Automated Generation of Visual Web Tests from DOM-based Web Tests

Maurizio Leotta, Andrea Stocco, Filippo Ricca, Paolo Tonella

Abstract:

Functional test automation is increasingly adopted by web applications developers. In particular, 2nd generation tools overcome the limitations of 1st generation tools, based on screen coordinates, by providing APIs for easy selection and interaction with Document Object Model (DOM) elements. On the other hand, a new, 3rd generation of web testing tools, based on visual image recognition, brings the promise of wider applicability and simplicity. In this paper, we consider the problem of the automated creation of 3rd generation visual web tests from 2nd generation test suites. This transformation affects mostly the way in which test cases locate web page elements to interact with or to assert the expected test case outcome.

Our tool PESTO determines automatically the screen position of a web element located in the DOM by a DOM-based test case. It then determines a rectangle image centred around the web element so as to ensure unique visual matching. Based on such automatically extracted images, the original, 2nd generation test suite is rewritten into a 3rd generation, visual test suite. Experimental results show that our approach is accurate, hence potentially saving substantial human effort in the creation of visual web tests from DOM-based ones.

Digital Object Identifier (DOI):

`http://dx.doi.org/10.1145/2695664.2695847`

Automated Generation of Visual Web Tests from DOM-based Web Tests

Maurizio Leotta[1], Andrea Stocco[1], Filippo Ricca[1], Paolo Tonella[2]

[1]DIBRIS, Università di Genova, Italy [2]Fondazione Bruno Kessler, Italy

{maurizio.leotta, andrea.stocco, filippo.ricca}@unige.it, tonella@fbk.eu

ABSTRACT

Functional test automation is increasingly adopted by web applications developers. In particular, 2nd generation tools overcome the limitations of 1st generation tools, based on screen coordinates, by providing APIs for easy selection and interaction with Document Object Model (DOM) elements. On the other hand, a new, 3rd generation of web testing tools, based on visual image recognition, brings the promise of wider applicability and simplicity. In this paper, we consider the problem of the automated creation of 3rd generation visual web tests from 2nd generation test suites. This transformation affects mostly the way in which test cases locate web page elements to interact with or to assert the expected test case outcome.

Our tool PESTO determines automatically the screen position of a web element located in the DOM by a DOM-based test case. It then determines a rectangle image centred around the web element so as to ensure unique visual matching. Based on such automatically extracted images, the original, 2nd generation test suite is rewritten into a 3rd generation, visual test suite. Experimental results show that our approach is accurate, hence potentially saving substantial human effort in the creation of visual web tests from DOM-based ones.

Categories and Subject Descriptors:
D.2.5 [**Software Engineering**]: Testing and Debugging–*Testing tools*

General Terms: Experimentation, Measurement

Keywords: Web Testing; DOM-based Testing; Visual Testing; Test Automation; Selenium WebDriver; Sikuli; Page Object; PESTO

1. INTRODUCTION

Web applications are developed and evolved at a very fast rate. Within such ultra-rapid development cycles, functional testing is an option only if it is strongly supported by automated tools, which can execute tests faster than a person can, and in an unattended mode, saving testing time and resources [8].

The 1st generation of web test automation tools was based on screen coordinates to locate the web elements to interact with, resulting in test suites that were very fragile when the application evolves. Minimal changes in the web page layout may in fact corrupt the existing test cases, requiring that developers redefine them from scratch. Second generation tools, also called DOM-based tools, access to the web page Document Object Model to overcome the limitations of the 1st generation. The web elements involved in the interactions performed by a test case are located by navigating through the DOM of the web page. This can be easily achieved via XPath queries or using unique anchors, such as web element identifiers or text. Selenium WebDriver[1] is an example of 2nd generation web testing tool. The test suites produced by these tools are more robust than those obtained from 1st generation tools [1], but their creation requires specialised skills, to properly navigate the DOM tree, as well as detailed, white-box knowledge of the web application. Third generation tools, such as Sikuli API[2] allow to define test cases visually, without requiring any white-box knowledge of the page structure. Web elements are identified by their visual representation using advanced image recognition algorithms, hence they do not suffer from the limitations of a pixel-based technique.

While DOM-based web testing is currently widely adopted in the industry [3], the new generation of visual tools [1, 2] offers an interesting alternative. This is particularly true when web applications evolve to modern technologies, e.g., Ajax or complex visual components (Google Docs or Maps), to offer increased user-friendliness and responsiveness. In these cases, DOM-based tools do not suffice because the DOM of these applications is complicated to retrieve. However, moving the existing DOM-based test suites to the 3rd generation may represent a substantial investment for companies; the task is boring, time-consuming and expensive.

In this paper, we propose a novel solution to *transform DOM-based into visual test cases, so as to allow developers to experiment with 3rd generation testing tools at minimal migration cost*. Our approach and its implementation, a prototype tool named PESTO, is able to transform a Selenium WebDriver test suite (2nd generation test suite) into a Sikuli API one (3rd generation test suite). Using PESTO, companies and professionals can evaluate the benefits of 3rd generation tools at minimum cost, without taking the risks of the manual migration. Automatically migrated test suites can be smoothly introduced in the existing testing process, so as to evaluate their effectiveness and robustness in comparison with the existing DOM-based suites.

Our work makes the following contributions:

– an approach for the automatic creation of visual locators starting from corresponding DOM-based locators. Our approach is based on the automated localisation of the web elements involved in any interaction or assertion performed by the original test suites. A

SAC'15, April 13 - 17, 2015, Salamanca, Spain
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00
http://dx.doi.org/10.1145/2695664.2695847

[1] http://seleniumhq.org/projects/webdