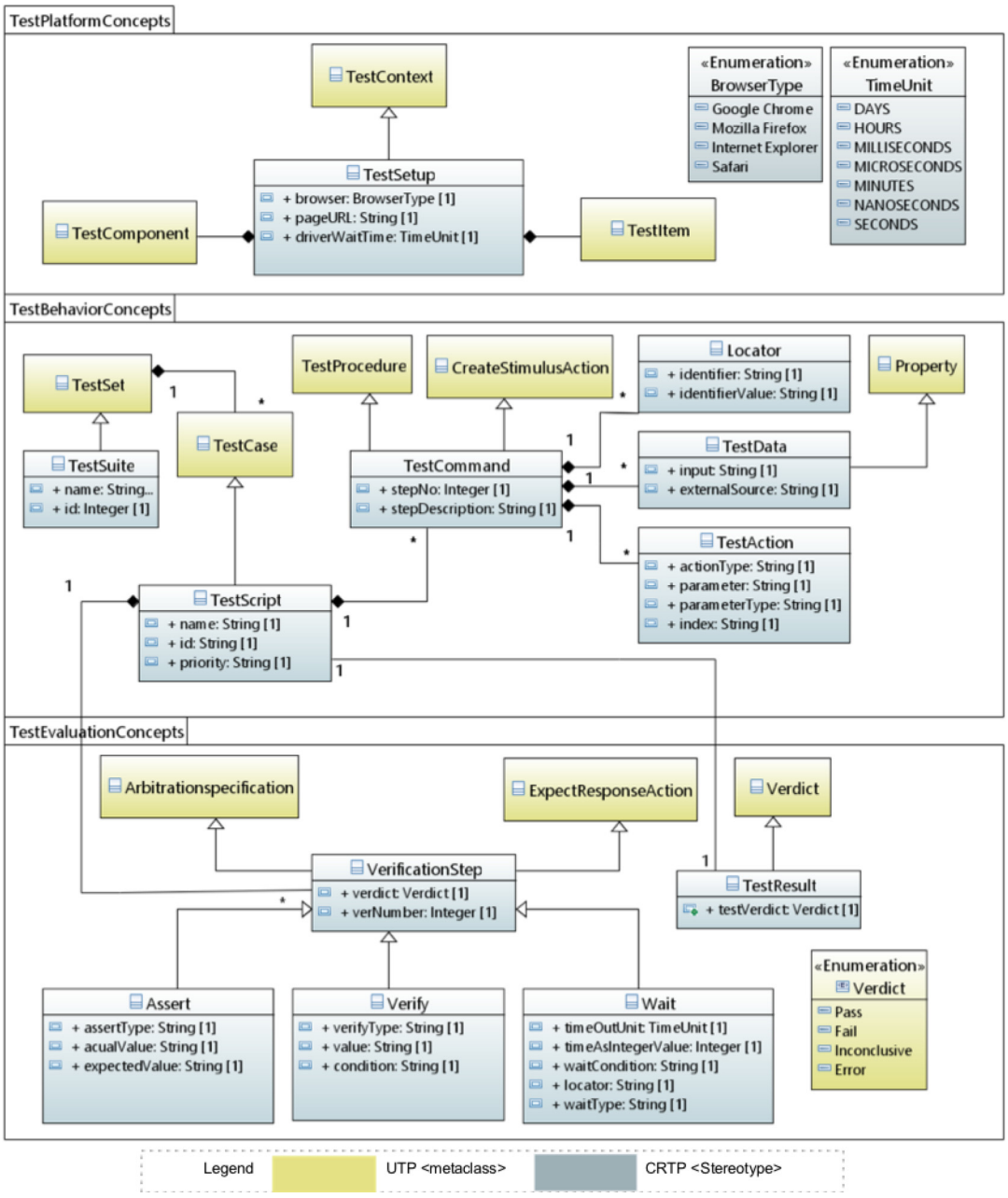

Fig. 9. Excerpt of Conceptual Model of Capture–Replay Test Profile.

Table 1
Stereotypes of UML capture–replay test profile.

Stereotype	Definition
TestSetup	It includes information about the reliable environment as well as the test itself.
TestCommand	It refers to a test command composed of <locator, action, value>.
TestAction	Type of event/action performed on the specific web element.
Locator	Key–value pair which is used to access/locate the specific element on web GUI.
TestData	It refers to the input text provided by the user through specific input control.
Assert	Confirmation of expected results (whether the given input is present or not on the web page).
Verify	Verify also ensure confirmation by continue execution and log the failure.
Wait	Wait for a specified condition to become true before proceeding to the next command
TestResult	It shows the verdict of the test script.

of TestSetup, TestCommand, and VerificationStep. These details are presented as attributes in the compartment view of activity test actions. For instance, the action "Enter Password" of activity diagram encapsulate information about the TestCommand which is capable of storing information about locators in the form of key–value pair <identifier:id, identifierValue:pass> ,TestData as "input=12345", and TestAction as "actionType= sendKeys". Upon the test execution, this information is used to parameterize the test execution environment.

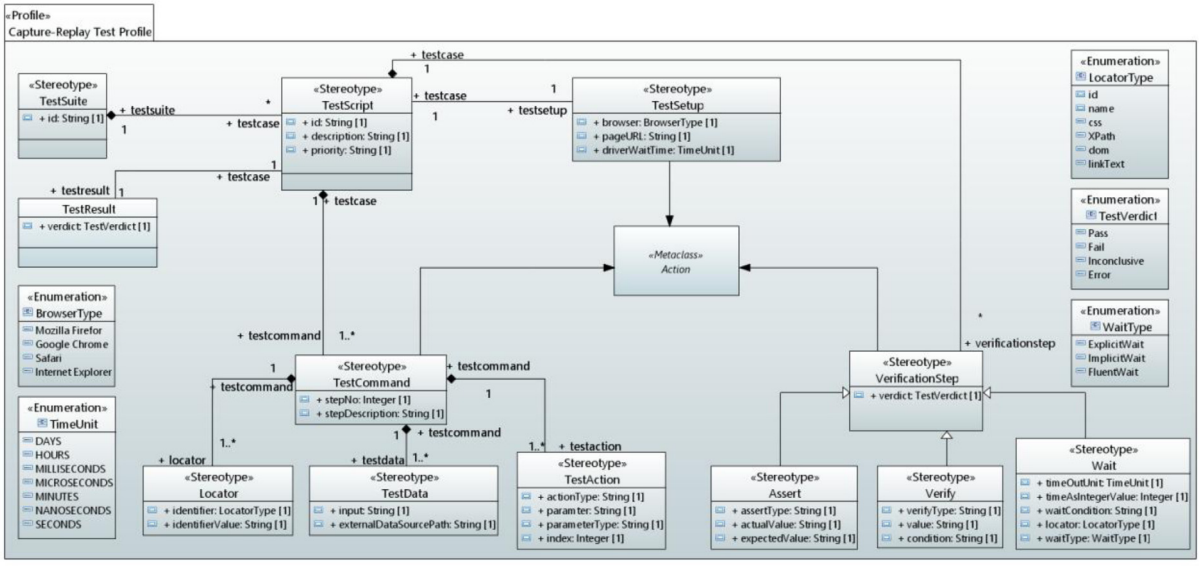

Fig. 10. Excerpt of UML Capture–Replay Test Profile.

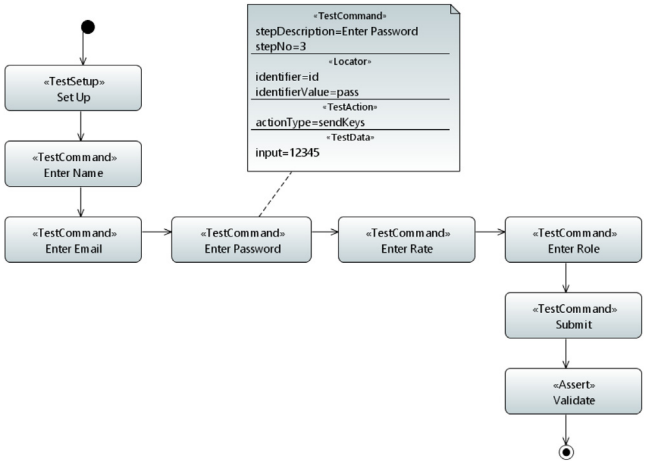

Fig. 11. An instance of a UML Capture–Replay Test Profile.

Apart from regression testing, the CTRP models may also be useful in a number of other situations. For example, the models can be used to migrate test scripts written in one automated testing framework into other test frameworks (write once, run anywhere). The models also allow running test scripts on different platforms (e.g., Windows, MAC) and different browsers (such as Firefox, Google Chrome).

3.3. Test suite classification

The first step of our approach (discussed in Section 3.1) provides us with a DOM Difference Model (DDM) containing the differences of web elements between the original and evolved web pages. The second phase of the approach (discussed in Sections 3.2 and 3.2.1) converts the tool-specific test scripts to instances of our CRTP profile to provide a tool-independent representation of the capture–replay test scripts. An important next step towards the repairing of test scripts is the classification of the original regression test suite as *Obsolete*, *Reusable*, and *Retestable* as defined in the literature (Leung and White, 1989). For this purpose, we proposed a test classification algorithm that takes the DOM difference model and a set of activity diagrams representing the baseline test scripts as input. The algorithm maps the difference model on various test commands for each test script to identify the test scripts that might be affected due to the changes in web pages. We use the following heuristics to identify the reusable, retestable, and obsolete test scripts for the evolved version of the web application.

1. The *reusable suite* contains test scripts for which there is no change corresponding in the delta version, therefore these test scripts can be executed directly. For example, a locator name=username in a test script that could select a DOM node is still able to select the corresponding DOM node in the newer version.
2. The *retestable suite* consists of test scripts for which there are changes in the evolved version of webpages such that the various test commands are affected. Affected test commands include at least one widget which is present as a changed element in the DOM difference model. For example, if a widget is deleted or modified on a web page, then all the test scripts that traverse the widget are retestable. The retestable test scripts are further divided into classes: *Fully Automated* (the retestable test scripts which are repaired automatically by applying repair transformations), *Human Assisted* (the retestable test scripts which required manual intervention for fixing the broken commands).
3. The *obsolete suite* is no more valid for the evolved version. In case of a web application, test scripts which correspond to deleted web element or web page are considered as obsolete test scripts. If a test script contains a single test command which refers to the deleted element, then we marked it an obsolete test script.

Fig. 12 shows the classification of the original regression test suite. We extracted all three classes of test scripts by using a difference model that contains the delta of original and modified versions of the same web application. Our algorithm takes two inputs: test suite TS of original web application and DDM. It provides Reusable T_{RE}, Retestable T_{RT}, and Obsolete T_{OB}test scripts for the evolved version of the web application as an output. For each test script TC $\in$ TS, it parses the test commands tci $\in$ TC which represents the action sequence performed on the web application. If the test command traverses the deleted web page, our algorithm classifies it TC $\in T_{OB}$. A test script is classified as

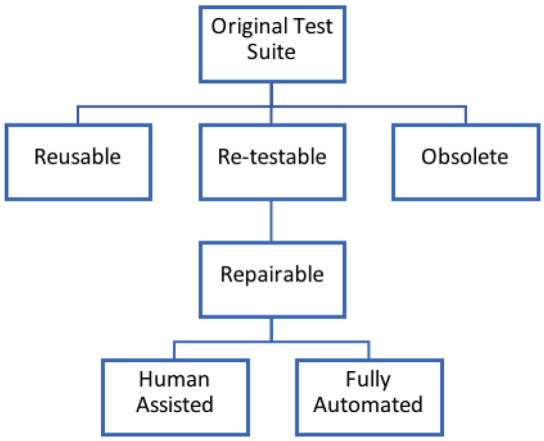

Fig. 12. Test Suite Classification of the original regression test suite.

T_{RT}if it traverses the changed web element. Every retestable test script is repairable if its initial test actions are valid on the evolved GUI. The remaining test scripts which are not marked as obsolete or retestable test scripts are classified as TC $\in T_{RE}$.

3.4. Test repair strategy

Once the classification of test scripts is completed, the next step is to repair the retestable test scripts. For this purpose, we developed repair strategies for various types of breakages that are discussed in this section. The existing DOM-based web test repair algorithms (Choudhary et al., 2011; Hammoudi et al., 2016a) suggest repairs/fixes for a limited set of test breakages from the possible set of breakages defined in the web breakages taxonomy (Hammoudi et al., 2016b). Other visual analysis-based repair algorithms suggest fixes for the broken Selenium tests by changing the incorrect locator using visual analysis. This only covers the category of locator-based breakages from the five categories discussed earlier (Section 2). Our work provides repair strategies for all the different categories of test breakages as reported in the existing taxonomy (Hammoudi et al., 2016b). Following are the various repair strategies provided by our approach.

A. *Replace Locator Value (RLV)*
This strategy applies when the unique identifier of a web element is modified. In a capture/replay testing, each element of a web page is referenced in a test script through locators (i.e. unique identifier). In the running example, we notice test breakage when a unique identifier ("id=pass") becomes ("id=password"), rendering the test unable to locate the web element. Our strategy automatically replaces the broken locator value with the modified value or uses XPath of the element using change information collected in a DOM difference model.

B. *Replace Assertion Value (RAV)*
This strategy applies when the textual content of a web element is modified. When the test script fails to locate the presence of certain text on the evolved web GUI (e.g. unexpected assertion value), resulting in a test failure. To solve this issue, our strategy verifies the validity of change that is not due to an error or bug and then provides a list of suggestions. This repair strategy requires human involvement to define the functional oracle of the tests.

   i. Get the updated value of web element from difference model and update the obvious assertion value e.g. label, title, etc.

```
Assert.assertEquals(‘‘Add updated Text’’,
driver.getTitle());
```

   ii. Use case-insensitive contains to match the part of label which remains same

```
Assert.assertTrue(driver.findElement
(By.xpath(‘‘//*[contains(text(),
‘someText’)]’’)));
```

   iii. Replaces a failing assertion with a similar assertion method that passes

```
assertTrue (condition) → assertFalse
(condition)
assertFalse (condition) → assertTrue
(condition)
assertEquals(Actualtext,ExpectedText) →
assertNotEquals(Actualtext,ExpectedText)
```

C. *Remove Test Command (RTC)*
This strategy applies when the web element is removed from a page. If a web element is removed from a page, all the test scripts that reference the deleted element stops working. Our strategy is to delete such test command from all the test scripts that reference the removed element.

D. *Update Element Type (UET)*
This strategy applies when the type of a web element is modified. If an element is removed, but a new element with the same label but different type exists, our strategy updates the specified interaction by using the following heuristics:

   i. When radio buttons become a dropdown list, use the following XPath-based format to correctly locate and select the matching option in the evolved version.

```
xpath(‘‘//select[@name=’ButtonName’]/
option[@value=’Buttonvalue’]’’);
```

   ii. When the dropdown list becomes radio buttons, use the following format to correctly click the corresponding radio button.

```
xpath(‘‘//input[@name=’ButtonName’]/
option[@value=’Buttonvalue’]’’);
```

To transform the radio button selection to a dropdown list, we use the XPath-based solution to map the corresponding selection. Our implementation extracts the value and name of the radio button using its locator from the earlier version to pick the same option from the updated dropdown list. We resolve the element type breakage issue of running an example, where the "Role" radio button becomes a dropdown list in the evolved version. The extracted value="3" and name="role" corresponding to the id="role3" is used to fix the web element type breakage. Our solution places such values into the following XPath-based dropdown option selection to successfully locate the matching element in the evolved version.

```
xpath("//select[@name='ButtonName']/option[@value='Buttonvalue']")
                    ↓                          ↓
          xpath("//select[@name='role']/option[@value='3']")
```

E. *Add New Test Command (ANTC)*
This strategy applies when the new web element is introduced on a web page. Mostly the addition of simple web elements may not have an impact on the existing test script. In case of a mandatory/required field in the evolved version, our strategy notifies the tester about mandatory/required fields.

i. Create a new test command corresponding to the added field.
`driver.findElement(By.id(‘‘indentifier Value’’)).sendKeys(‘‘value’’);`
ii. Ask the test value from the tester directly or automatically select the values from the user-provided data dictionary.

F. *Reload Web Page (RWP)*
This strategy applies when the page reloads are needed to ensure the presence of all web elements on the page before test execution. Our strategy suggests the addition of following commands in the test scripts according to the testing framework that navigates through the reloaded page.

i. `driver.findElement(By.id (‘‘placeholder’’)).sendKeys(Keys.F5);`
ii. `driver.navigaterefresh();`

G. *Alter Option in Dropdown (AOD)*
This strategy applies when the modification of choices in a dropdown list can cause index-based locators to break (e.g., they become unable to select the desired element on the web page). This case is about the deletion of certain options from the dropdown list. Our strategy recommends the selection of an option through its text rather than using its index value. The index-based selection of an option has more chances of breakages. Our strategy provides the following suggestions:

i. Update the index of an option in the test command `dropdown.selectByIndex(2);`
ii. Use “selectByVisibleText” to select required option `dropdown.selectByVisibleText(‘‘text’’);`

H. *Update Dynamic IDs (UDI)*
This strategy applies when the dynamic IDs of web element under test varies. Dynamic IDs often leads to problems in test automation because they are different every time an element is displayed. For example, in the first run, the test script is executed with ID: ’ext-gen123’ but when the same test script executed again to locate such element, it may return NoSuchElementException because the very same element would have the updated ID: ’ext-gen124’. Our strategy solves this problem by providing the following suggestions:

i. Select the locator which is not changing frequently like name, class, etc.
ii. If the name or class attributes are also evolving, we can use absolute XPath where attribute is not required.
iii. If some part of ID remains the same every time e.g. ’ext-gen123’, we can use the “starts-with” or “contains” function to repair the broken command.
`//input[starts-with(@id, element_name_ starting_word)`
`//input[contains(@id, element_name_ starting_word)`

I. *Update Widget Position (UWP)*
This strategy applies when the arrangement of elements on the web page changes. When the position of web element changes on the screen, then the test scripts may fail to locate such elements via structure-based locators (i.e., XPath or CSS). Our strategy solves this problem by using the XPath of an element in the evolved page.

The key challenge faced in repairing broken test scripts is to retain their purpose as much as possible. The typical action performed to “repair” a failing test script is by removing broken test commands so that it passes on the evolved version but it reveals nothing about the correctness of the application under test. Our approach makes changes in the test code and allows a tester to inspect and approve the suggested repairs. Table 2 shows the type of repair actions performed on the test models.

Once the test scripts are classified (discussed in Section 3.3), the retestable test scripts are fixed based on our proposed repair strategies. Fig. 13 shows the repair algorithm to fix the test breakages at the model level. This algorithm takes difference model *DDM* and test suite of retestable activity test models T_{RT} as an input and provides repaired test scripts T_{RP} as an output. This algorithm takes test scripts *TC* one by one and for each test command *Ta* in an activity test model, the algorithm checks its type. If the test command is of type TestSetup, then check for the corresponding web page in the difference model. Line 5 checks for the modification in the existing web page URL, get the updated URL from the *DDM* and update the test action at Line 6. Our algorithm also accounts for the page reloads at Line 8 because in some cases it is necessary to refresh the web page to ensure that all the elements are loaded before the test execution. There are different ways and commands to refresh the web page in test scripts according to CP tools. In the case of Selenium test scripts, add a statement “driver.navigate.refresh()” before the actual test commands. At Line 9, the algorithm checks the breakages at the test command level. After parsing the web element used in the test command Ta, then retrieve all changes related to that element through *DDM*. If the change is at attribute level that is a unique identifier (locator) of a web element modified, our algorithm updates the identifier of test command at Line 13 by replacing the old with the new value of an attribute from *DDM* or replaces the with the XPath of changed elements if the identifier is removed in the evolved version.

If the corresponding element is present as a deleted element in *DDM*. This algorithm removes the broken command *Ta* from the *TC* at Line 16. For the element type changes, our algorithm updates the specified interaction at Line 18. If an option is deleted from the dropdown list and the test script is unable to click on the option due to an updated sequence, our algorithm updates the index value in a test action at Line 21. The algorithm also checks the breakages at the Assert level because the modification of element text can affect the assertions in web tests. In case of such breakage at line 22, the expectedValue of *Ta* is updated with the new value of the modified element. At the end algorithm return repaired test models for testing the evolved version of web applications. Our algorithm can provide multiple repairs for more than one breakage in a test script.

3.5. Transforming test models to Capture–Replay tool-specific test scripts

After applying repair strategies on the instance models of the CRTP profile, updated models with valid action sequences are generated for testing the modified version of the application. In this step, platform-specific test scripts (i.e., capture–replay tool-specific test scripts) are generated from the updated test models by applying model-to-text transformations. For transforming the UML activity diagram to their equivalent code, we use the EMF tool to programmatically generate the test code for the specific testing framework. EMF provides an integrated environment for model development, model transformation, and Java code generation, for applications that are based on a structured model. To achieve this step, we implemented a component called ModelReader, whose task is to read the updated CRTP test models

Table 2
Repair actions.

Repair actions	Description
Add	Add test commands in the test model.
Delete	Delete test commands in the test model.
Update	Update test commands by changing the Locator(identifier), Locator(identifierValue) and TestData(input) etc.

Algorithm-2 Repair Algorithm

```
Input:  DDM: Difference Model
        T_RT: Activity models of retestable test cases
        TC:  Retestable test case
Output: T_RP: Repair Test case
Begin: Repair Algorithm
1.  foreach TC in T_RT do
2.    foreach Ta in Tc do
3.      if Ta.type==TestSetup
          //check the corresponding page in Difference model
4.        page=  findElement(DDM, Ta.pageURL)
5.        if (change== "changedPage") then
6.          Ta.PageUrl= page.URL
7.        if (page -> location.reload(true )) then
8.          Tc.addStatement("driver.navigate().refresh()" )
9.      if  Ta.type==TestCommand
10.       element= findElement(DDM, Ta.identifier)
11.       cc[]= element.getChanges()
12.         if (cc== "attribute-based change") then
13.           Ta.identifier = element.attribute.newValue;  //update statement
14.           Ta.identifier = element.attribute.XPath;
15.         else if (cc== "DeletedElement") then
16.           Del[]=  Tc.removeAction(Ta)  //remove statement
17.         else if (deletedElement.name == addedElement.name)
18.           Ta.indentifier= addedElement.id     //type change
19.           Ta.indentifier= addedElement.XPath
20.         else if (cc == "option-based change") then
21.           Ta.TestAction.index= elemnt.option.positionIndex //update index
22.           Ta.TestAction.index= elemnt.option.XPath
23.     if Ta.type==Assert
24.       ele= findElement(Ta.indentifier)
25.       if (text-level change) then
26.         newValue =ele.getUpdatedText()
27.         Ta.expectedValue= newValue
28.   endfor
29. endfor
30. return T_RP
end
```

Fig. 13. Test Model Repair Algorithm.

and transform the updated actions of the activity diagram into the actual test steps. For example, the updated action "Enter Password" of the activity test model holds the updated value corresponding to the repair strategy (RLV).

This repaired action is transformed into Selenium test step by utilizing its concrete syntax which is "*driver.findElement(By.id ("password")).sendKeys("12345");*". This Selenium test step uses the values encoded in the test action of test models in the form of Locator <identifier:id, identifierValue:password>, TestData as "input=12345", and TestAction as "actionType= sendKeys". Additionally, this test model has an assertion to verify the expected behavior. An assertion can be of different types in different testing frameworks which are specified by its attribute type, e.g. the Selenium testing framework has assertEquals, assertNotEquals, assertTrue, assertFalse, assertNull, assertNotNull assertions. By reading the assertType encoded in the Assert test action, an assert statement containing actual and expected values is formed. In QTP test scripts, Checkpoints are used as a verification point that compares the actual and expected values for a given test object. There are many types of checkpoints use to verify the text, images, content of the table, etc. Figs. 14 and 15 show the snippet of repaired Selenium and QTP test scripts "testAddNewUser" for version 1.0.0 generated from one test CRTP test model.

4. Tool implementation

To automate the model-based web test repair process, we developed a tool[5] using Java language that supports Capture–Replay test scripts. The tool is released as an open-source tool. The current implementation supports the repair process of the most widely used web testing tool e.g. Selenium IDE, Selenium web driver, Katalon studio, Unified Functional Testing (QTP), and Test Complete but can easily be extended to support other capture and replay testing tools. The tool consists of five major components, (i) Difference Detector (ii) Model Generator (iii) Test Classifier (iv) Repairer, and (v) Transformer. Fig. 16 shows the component diagram comprises of all major components of the model-based web test repair tool and the interaction among them. In the following, we discuss each component individually. The "DifferenceDetector" component takes old and new versions of the

[5] https://github.com/javariaimtiaz12/Model-basedWebTestRepairTool.

```
driver.findElement(By.id("name")).sendKeys("John");
driver.findElement(By.name("email")).sendKeys("john@pm.com");
driver.findElement(By.id("password")).sendKeys("12345");
driver.findElement(By.name("rate")).sendKeys("$500");
driver.findElement(By.xpath("//select[@name='role']/option[@value='1']")).click();
driver.findElement(By.id("company")).sendKeys("Company");
driver.findElement(By.xpath("//button[@type='submit']")).click();
assertEquals("Added Successfully", driver.findElement(By.id("userSystemMessage")).getText());
```

Fig. 14. A snippet of Repaired SeleniumTest Script for "Add User" web page of Collabtive 1.0.0.

```
Browser("Login @ Collabtive").Page("Project administration").WebEdit("name").Set "John"
Browser("Login @ Collabtive").Page("Project administration").WebEdit("email").Set "john@pm.com"
Browser("Login @ Collabtive").Page("Project administration").WebEdit("password").Set "12345"
Browser("Login @ Collabtive").Page("Project administration").WebEdit("rate").Set "$500"
Browser("Login @ Collabtive").Page("Project administration").WebList("role").Select "Project Manager"
Browser("Login @ Collabtive").Page("Project administration").WebEdit("company").Set "Company"
Browser("Login @ Collabtive").Page("Project administration").WebButton("Add").Click
Browser("Login @ Collabtive").Page("Project administration").Check CheckPoint("Added Successfully")
```

Fig. 15. A snippet of Repaired QTP Test Script for "Add User" web page of Collabtive 1.0.0.

same web application to computes the DOM-level differences. This component uses the Crawljax (to extract the HTML pages from both versions) and DiffUtils java library (to detect difference patches) as an external component to compute the differences between both applications. The output of this component is DDM containing the difference information which can affect the existing automated test scripts. The second step is to automatically transform the existing test suite into capture–replay tool independent representation by using our proposed CRTP models. For this purpose, we implemented a component "ModelGenerator", whose task is to load all the test scripts and automatically generate corresponding activity diagrams using the EMF tools. As our approach uses UML CRTP for modeling, therefore the profile reader module uses CRTP to load the instance of the profile. This step provides the instance of a capture–replay profile as UML activity test models. The third component "TestClassifier" uses the DDM and UML activity test models, generated by the model generator component, as input to classify the original regression test suite. The output of this component is "reusable", "retestable" and "obsolete" test models for the evolved version. The fourth component "Repairer" is responsible for automatically applying the repair strategies on the retestable test models based on the type of test breakage. This component uses DDM and proposed repair strategies to update the information capture in test models. The last module of our approach is the "Transformer" whose task is to transform the repaired instance of CRTP models (updated retestable test models) into testing dependent test scripts. This component uses EMF tools to generate the test script specific code corresponding to the updated test models.

5. Empirical evaluation

In this section, we conduct an empirical evaluation to evaluate the effectiveness of our proposed automated repair approach. Our goal is to assess whether the repaired test scripts are as effective as the original test suite on the evolved application. Several different approaches have been used by researchers for such evaluations: (i) by executing the repaired test scripts on the evolved applications without any breakages as done by Mirzaaghaei et al. (2012, 2010), Mirzaaghaei (2011), or (ii) by computing the coverage of test objects of the two versions of test suites as done by Li et al. (2017), Gao et al. (2015), Pinto et al. (2012), or (iii) asking a group of testers to manually evaluate the repaired test scripts, as done by Grechanik et al. (2009), Daniel et al. (2009, 2010) and (iv) by finding the fault detection capability of the repaired test scripts. We evaluate the effectiveness of our approach by all the above four mechanisms.

Additionally, we also compare our approach with the existing state-of-the-art web test repair approach. The existing approaches Vista (Stocco et al., 2018) and COLOR (Kirinuki et al., 2019) only repairs the locator-based breakages in the test scripts whereas WATER (Choudhary et al., 2011) supports the repair of test scripts corresponding to breakages due to locators and assertions. WATERFALL (Hammoudi et al., 2016a) is an extension of the WATER, however, it does not increase the support of test repair of breakages. The focus of WATERFALL is to repair breakages using WATER strategy between immediate commits of a web application. For WATER we can compare two different major versions of the approach, which is more suitable in our context. Hence, we compare our approach with WATER (Choudhary et al., 2011) because it is a DOM-based (similar to ours) and provides the largest number of repairs corresponding to the test breakages.

5.1. Research questions and metrics

To achieve our goal, we perform a series of experiments to (1) investigate the effectiveness of the proposed approach in terms of broken test scripts repaired and executed successfully on the evolved versions, (2) measure the overall DOM coverage achieved by the repaired test scripts on the evolved web applications with respect to the coverage reached with the original test scripts, (3) determine the usefulness of repaired test scripts according to the opinion of professional testers, (4) evaluate the fault-finding capability of the repaired test scripts, and (5) assess how effective the proposed approach is when compared to another state-of-the-art DOM-based approach. The experiments are performed to answer the following five research questions:

RQ1. *How many retestable test scripts can be repaired by the proposed approach?* The goal of this research question is to measure the number of broken test scripts that can be repaired successfully by our approach. We consider the test script to be successfully repaired if it does not contain any test breakage. The metric used to answer RQ1 is the *'number of test scripts repaired (repairability)'* by our approach either automatically by applying the repair strategies (automated fixes) or by involving testers to provide the test oracle (human-assisted fixes).

RQ2. *How does the DOM coverage of the repaired test scripts compare with the original test scripts?* This research question aims to evaluate the effectiveness of the proposed approach in retaining the coverage of DOM elements in the evolved version. The metric used to answer RQ2 is the *'DOM coverage'* achieved by executing repaired test scripts on the evolved versions to gain confidence that the same percentage of the DOM coverage is

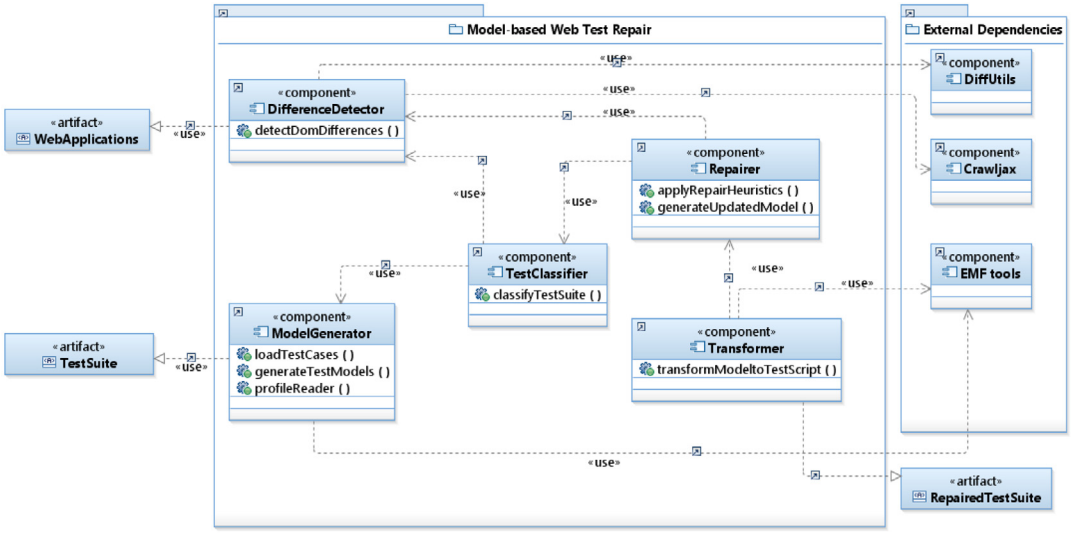

Fig. 16. Component Diagram of Model-based Web Test Repair Tool.

attained when compared to the execution of the original test scripts on the original version of the system. DOM coverage is a widely used metric for evaluating the strength of a test script.

RQ3. *Are the repairs suggested by our approach useful for testers?* This research question aims to evaluate the usefulness of the repaired test scripts according to the opinion of professional testers. The metric used to answer RQ3 is the *'test repair quality'* of the repaired test scripts. This metric reflects the validity of the test script repairs.

RQ4. *What is the fault detection capability of the repaired test scripts?* The research question aims to evaluate the DOM-based fault detection capability of repaired test scripts. The metric used to answer RQ4 is the *'number of faults detected'* by the repaired test scripts compared with the original test scripts.

RQ5. *How effective is the proposed approach at repairing test breakages compared to the existing approaches?* The goal of this research question is to compare the effectiveness of our test suite repair approach with the state-of-the-art approach WATER (Choudhary et al., 2011) at repairing test breakages. The metric used to answer RQ5 is the *'percentage of test scripts repaired'*.

The following subsections report the evaluations corresponding to the five research questions. We follow the guidelines presented by Wohlin et al. (2012) to present the experiment design and methodology (including subject applications, selection criteria), experiment results, and the discussion.

5.2. Experiment design & methodology

We conducted the five evaluations on seven different subject applications, including one industry application and six open source web applications. For the industrial case study, we used a web-based academic management system (Flex- Academic Portal https://flexstudent.nu.edu.pk/Login). This system is serving as an interface between students and institution management to enable students to promptly check their grades, as well as track their performance in every course. It allows users to effectively administer and update the student database. The system provides an efficient means to manage and organize student results in an error-free manner. It also enables prompt processing and releasing of relevant academic documents (e.g., transcripts, student attendance, grade sheets) in a real-time fashion. The two versions included for our evaluations are versions 1.2 and 1.5. The test scripts for the industrial case study are written in Selenium Web driver.

For open-source applications, we used the six case studies that are widely used in web testing research (Leotta et al., 2016), primarily in the studies evaluating test case repair strategies (Leotta et al., 2015, 2014, 2013). According to a recent literature review on test breakage prevention and repair techniques (Imtiaz et al., 2019), these six open-source web applications are the most commonly used subject applications in literature and therefore can be considered as a benchmark for empirical evaluation. Table 3 provides information about the selected web applications, including their names, versions, lines of code (LOC), and the number of test scripts. The *Cloc* tool (Cloc. https://github.com/AlDanial/cloc) is used to count the number of lines in each case study. In the following, we provide a brief description of these six case studies:

- *AddressBook* (AddressBook. https://sourceforge.net/projects/php-addressbook/) is a PHP based open source web application to manage phone contacts and groups. Two versions of the application used for evaluations are versions 4.0 and 8.2.5.
- *Claroline* (Claroline. https://sourceforge.net/projects/claroline/) is an open-source PHP based eLearning platform for a collaborative learning environment through the web. The two included versions are Claroline (versions 1.10.7, and 1.11.5).
- *Meeting Room Booking System (MRBS)* (Meeting Room Bookin System. https://sourceforge.net/projects/mrbs/) is a PHP based web application used by companies for booking meeting rooms and other resources such as computers, etc. The two included versions are versions 1.2.6.1 and 1.4.9.
- *Mantis Bug Tracker* (MantisBT. https://github.com/mantisbt/mantisbt) is a PHP based open-source web-based bug tracking system. The two included versions are versions 1.1.8 and 1.2.0.
- *Collabtive* (Collabtive. https://sourceforge.net/projects/collabtive/) is an open-source web application written in PHP. It provides a collaboration platform for the geographically scattered team members. We use the versions 0.65 and 1.0.0 for our evaluation.
- *PHP Password Manager (PPMA)* (P.P.M. http://sourceforge.net/projects/ppma/) is a web-based password manager. Each generated password is (DES-)encrypted with an individual user password. The two included versions are versions 0.2 and 0.3.1.

For all these six case studies, we selected two major release versions of the applications. These two versions are also used in prior studies of test case repair (Leotta et al., 2015, 2014, 2013). For the open-source applications, we used the set of test scripts that are available for the application versions on their websites and repositories. We have also made the test suites that are used for the experiments available online.[6]

For our experiments, we used Selenium (The Selenium Project. http://seleniumhq.org/docs/03_webdriver.html/) as the underlying Capture–Replay tool for the experiments as the most-widely used, freely available, Capture–Replay tool. The test scripts of the six case studies were migrated to their Selenium equivalent by the experienced test engineer. This activity is similar to the prior studies of automated web testing (Leotta et al., 2015; Biagiola et al., 2019) that also migrate the test scripts of these six case studies to Selenium. All changes made during migration were syntactic to make the scripts compatible with Selenium Web Driver API. By executing both the original and translated test scripts, we confirmed that both are semantically equivalent and reveal the same number of defects. We also manually checked the conformance of the translated and original test scripts to ensure equivalence.

For RQ1, the repairability of the original test scripts is measured for the six open-source and one industrial case study. The goal of this evaluation is to measure the effectiveness of our approach by checking how many test scripts are repaired without any remaining test breakage. We also discuss the reasons for any test breakages that cannot be repaired.

For RQ2, we compared the DOM-coverage obtained by the repaired test scripts on the evolved versions of the seven case studies with the DOM-coverage obtained by the original test scripts on the original versions of the seven case studies. The goal of this experiment is to measure how effective our approach is in retaining the coverage of DOM elements in the evolved version.

For RQ3, we evaluated the usefulness of test scripts repaired by our approach according to the opinion of professional testers. For this purpose, a team of experienced testers from a local organization was invited to review the repaired test scripts and to evaluate the usefulness of repairs provided by our approach. The team comprised of three experts with more than five years' experience of automated testing of web applications. The experts were provided with original versions of case studies, original versions of test scripts, and evolved versions of case studies along with evolved versions of the test scripts. The experts were asked to qualitatively evaluate the quality of the tests.

For RQ4, we used mutation analysis to compare the fault-finding capability of the repaired test scripts with the original test suite. Mutation analysis is a well-known method associated with real fault detection which allows measuring the quality of software tests (Just et al., 2014). By using this technique, faults were seeded in the seven case studies. We selected DOM level mutation operators for targeting our DOM level testing (Praphamontripong et al., 2016). A detailed description of each web mutant is presented in Table 4. The effectiveness is then measured based on the number of mutants killed by the original and repaired test scripts.

For RQ5, we compared the repairability of test scripts by our proposed approach on seven case studies with the repairability of test scripts by the WATER approach. The repairability was evaluated corresponding to a comprehensive test breakage taxonomy (Hammoudi et al., 2016b). Among the available tools and approaches in the literature, WATER provides the largest number of repairs corresponding to the test breakages. As the implementation of WATER was not available, similar to other studies in our area such as Stocco et al. (2018), we implemented the algorithm presented in the actual paper (Choudhary et al., 2011) to create a baseline for comparison with our approach.

5.3. Experiment execution

For the evaluations related to *RQ1*, *RQ2*, *RQ4*, and *RQ5*, we executed the experiments on a machine running Microsoft Windows 10 on a 3.19 GHz Intel Core i7 with 32 GB RAM and Chrome 77.0. We used the Selenium version (Selenium-java-2.25.0). A total of 528 Selenium web driver test scripts were used in the evaluation as shown in Table 3.

For *RQ1*, we applied our tool on the original Selenium test scripts of original versions of seven case studies to identify the reusable, retestable, and obsolete test scripts for the modified versions. We identify the retestable test scripts that are affected by the test breakages and can be repaired by applying repair heuristics.

For *RQ2*, we compare the DOM-coverage of the original test scripts with the repaired test scripts. We evaluate the effectiveness of our repaired test scripts on the evolved application, in terms of DOM objects (i.e., web elements) coverage. We computed the DOM element coverage before and after the repair process to measure the effectiveness of our test case repair technique whether it improves or decreases the coverage of evolved applications under test. For computing the DOM-coverage, we adapt the concept presented by Mirzaaghaei and Mesbah (2014), and collect the total count of DOM objects in the DOM tree for a web page and compare it with the DOM objects that are covered by the test scripts for that web page. The DOM Coverage of original test DOM-Coverage^{O} is compared with the DOM Coverage of the evolved test scripts DOM-Coverage^{E}.

$$\text{DOM Coverage} = \frac{\textit{Total Elements in DOM Tree of a Web Page}}{\textit{Total Elements covered by Test Scripts}}$$

For *RQ3*, we asked a team of testers to execute our tool for the evolving subject applications that caused the Capture–Replay test scripts to break. The testers were given the original test scripts and the original and evolved versions of each web application. The testers executed our web test repair tool for various broken test scripts and evaluated the usefulness of the test script repairs. These broken scripts were selected by the testers to cover the various types of test breakages. The testers classified the repairs in terms of *valid*, *partially valid,* and *invalid* repairs. After the study, we also conducted interviews with each of the testers to obtain their feedback on the tool. The overall duration of the whole experiment was five hours.

For *RQ4*, we seeded mutants corresponding to mutation operators of DOM and these are taken from Praphamontripong et al. (2016). A detailed description of each web mutant is presented in Table 4. The potential candidates for the mutation in the web pages are DOM elements which are accessed within a test script and have a direct impact to reveal the error. We collected the common web pages representing the similar functionality between both the versions of web applications. So, mutants were created in both versions of web applications to evaluate the fault detection capability of the original and repaired test scripts. Using the eight DOM-specific mutation operators, we generated a total of 604 mutants for all the seven subject applications.

For *RQ5*, the goal of this experiment is to compare the number of test breakages repaired by our tool with the state-of-the-art WATER tool. The total number of test breakages in the original and evolved versions of the seven applications were collected by executing the difference detection module (Section 3.1) on these versions and the original test scripts. The module provides a list of all test breakages in each of the applications. Both the WATER tool and our proposed tool was then executed on the seven case studies. The total number of repairs by each of the tools were then compared for each case study.

[6] https://github.com/javariaimtiaz12/Selenium-Test-Scripts-for-Web .

Table 3
Characteristics of subject case studies.

No.	Subject application	Description	Original version		Modified version		Test scripts
			LOC	Version	LOC	Version	
C1	AddressBook	Phone Book	10338	4.0	34548	8.2.5	55
C2	Claroline	E-Learning System	352565	1.10.7	333551	1.11.5	53
C3	MRBS	Meeting Room Booking System	11839	1.2.6.1	34144	1.4.9	35
C4	MantisBT	Bug Tracking System	108473	1.1.8	180741	1.2.0	49
C5	Collabtive	Collaboration Software	149887	0.65	220993	1.0.0	44
C6	PPMA	Password Manager	112010	0.3.1	575976	0.6.0	42
C7	Industrial Application	University Management System	743624	1.2.0	1055988	1.5.0	250

Table 4
Summary of DOM-based mutation operators.

Mutation operator	Type	Description
WLR	Simple link replacement	Replace the destination of link specified in href attribute of a <A> tag from the same domain of the targeted web application
WLD	Simple link deletion	Remove URL defined in href attribute of a <A> tag
WFR	Form link replacement	Change the destination of a form link with another link from the same domain of the targeted web applications
WTR	Transfer mode replacement	Replace all GET requests with POST requests and all POST requests with GET requests
WHR	Hidden form field replacement	Replace the value of a hidden field with another value of hidden control in the same application domain
WHD	Hidden form field deletion	Remove a hidden field
WIR	Server-side-include replacement	Replace the file attributes of include directives to another destination in the same domain of the targeted web application
WID	Server-side-include deletion	Remove an entire include directive from the HTML file

5.4. Results

In this section, we report the results of all five research questions to evaluate the effectiveness of our proposed approach.

5.4.1. RQ1. How many retestable test scripts can be repaired by the proposed approach?

In this *RQ*, we measured the effectiveness of our approach by determining the number of broken test scripts successfully repaired for the evolved versions. We calculated the retestable test scripts for all seven case studies. Table 5 shows the classification of the original Selenium test suites. We can see that a large number of test scripts from C1 (60% approx.), C2 (70% approx.), and C7 (78% approx.) are classified as retestable. We also noticed that a huge number of test scripts for C4 (MantisBT) are classified as obsolete due to the modifications in the evolved version. These modifications include the deletion of web pages (e.g., "My view" and "Edit News" etc. are removed from evolved version), or grouping of various main options (i.e., moving to Bootstrap template) and most importantly moving from a single page application to multi-page application. Such obsoleted test scripts cover the functionality which is removed from the evolved versions or contain the test sequences which are impossible to cover by the existing test scripts. We also notice that the majority of the retestable test scripts are affected by more than one test breakage. We detect that overall, 81/353 test scripts (23% approx.) exhibited at least one breakage, and 77% of retestable test scripts suffered from multiple breakages. Based on our analysis, we observe that, on average, between 1–4 breakages are present per test script.

The retestable test scripts were then repaired by applying repair strategies. A summary of the results is shown in Table 6. The column (Repaired Tests) shows the test scripts repaired by our approach, for all subject applications. Our approach has cumulatively repaired the (323/353) retestable test scripts for all subject applications. The results are quite encouraging for Claroline (C2), Collabtive (C5) and industrial application (C7) with (92% approx.), and (91% approx.) and (93% approx.) repaired test scripts, respectively.

Table 5
Classification of regression test suite.

Subject application	Reusable tests	Broken test scripts		Total
		Retestable	Obsolete	
C1	18	33	4	55
C2	9	37	7	53
C3	14	19	2	35
C4	12	23	14	49
C5	15	24	5	44
C6	13	23	6	42
C7	46	194	10	250
Total	**127**	**353**	**48**	**528**

Table 6 shows the overall repair results. The column (Total Fixes) highlights the number of automated repairs and the repairs in which the tester is asked to provide test data and updated oracle due to a change of functional requirements in the evolved version of the web application (*human-assisted fixes*). The detailed discussion on the types of test breakages and their corresponding fixes is presented while answering *RQ5* (Section 5.4.5). In our case, 78% of the broken capture–replay test scripts were repaired automatically without any human intervention, whereas 12% of the test scripts required human intervention for functional changes.

Most of the breakages were fixed automatically by utilizing the updated values of their matching elements in the evolved version (i.e., replace old values with the new values) or by applying customized repair strategies (Section 3.4). Such automatic fixes 1179/1386 (85%) facilitate breakages such as attribute-based locators, assertion content updates, element type changes, page reloading problems, etc. In some cases, the tester was required to select one option from the set of fixes automatically generated by the tool.

Similarly, in some of the breakages, the tester provided test data and updated oracle due to a change of functional requirements in the evolved web application. For example, in case of new mandatory fields in the evolved web page for which page expects data to submit a form request. Such changes affect the assert

commands which leads to an assertion failure. To solve this issue, our approach notifies the user and asks for the intended values to retain the originality of test scripts. Such human-assisted fixes 207/1386 (15%) facilitate breakages such as the addition of mandatory fields, password validation, and to confirm the test oracle in case of functionality changes.

To measure the usefulness of the repaired test scripts, we executed the test scripts repaired by our approach and checked how many test scripts execute correctly (i.e., do not fail after a repair) on the evolved versions. Table 6 shows that repaired test scripts (column Repaired Tests) execute correctly (column Executable Test Scripts). Only 22 repaired test scripts failed to successfully execute on the evolved version. After analyzing those failures cases, we identified that one of the main reasons for their failure to execute is timing errors. Test scripts that use assertion on dynamic content have to wait for content to load on the screen. Timing issues occur in Capture–Replay tools because automated test scripts follow a specific sequence to perform a set of actions on the web applications GUI. Any kind of delay in these actions or sequence changes produces invalid results. For example, in the case of C7 (developed using Ajax and JavaScript), we observe the use of dynamic elements or elements which are not always present in the DOM but the success of test scripts largely dependent on their presence. Such cases required the presence of elements on the webpage to interact.

We found the poor use of "Thread.sleep" methods in the C3, C4, C6, and C7 test scripts to wait for specified conditions. These sleep methods are static and pause the test script for the specified amount of time to meet a specific condition. These methods unnecessarily increase the test execution time and fail when the test environment varies. Our repair strategy to load all web page elements before test execution using browser refresh operations helped to solve many timing errors. Another reason observed for their non-executability is due to the non-interactable web elements which throw the "ElementNotInteractableException" exception. It means that the element is present in the DOM and correctly repaired by the test script but not interactable (i.e., not able to click, or unable to enter data into the web element). There could be many reasons for this problem such as the target element is not clickable because it is covered by another web element, or it requires a certain time to properly load on the page, etc. We find that timing issues related to dynamic web elements targeted by web test scripts require more attention.

Furthermore, we believe that it alone is not an adequate way to assess the quality of the repaired test scripts. For example, there may be empty test scripts that would also execute properly. To address this potential limitation, we conducted experiments 2 and 3 to identify those cases. The manual inspection and coverage-based experiment would help to identify such repairs if they are present in test scripts.

> *Our approach has effectively repaired the (323/353) retestable test scripts and 93% of the repaired test scripts were successfully executed on the modified versions of the subject applications.*

5.4.2. RQ2. How does the DOM coverage of the repaired test scripts compare with the original test scripts?

In this *RQ,* we compared the DOM-coverage obtained by the repaired test scripts on the evolved versions with the DOM-coverage obtained by the original test scripts on the original versions to measure how effective our approach is in retaining the coverage of DOM elements in the evolved version. For each subject application, Table 7 shows the target DOM objects, their coverage by original test suite (DOM-Coverage^{O}), and coverage observed by the repaired test suite (DOM-Coverage^{E}). Based on the results presented, we can see that our repaired test suite has maintained almost similar coverage to the coverage obtained on the original version by its corresponding test suites. Furthermore, there are other functional changes and new functionalities are introduced in updated versions of subject applications that require new test scripts and human assistance to fix them.

Interestingly, the coverage of repaired scripts is quite close to the coverage of original test suites on some applications like C1(81%), C3(84%), and C6 (81%). These changes observed in such subject applications are modifications in the existing web elements. By providing the correct fixes to repair the broken test scripts have retained almost the same coverage as by the original test suite. The percentage of repaired scripts is not as high for C4 (59%) neither through automatic fixes nor by involving tester's assistance. The DOM element coverage of repaired scripts for C4 is less than the original scripts. We found that many test scripts were removed from the original test suite because those scripts cover the obsolete functionality when executed on the evolved application. So, the removal of obsolete test scripts from the whole test suite resulted in reduced coverage. Upon further analysis, we identified that only 10% of the coverage is affected by the obsoleted test scripts. While in other cases, certain C4 test scripts depend on external resources such as input files (e.g. bug reports). Such failures cannot be repaired by our approach because it requires custom strategies tailored to the applications.

Table 7 shows the improvement in DOM coverage achieved by the repaired test scripts. The column (DOM-Coverage^{E}) shows the DOM coverage achieved by the automatic fixes and the one achieved after adding human assistance for the functional changes. We can see improvements in DOM coverage achieved after involving the tester's assistance in the overall repair process. In some test breakages, human assistance is required to provide application-specific fixes to retain the originality of test scripts. Tester provides test data and updated oracle due to a change of functional requirements in the evolved version of the web application. These cases include the addition of mandatory fields, password validations, and confirmation of function oracle of the test scripts. Such changes have motivated us to improve the repair process by leveraging tester's knowledge to reduce the ratio of false-positives. Almost 78% of the broken capture–replay test scripts were repaired automatically whereas 12% of the test breakages required human intervention for providing test oracles for the functional changes in the existing web elements.

An interesting case occurs in the case of industrial web application (C7) where the repaired test scripts resulted in high DOM-coverage by using the tester's assistance in providing test oracles. In C7, the obtained coverage on the evolved version is higher (79%) than the original (72%) due to the additional required fields. The major change observed in the industrial application is that validations are applied to the existing web elements. In the previous version, all the target web elements were optional but in the evolved version all become mandatory and expect values before form submission. To solve this breakage, our approach takes value from a user-defined data dictionary for the field values and in some cases involves testers to provide correct values for required web elements rather than writing whole new test suites. In this way, the repaired test scripts cover the more web elements than the original test suites and resulted in higher DOM element coverage. It shows that for functional changes, involving the testers into the repair process, increases the overall DOM coverage and also improve the fault detection effectiveness of the repaired test scripts as the original suite.

We also find some other cases from C2 and C3 where several test scripts were repaired and executed successfully but resulted in the lower DOM element coverage. The reason found that previous test scripts covered more DOM elements to achieve the same

Table 6
Repair results.

Subject application	Retestable tests	Total fixes		Repaired tests	Executable test scripts	Updated regression test suite
		Human assisted fixes	*Automatic fixes*			
C1	33	26	101	30	30	48
C2	37	19	165	34	33	42
C3	19	21	91	16	15	29
C4	23	34	98	19	17	29
C5	24	28	168	22	22	37
C6	23	25	114	21	20	33
C7	194	54	442	181	164	210
Total	**353**	**1386**		**323**	**301**	**428**

Table 7
Coverage results.

Subject applications	DOM objects	DOM-coverage^{O}	DOM-coverage^{E}	
			Automatic fixes	With human assisted fixes
C1	80	86%	79%	81%
C2	235	84%	78%	78%
C3	102	91%	82%	84%
C4	103	**76%**	**59%**	**59%**
C5	125	77%	73%	75%
C6	98	83%	78%	81%
C7	398	**72%**	**74%**	**79%**

functionality. For example, the number of test actions performed to reach the intended target element is less in the evolved version due to the reorganization of web elements (e.g., when elements start appearing on the main pages instead of menus). These type of changes has affected the DOM coverage in such subject applications. The overall coverage achieve by our web test repair approach is significantly higher than the coverage achieved by the un-repaired test suite on the evolved version.

> *Our approach is effective as it achieves almost the same DOM coverage as the original test scripts (obvious reduced coverage when the pages are not available in the evolved version) of the web application. In some cases, the DOM coverage is also improved.*

5.4.3. RQ3. Are the repairs suggested by our approach useful for testers?

RQ3 evaluates the usefulness of test scripts repaired by our approach according to the opinion of professional testers. The overall responses regarding test script repair of the tool were very positive. All of the test repairs of the selected test scripts, except two, were regarding as valid test repairs. The remaining two repairs were considered as *partially valid.*

The two test repairs that were considered *partially valid* were related to best practices of writing test scripts. These included the use of absolute XPath and the use of names as locators. The test scripts that were using an absolute XPath were repaired with a new absolute XPath referring to the web elements in the evolved web application (e.g., "/html/body/form/div/div/input[5]"). Similarly, the test scripts using the 'name' of a field as a locator were repaired by updating the locator with the modified name of the field in evolved applications. The testers thought that relative XPath should be used rather than absolute XPath and 'ID' of a field should be used as a locator rather than 'name' of a field. The focus of our approach is not to remove test smells or modify test scripts according to the best practices. The target of our approach is repairing the breakages that are caused by changes in web elements of web applications. Our approach currently repairs the test scripts using the same assumptions as that of the original test scripts. Automated repairing of test scripts for evolved web applications is a major challenge being faced by the industry, as mentioned by Chen et al. (2012) and agreed by the professional testers involved in the evaluation.

After the evaluation, an interview of these testers was conducted about their experiences of using the tool and the quality of suggested repairs. The testers found our tool effective in repairing the various test breakages of evolving web application. They found it beneficial in terms of repairing more than one test breakage in a single test script without changing the original test intent. They found it useful for the automated testing of their web-based software projects. Two of them recommended that this tool should be integrated with continuous integration and continuous deployment (CI/CD) tools to repair the broken test scripts fail due to modifications made after every build whereas one of them suggested that it should be the part of existing capture–replay tools to support regression activities. One of the testers also identified that the model-based part of our approach can also be used to assist organizations migrating from one Capture–Replay tool to another Capture–Replay tool. This is a common trend as a large number of small to medium-sized companies start by using open-source Selenium tools and as they grow they move to more sophisticated Capture–Replay tools, like QTP.

> *Our approach can suggest repairs for various types of test breakages and testers find the suggested repairs useful for the evolved web applications.*

5.4.4. RQ4. What is the fault detection capability of the repaired test scripts?

In this *RQ*, we used mutation analysis to compare the DOM-based fault detection capability of the repaired test scripts with the original test suite. Table 8 shows a summary of the mutants generated for seven subject applications. For each mutated version of the application, only a single line was mutated. For example, in the Addressbook application (C1), the WLR operator is applied to replace the destination of a simple link transition with the most frequently used URL. It has replaced the value of '*href*" attribute from groups to the most frequently used URL in the application groups. Such a mutation will cause the

test script to raise exceptions (i.e., element not found). The column (Total Mutants) shows the number of mutants generated for each subject applications. The mutant from each fault class is seeded into the web applications. Using the eight DOM-specific mutation operators, 604 mutated versions for all the seven subject applications.

After creating the mutants for all subject applications, the original and repaired test suites were executed on the mutated versions. While answering *RQ5* (Section 5.4.1), we evaluated the successful execution of both test suites on their original and modified web applications. The original and repaired test suites are then executed on their respective mutated versions and the difference between both their outputs was calculated.

Table 8 presents the statistics of mutation testing. The column ($MutantKilled^{O}$) shows the number of mutants killed by the original test suite whereas the column ($MutantKilled^{E}$) represents the mutants killed by the repaired test suite along with the mutation score for each subject application. The final mutation score for each test suite is calculated from the number of killed and live mutants. The information regarding the number of killed and live mutants by a test suite is gathered during the execution. If the result of the executing test suite on the original application and the mutated version is different, a mutant is considered as killed otherwise it is live. We can see that the original test suite has detected 547/604 mutants effectively with the mutation score of 90% which indicates the high quality of the designed test scripts. On the other hand, our repaired test suite has killed (526 out of 604) DOM-based mutants with a mutation score of 87%. Here is an important point to note that 480/528 original test scripts (without obsolete scripts) have detected 90% of the seeded mutants whereas our repaired test scripts (428) have killed 87% of the total seeded mutants which highlights the strength of our proposed approach. The 3% reduction in fault detection effectiveness is due to the obsolete test scripts and can be improved by adding new test cases for the new functionality. It shows that the fault detection capability of our repaired test suite is equivalent to the original suite. Upon further analysis, we found that 51 mutants were functionally equivalent to the original web pages, so they were excluded manually. All the equivalent mutants were of types WHR and WHD where replacing or removing the value attributes of hidden inputs to null, space, an empty string, and a negative integer did not impact the application's behavior.

> *Our repaired test suite has successfully detected the (526/604) DOM-based seeded mutants with the mutation score of 87% and achieved almost similar fault detection effectiveness as the original test suite.*

5.4.5. RQ5. How effective is the proposed approach at repairing test breakages compared to the existing approaches?

In this *RQ*, we compared the effectiveness of our test suite repair approach with the state-of-the-art approach WATER (Choudhary et al., 2011) at repairing test breakages. Table 9 lists the number of individual breakages observed per subject application according to each category of change mentioned in the taxonomy. The first column shows the five main categories of test breakages and the type of changes included in each category. Based on the collected breakage information, we can see that the most prevalent category concerns the locator-based changes (1065/1511) where attribute-based locators are the most brittle among other subclasses of test breakages. For example, we observed that breakages containing attribute-based locators resulted from simple name changes, i.e., changing an attribute from (name="submitButton") to (name="SubmitButton"). Such capitalization of the first letter to enhance the code readability effects the test quality by failing it. In three cases (i.e., Claroline, Collabtive, industrial application) out of seven, a large percentage of attribute-based locators become modified. Considering all seven applications, almost 717 over 1511 (i.e., 47% approx.) attribute-based locators are either renamed or deleted from the evolved version of web applications.

The second main category consists of value-based breakages which represent the changes like renaming labels, missing values, adding or removing web elements (text field or buttons), etc. By looking at the results from this category, we can see that "Missing Value" is evidenced as the prevalent class of change especially in the industrial application (67% approx.). We investigated the root cause of such breakage type and identified that it is resulted due to the web elements which were optional in the previous version but becomes mandatory/required in the evolved version. Across all categories of test breakages, overall 70.48% involved locators, 21% involved values and actions, 3.70% involved page reloading, 2.71% involved user sessions, and 2.11% involved popup boxes.

After collecting the test breakage information, we performed a comparison between our approach and WATER to evaluate their ability in generating repairs for different types of test breakages. We apply both tools on the same subject applications and available Selenium web driver test scripts. Table 10 shows the repairs generated by both approaches corresponding to the test breakages mentioned in Table 9. WATER act as a suggestive approach that offers the list of fixes for the test breakages to repair the broken commands. By looking at the results, we can say this WATER has provided fixes for only two categories of test breakages (i.e., locator-based changes and value-based changes) whereas our approach has provided the fixes for the changes from all categories.

The last column (Total Repairs) of Table 10 shows the repairs provided by both approaches. In our case, the generated repairs correspond to the modifications in existing DOM nodes (such as addition, deletion, or modification of web elements), web element attribute changes (e.g., modification in an identifier of a web element), label changes (e.g., the label of a widget is updated), and function-level changes (pop-up boxes). On the other hand, WATER has provided repairs for the changes such as attribute-based locators (modification in the attribute of web element) and value-based changes (broken labels and titles).

By looking at Table 10 we can say that WATER performed well for locator-based changes 786/1065 (74% approx.) but less than our approach due to structural changes occur in the DOM tree. Structural changes refer to changes such as modification (addition or deletion) or repositioning of web elements on the web page. Such hierarchal changes resulted in the wrong selection of DOM elements. Instead, our approach successfully handles the miss-selection problem by utilizing the XPath of matching widgets in the evolved pages. From this specific category, our approach provided almost 1002/1065 (94% approx.) fixes for almost all types of locator-based changes.

For the value-based breakage category, WATER has provided repairs for the updated content of DOM nodes (i.e., web elements) which may affect the assertions due to miss-match between the actual and expected values. For this specific category of change, WATER provided 189/317 (60% approx.) test repairs for the obsolete content problems. Our approach handles a wide range of value-based changes (89% approx.) including unexpected assertion values, invalid text fields inputs, missing values, altering the choices in a dropdown by providing a direct solution, or by involving humans to confirm the test oracle in case functionality changes. We observe that WATER can fix only simpler changes and it ignores other complex changes that occur frequently during evolution e.g. element type changes, the page reloads and dynamic locators, etc. Our approach also performed well on other

Table 8
Summary of generated mutants and repaired tests.

Web apps	Mutants HTML								Total mutants	Mutants killed^{O}	Mutants killed^{E}
	WLR	WLD	WFR	WTR	WHR	WHD	WIR	WID			
C1	24	13	7	3	9	4	1	4	65	60 (92%)	58 (89%)
C2	36	16	8	6	7	1	3	2	79	73 (92%)	70 (89%)
C3	33	18	11	9	5	6	3	1	86	80 (93%)	78 (86%)
C4	29	17	5	10	4	5	4	3	77	66 (85%)	63 (81%)
C5	37	12	6	14	5	9	7	5	95	89 (93%)	86 (90%)
C6	31	9	5	9	12	11	5	1	83	76 (91%)	72 (87%)
C7	44	21	14	13	9	7	5	6	119	103 (86%)	99 (83%)
Total	**234**	**106**	**56**	**64**	**51**	**43**	**28**	**22**	**604**	**547 (90%)**	**526 (87%)**

Table 9
Test breakages extracted from each subject application.

Categories of test breakage	C1	C2	C3	C4	C5	C6	C7	Total	%
Locator-based Changes									
Element attribute not found	62	142	58	70	120	76	189	717	47.45
Element text not found	16	17	12	20	15	15	20	115	7.61
Hierarchy-based locator target not found	8	15	10	5	17	12	65	132	8.73
Index-based locator target not found	9	12	15	8	16	11	30	101	6.68
Value-based Changes									
Invalid text field value input	6	5	4	7	5	5	20	52	3.44
Missing Value	6	16	2	12	4	4	80	124	8.20
Value deleted from drop down list	8	3	3	21	7	5	33	80	5.29
Updated text value	4	2	2	5	6	3	39	61	4.03
Page Reloads									
Page reload needed	2	1	3	1	5	5	15	32	2.11
Page reload no longer needed	3	2	1	1	4	2	11	24	1.58
Session-based Changes									
User session made longer	1	3	2	1	1	3	18	29	1.91
User session made shorter	2	2	2	1	1	2	2	12	0.79
Pop-up box related Changes									
Pop-up box added	2	3	2	5	1	2	0	15	0.99
Pop-up box deleted	3	2	3	3	5	1	0	17	1.12
Total	**132**	**225**	**119**	**160**	**207**	**146**	**522**	**1511**	**100**

Table 10
Distribution of test breakages across breakage classes.

Category	C1	C2	C3	C4	C5	C6	C7	Total breakages	Total repairs	
									WATER	Our approach
Locator-based changes	95	186	95	103	168	114	304	1065	786	1002
Values based changes	24	26	11	45	22	17	172	317	189	283
Page reloading	5	3	4	2	9	7	26	56	–	42
User session related	3	5	4	2	2	5	20	41	–	33
Pop up boxes related	5	5	5	8	6	3	0	32	–	26
Total	**132**	**225**	**119**	**160**	**207**	**146**	**522**	**1511**	**975**	**1386**

types of breakages categories such as page reloads 42/56 (75% approx.), user session-related breakages 33/41 (80% approx.) and pop up box related breakages 26/32 (81% approx.) due to having complete DOM difference information of both previous and modified versions and application of repair strategies.

A more detailed comparison of our approach with WATER is presented in Table 11. This shows the test repairs provided by both approaches for the changes of two major categories. From the locator-based test breakages category, WATER suggested repairs for the broken attribute-based locator 645/717 (90% approx.), and broken link text 98/115 (85% approx.) whereas, from the value-based category, WATER suggested fixes for missing value 84/124 (68% approx.) and 46/61 (75% approx.). WATER uses random values for empty required web elements to submit a form request successfully. These random values show the deviation from the expected behavior. On the other hand, our approach provides repairs for all types of changes from both categories. These results also show the number of correct fixes generated by both approaches out of the total repairs generated (see Table 10).

For all subject applications, WATER has generated 873/1511 (58% approx.) correct repairs from the overall test breakages whereas our approach has effectively repaired 1260/1511 (83% approx.) of breakages for the given subject applications. Both approaches have generated some false-positive (i.e., wrong element selection) repairs for the test breakages. We investigated the main reasons behind such behavior is the inherent nature of our subject applications and tests. Both approaches somehow suffer from the same problem of non-determinism of web applications. When the web element moves unpredictably on the web page, for example, when the page starts displaying random news headlines, notifications, and vacancies, etc. Due to such changes, the position of web element gets changed in the DOM and resulted in either test failure (“ElementNotFound”) or submit values to the wrong web elements.

WATER repairs 58% of overall test breakages including attribute-based locators and broken assertion values. Our approach, on the other hand, significantly repairs 83% of overall test breakages by considering the various types of test breakages.

Table 11
Comparison of WATER with proposed approach.

Fixes		Types of changes from Locator-based and value-based category	Correct repairs	
WATER	Our approach		WATER	Our approach
✓	✓	Element attribute not found	645	703
✓	✓	Element text not found	98	102
×	✓	Hierarchy-based locator target not found	0	107
×	✓	Index-based locator target not found	0	87
×	✓	Invalid text field value input	0	46
✓	✓	Missing Value	84	119
×	✓	Value deleted from the drop-down list	0	41
✓	✓	Unexpected assertion value	46	55
			873	**1260**

5.5. Discussion

In this section, we discuss the summary of each research question, lesson learned to reduce the occurrences of test breakages, tool development, and its usage.

Our empirical evaluation has confirmed that a huge percentage of automated test scripts (67%) break when the changes are introduced and therefore cannot be used for regression testing. It is prominent that test breakages (1511) largely affects the regression test suite. This shows the significance of repairing the test breakages to maintain the quality of the regression test suite. Our model-based approach significantly reduces the test repair effort by providing automatic fixes for the broken test scripts. We performed a set of five experiments to evaluate the effectiveness of our proposed approach through different measures (i) evaluation of test script repair, (ii) evaluation of DOM coverage, (iii) evaluation of test repair quality, (iv) fault-finding capability, and (v) comparison with WATER. The experiments were executed on an industrial and six open-source web applications with 528 Selenium web driver test scripts. The results show that the approach successfully repairs the broken test scripts while maintaining the same DOM coverage and fault-finding capability. Our empirical study has revealed that the proposed approach can successfully repair more test breakages, mentioned in the existing web test breakage taxonomy (Hammoudi et al., 2016b), than the WATER tool.

RQ1 Evaluation of Test Script Repair: In our first experiment, we evaluated the effectiveness of the proposed approach by determining the number of broken test scripts successfully repaired for the evolved versions. The result of the experiment confirmed that our approach successfully repairs the (323/353) retestable test scripts for all subject applications. By using our approach, almost 78% of the broken Capture–Replay test scripts were repaired automatically without any human intervention whereas 12% of the test breakages required human intervention for providing test oracles for the functional changes in the existing web elements. Through our approach, approximately 301/323 (93%) repaired test scripts executed successfully on their respective evolved versions without any failure. We also investigated the reasons for the non-executability of repaired test scripts. The timing issues, absence of dynamic web elements, and improper use of sleep, and wait methods were identified as the main reason for the non-executable repaired test scripts.

RQ2 Evaluation of DOM Coverage: In our second experiment we evaluated the effectiveness of the proposed approach in terms of DOM objects (i.e., web elements) coverage. We computed the DOM element coverage of the original and repaired test suite on their respective versions. Our repaired test suite has maintained almost similar coverage and in some cases more than the original test suite. A significant increase in coverage is observed by involving tester's assistance in providing test oracles for functional changes like the addition of mandatory fields, password validations, etc. The repaired scripts achieve over 80% of the coverage by the original scripts on the evolved versions except for those cases where the major functionality is removed from the evolved versions.

RQ3 Evaluation of Test Repair Quality: This experiment is performed to evaluate whether the test repairs suggested by our approach are useful for the testers. A team of experienced testers has reviewed the quality of repaired test scripts and shared their observations. They have randomly selected the broken test scripts to cover the various types of test breakages. After applying our web test repair tool on the broken scripts, testers found the suggested repairs useful for the regression testing of web applications. They have suggested the use of robust locators in the test scripts for reducing the chances of test breakages. In general, testers found this approach effective in repairing various types of test breakages without changing their original intent. Testers ensure that repaired test scripts are valid and maintain the same behavior as before evolution.

RQ4 Fault-finding Capability: In the fourth experiment, a fault-finding capability of the repaired test suite in comparison with the original test suite is evaluated through mutation analysis. By using this technique, faults are deliberately seeded into the web application by using well-known mutation operators. We apply the existing DOM-based mutation operators (Praphamontripong et al., 2016) on the subject applications. These are the common web application faults made by web developers. Examples include replacing the destination of links, deleting existing links, replacing form links, changing the transfer request methods, remove hidden fields, swapping the value of one hidden control with another from the same domain, etc. Results show that the original test suite has detected 547/604 mutants effectively with the mutation score of 90% whereas the repaired test suite was able to detect the (526/604) seeded DOM-based mutants with a mutation score of 87%. It shows that the fault detection effectiveness of the repaired test suite is equivalent to the original suite. We also found that 51 mutants were equivalent and are of types WHR and WHD where replacing or removing values of hidden web elements did not impact the application's behavior.

RQ5 Comparison with WATER: In our last experiment, we compared our proposed approach with the existing state-of-the-art DOM-based web test repair approach WATER. We considered WATER as a baseline because it repairs more than one type of test breakage. We executed our repair tool and WATER on the seven subject applications with 528 Selenium web driver test scripts. WATER has generated fixes for the two categories of test breakages such as locator-based changes and value-based changes (Section 2.1). It performed well for locator-based changes (74% approx.) but fail to generate fixes when structural changes occur in the DOM tree (i.e., repositioning of web elements on the web page). For the value-based breakage category, WATER only provided fixes for the updated content of broken assertions. Our evaluation revealed that the proposed approach can successfully repair a large number of test breakages (83%) correctly,

outperforming the existing WATER (58%). Our approach provides fixes for all five categories of test breakages mentioned in the existing taxonomy (Section 2.1) whereas the existing approach is limited to provide fixes for attribute-based locators and broken assertions.

Lesson learned: In the following, we report some lessons learned during the experimental evaluation through which we can reduce the chances of unexpected test breakages. We observed that many of the test breakages occurred due to bad testing practices. We listed some of them which should be taken care of by the testers and practitioners while designing their automated test suites to avoid unwanted test breakages.

- *Avoid the use of fixed waits/delay*: It is a common industrial practice to add fixed delays into the test scripts because the web page does not load all the elements instantly. For instance, click a button, wait for a specified amount of time (i.e., Thread.Sleep(2000) ;) and then verify the expected result. These fixed waits become problematic in situations like network latency or when the application under test takes unusual time to load content or waiting for some particular event to happen etc. Such fixed waits cause random test failures depending on how fast the response occurs. A better practice could be to add explicit wait for some expected condition to be met (for instance, waiting for a specific element to become present or visible) to ensure the synchronization.
- *Use robust locators for identifying the web elements*: Capture–replay tools generate complex locators to target the web elements which may lead to frequent test breakages. Such locators contain unnecessary information, for example, By.XPath(“//div[@class=”class-220″]/div[6]/h2”), that may change when the modifications take place in page structure. Such locators are brittle and less maintainable. It is very crucial to use robust locators to avoid brittle tests. The locators should precise and dependent on a single element rather than multiple to reduce the likelihood of change.
- *The ajax-based application requires extra consideration*: Ajax-based applications are dynamic and asynchronous. In these applications, we can never predict the exact time when the AJAX will appear on the screen. Thus, the automated test script waits for some elements to change instead of waiting for a new page to load. So, it fails to test when the web element does not appear in the specified time frame. The test scripts for such applications may face unexplainable nondeterministic failures like random timeout errors, or elements not found although visible, and difficult to manage and access ajax controls. These problems require some custom methods to handle such type of test scripts.
- *All the fixes should be approved by the tester*: The tester must verify the fixes suggested by the tool to ensure that it will not cause any deviation from the expected behavior that they are testing. Human involvement should be integrated into the repair process for the confirmation of suggested fixes by the repair tool, and providing customized fixes for application-specific breakages to retain the originality of the test scripts.

Web Test Repair Tool and Applications: We implemented our approach in a tool that can repair the automated web test script written in widely used capture–replay tools such as Selenium IDE, Selenium web driver, Katalon studio, Unified Functional Testing (QTP), etc. It allows the tester to compare the original and modified versions of web applications and provide suggestions for the various types of test breakages. This tool can repair different types of test breakages reported in existing test breakage taxonomy and is also extensible, where testers can also provide application-specific repair strategies in case of functional changes. The approach implemented in our web test repair tool is independent of any Capture–Replay tool. The model part of our approach makes it independent of testing frameworks and allow us to repair test scripts of various web testing tools. This technique can be useful in a large variety of regression testing activities, of which maintenance is the most important one. Apart from regression testing, our tool can be used to migrate test scripts written in one automated testing framework into other test frameworks. It also allows running test scripts on different platforms (e.g., Windows, MAC) and different browsers (such as Firefox, Google Chrome). The same approach can be used for automated testing of web application product lines. This tool can also use to assist the testers about the root cause of those test breakages which are not repairable due to having complete DOM information associated with test commands.

6. Threats to validity

Internal validity threats are associated with the uncontrolled factors which may affect the findings of the study. Our implementation for finding the DOM-level differences between subsequent HTML pages is based on the extension of the Java-DiffUtils (D.D.D.L. https://github.com/java-diff-utils/java-diff-utils) library. To reduce the chances of false-positives (i.e., irrelevant differences), we used XMLUnit (XMLUnit. https://github.com/xmlunit/xmlunit) to validate our results. This method compares and describes differences in semi-structured documents such as XML or DOMs. It provides a difference in the form of absolute XPath, where the difference is found, from both the documents. In our case, we require a more detailed comparison of the various HTML elements and are interested in all changes that may cause a test breakage. DiffUtils provide the difference patches which helps to achieve detailed differences. As our work is based on the existing test breakage taxonomy, we are not sure that we covered all breakages scenarios. Through our findings, we find another type of breakage which occurs due to the improper use of “sleep” and “wait” methods in the test scripts. This type of breakage may affect existing test scripts.

External validity threats are associated with the generalizability of our results. For empirical evaluation of our approach, we considered the seven subject applications from different domains, of different sizes and logical complexities to make the context realistic. Our criteria for selecting the subject applications, their versions, and test scripts ensure that we did not influence their codes or evolution. By sharing the experiment data (UML CRTP, model-based web test repair tool, Selenium Test Scripts), we have used in this study, we aim to motivate other researchers to contribute to this problem of web test repair for evolving applications.

7. Related work

There are four approaches in literature (Kirinuki et al., 2019; Stocco et al., 2018; Choudhary et al., 2011; Hammoudi et al., 2016a) that deal with automated repair of capture–replay test scripts for web applications. Work discussed in Kirinuki et al. (2019) utilizes DOM information to recommend the correct locator for the broken web element. This approach is designed to facilitate the locator-based breakages by using screen information such as attributes, text, images, and position in the DOM hierarchy. The approach is focused on fixing broken locators whereas our approach emphasizes various test breakages and generates the repaired test scripts.

Table 12
Summary of related work.

Ref	Approach type	Type of test scripts	Type of changes	Domain	Tool support
Kirinuki et al. (2019)	DOM-based	Selenium IDE	Locator breakages	Web Applications	No
Stocco et al. (2018)	Computer Vision-based	Selenium Webdriver	Locator breakages	Web Applications	Yes
Hammoudi et al. (2016a)	DOM-based	Selenium IDE	Locator breakages, Broken assertions	Web Applications	No
Choudhary et al. (2011)	DOM-based	Selenium IDE	Locator breakages, Broken assertions	Web Applications	No
Gao et al. (2015)	Model-based	QTP	Broken UI actions	Desktop Application GUI	No
Memon and Soffa (2003)	Model-based	GUI Test Scripts	Broken UI actions	Desktop Application GUI	No
Zhang et al. (2013)	Model-based	GUI Test Scripts	Broken UI actions/workflows	Desktop Application GUI	No
Xu et al. (2014)	Search-based	JUnit	Broken assertions values	Desktop Application	No
Pinto et al. (2012)	Heuristics-based	JUnit	Modified method names and broken assertions values	Desktop Applications	Yes
Daniel et al. (2011)	Symbolic execution-based	JUnit	Modified method names and broken assertions values	Desktop Applications	Yes
Mirzaaghaei and Pastore (2011)	Symbolic execution-based	JUnit	Compilation errors which lead to changes in the method declaration	Desktop Applications	Yes
Our	**Model-based**	**Test Script Independent**	**Web Test Breakages mentioned in change taxonomy**	**Web Applications**	**Yes**

Stocco et al. (2018) proposed a computer vision-based approach to repair the tests. The approach obtains visual information (i.e., image capture of web elements) during the initial test script execution. This information is later used to identify any change in a new version of the application. In case a change is identified, the corresponding test scripts are repaired. A major limitation of this approach is that it is highly sensitive to any visual change in the appearance of a web element. For example, in case the applied Cascading Style Sheet (CSS) is modified to change the color of a UI element, the approach will fail to detect the element and therefore will not repair the corresponding test scripts. Such visual changes are very common in web applications (Burg et al., 2015). Our proposed work is DOM-based and does not suffer from the same limitation.

Choudhary et al. (2011) present an approach (WATER) that utilizes the DOM information of two versions of a web application and suggests potential fixes for the broken elements to the tester. The work only covers the breakages due to locator-based changes and broken assertions. Our approach covers various changes e.g. web element type changes, repositioning of web element, addition, and deletion of dropdown list options, etc. which are very common in web applications (Imtiaz et al., 2019). Our proposed approach not only identifies the potential fixes but also repairs the existing breakages that are due to modification in previous web elements. Hammoudi et al. (2016a) present an extension of the WATER approach and focus on fine-grained changes in a web page rather than coarse-grained changes as in Choudhary et al. (2011). The approach shows better results than (Choudhary et al., 2011), however, it suffers from the same limitations as the WATER approach (discussed above).

Following we discuss the papers that are focusing on the repair of test scripts for desktop applications user interfaces (UI). Gao et al. (2015) presented a model-based approach, SITAR, to repair the obsolete UI test scripts. Memon and Soffa (2003) describes an approach to model UIs through event-flow graphs to identify the modification that occurs during software evolution. Zhang et al. (2013) presented a tool FlowFixer to suggest replacement actions for the broken sequence of UI actions. These approaches require manual conversion from test-cases to an intermediate model, are not targeting the web applications, limited to fixing the flow of test scripts. Some approaches in the literature discuss test script repair of unit test scripts writing in xUnit[7] tools. These include approaches by Xu et al. (2014), Pinto et al. (2012), Daniel et al. (2011), and Mirzaaghaei and Pastore (2011). These approaches do not apply to the repair of capture–replay test scripts of evolving web applications.

Table 12 presents a summary of recent works proposed in the area of test script repair. The first four studies target the testing of web applications. These studies are testing framework dependent (e.g. Selenium IDE, Selenium Webdriver) and only tend to repair a limited set of test breakage categories. Our web test repair approach uses a combination of both DOM and Model-based approaches in a single solution to provide fixes for the various categories of test breakages (Hammoudi et al., 2016b) specific to the web application domain. Furthermore, the model part of our approach makes our approach independent of testing frameworks.

8. Conclusion

Automated testing of web applications using capture and replay testing has recently received significant attention in the industry. Various tools support such testing, including the most widely used Selenium, TestComplete, and QTP. Developing and maintaining automated test scripts is a time-consuming activity. A major problem being faced in practice is the test breakages due to the evolving nature of web applications. In this paper, we discussed a model-based test repair approach for evolving web applications. Our model-based approach is independent of the underlying capture–replay tool. The approach targets the repair of test scripts that may break due to any of the breakage categories defined in the web test breakage taxonomy (Hammoudi et al., 2016b). It uses DOM-level change information to repair the various types of test breakages reported in the literature. To prove the generalizability of the proposed strategy it was evaluated on an industrial and six open-source case studies. We developed an open-source tool to demonstrate the applicability of the approach. The effectiveness of the proposed approach is evaluated through different measures: (i) evaluation of test script repair, (ii) evaluation of DOM coverage, (iii) evaluation of test repair quality, (iv) fault-finding capability, and (v) comparison with existing state-of-the-art DOM-based web test repair approach. Results indicate that the proposed approach effectively repairs the 91% of broken web test scripts and achieves almost similar DOM-coverage as the original test suite. Testers have found the suggested repairs useful for the regression testing of evolving web applications. The fault-finding capability of the repaired test suite is equivalent to the original test suite. Our empirical evaluation on 528 Selenium web driver test scripts of seven web applications shows that the proposed can effectively repair 83% of the overall test breakages, whereas the existing technique WATER repairs 58% test breakages which only includes attribute-based locators and broken assertion values.

[7] xUnit is a family of unit-testing frameworks used to write and run repeatable tests for software applications. It includes various testing frameworks such JUnit, SUnit and NUnit for different programming languages. .

CRediT authorship contribution statement

Javaria Imtiaz: Conceptualization, Methodology, Software, Writing - original draft, Validation. **Muhammad Zohaib Iqbal:** Writing - review & editing, Supervision, Project administration, Funding acquisition. **Muhammad Uzair khan:** Writing - review & editing, Validation, Formal analysis, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research is supported through a research grant titled 'Establishment of National Center of Robotics and Automation (NCRA)' by the Higher Education Commission (HEC) Pakistan.

References

Biagiola, M., et al., 2019. Web test dependency detection. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Budinsky, F., et al., 2004. Eclipse Modeling Framework: A Developer's Guide. Addison-Wesley Professional.

Burg, B., Ko, A.J., Ernst, M.D., 2015. Explaining visual changes in web interfaces. In: Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology. ACM.

Chen, J., et al., 2012. When a GUI regression test failed, what should be blamed? In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE.

Choudhary, S.R., et al., 2011. Water: Web application test repair. In: Proceedings of the First International Workshop on End-To-End Test Script Engineering. ACM.

Daniel, B., D.D., Gvero, T., Jagannath, V., 2011. ReAssert: a tool for repairing broken unit tests. In: 2011 33rd International Conference on Software Engineering. ICSE, pp. 1010–1012.

Daniel, B., Gvero, T., Marinov, D., 2010. On test repair using symbolic execution. In: Proceedings of the 19th international symposium on Software testing and analysis.

Daniel, B., et al., 2009. ReAssert: Suggesting repairs for broken unit tests. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE.

De Lucia, A., et al., 2007. Clustering algorithms and latent semantic indexing to identify similar pages in web applications. In: 2007 9th IEEE International Workshop on Web Site Evolution. IEEE.

Gao, Z., et al., 2015. Sitar: Gui test script repair. IEEE Trans. Softw. Eng. 42 (2), 170–186.

Grechanik, M., Xie, Q., Fu, C., 2009. Maintaining and evolving GUI-directed test scripts. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE.

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

Hammoudi, M., Rothermel, G., Tonella, P., 2016b. Why do record/replay tests of web applications break? In: 2016 IEEE International Conference on Software Testing, Verification, and Validation. ICST. IEEE.

Hartmann, M., et al., 2008. Using similarity measures for context-aware user interfaces. In: 2008 IEEE International Conference on Semantic Computing. IEEE.

Imtiaz, J., et al., 2019. A systematic literature review of test breakage prevention and repair techniques. Inf. Softw. Technol. 113, 1–19.

2019. Jsoup: java html parser.

Just, R., et al., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Kirinuki, H., Tanno, H., Natsukawa, K., 2019. COLOR: Correct locator recommender for broken test scripts using various clues in web application. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER. IEEE.

Leotta, M., et al., 2013. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: 2013 20th Working Conference on Reverse Engineering. WCRE. IEEE.

Leotta, M., et al., 2014. Visual vs. DOM-based web locators: An empirical study. In: International Conference on Web Engineering. Springer.

Leotta, M., et al., 2015. Using multi-locators to increase the robustness of web test cases. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation. ICST. IEEE.

Leotta, M., et al., 2016. Approaches and tools for automated end-to-end web testing. In: Advances in Computers. Elsevier, pp. 193–237.

Leung, H.K., White, L., 1989. Insights into regression testing (software testing). In: Proceedings. Conference on Software Maintenance-1989. IEEE.

Li, X., et al., 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In: 2017 IEEE International Conference on Software Testing, Verification and Validation. ICST. IEEE.

Lin, J.-W., Wang, F., Chu, P., 2017. Using semantic similarity in crawling-based web application testing. In: 2017 IEEE International Conference on Software Testing, Verification and Validation. ICST. IEEE.

Memon, A.M., Soffa, M.L., 2003. Regression testing of GUIs. ACM SIGSOFT Softw. Eng. Notes 28 (5), 118–127.

Mesbah, A., Van Deursen, A., Lenselink, S., 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web (TWEB) 6 (1), 3.

Mirzaaghaei, M., 2011. Automatic test suite evolution. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

Mirzaaghaei, M., Mesbah, A., 2014. DOM-based test adequacy criteria for web applications. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis.

Mirzaaghaei, M., Pastore, F., 2011. Testcareassistant: Automatic repair of test case compilation errors. In: Proceedings of 6th Italian Workshop on Eclipse Technologies. Citeseer.

Mirzaaghaei, M., Pastore, F., Pezze, M., 2010. Automatically repairing test cases for evolving method declarations. In: 2010 IEEE International Conference on Software Maintenance. IEEE.

Mirzaaghaei, M., Pastore, F., Pezze, M., 2012. Supporting test suite evolution through test case adaptation. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE.

Pinto, L.S., Sinha, S., Orso, A., 2012. Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.

Praphamontripong, U., et al., 2016. An experimental evaluation of web mutation operators. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops. ICSTW. IEEE.

Rau, A., Hotzkow, J., Zeller, A., 2018. Transferring tests across web applications. In: International Conference on Web Engineering. Springer.

Selic, B., 2007. A systematic approach to domain-specific language design using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. ISORC'07. IEEE.

Stocco, A., Yandrapally, R., Mesbah, A., 2018. Visual web test repair. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM.

UML Test Profile (UTP), 2005. Object management group (OMG): UML testing profile 2.0 beta and h.w.o.o.s.u.b.p. ptc/17-09-29.

Wohlin, C., et al., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

Xu, Y., B.H., Wu, G., Yuan, M., 2014. Using genetic algorithms to repair junit test cases. In: Asia-Pacific Software Engineering Conference, vol. 1, 2014, pp. 287–294.

Zhang, S., Lü, H., Ernst, M.D., 2013. Automatically repairing broken workflows for evolving GUI applications. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM.

Javaria Imtiaz is a Ph.D. scholar at Software Quality Engineering and Testing (QUEST) Laboratory, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. She received her MS(SE) degree in 2017. Her research interests include model-driven engineering, web test automation, and empirical software engineering.

Muhammad Zohaib Iqbal is currently an Associate Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He is also the chief scientist at Software Quality Engineering and Testing (QUEST) Laboratory and President of Pakistan Software Testing Board. He received his Ph.D. degree in software engineering from University of Oslo, Norway in 2012. Before joining Fast-NU, he was a research fellow at Simula Research Laboratory, Norway. His research interests include model-driven engineering, engineering mission & safety critical systems, software testing, and dependable avionics systems.

Muhammad Uzair Khan is currently an Assistant Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He is heading the Software Quality Engineering and Testing (QUEST) Laboratory, and is a founding member of Pakistan Software Testing Board. He completed his Ph.D. research work at INRIA, France and received his Ph.D. degree in computer science from University of Nice, France in 2011. His research interests include model-driven engineering, empirical software engineering, aspect-oriented software engineering, real time operating systems, and engineering dependable software systems.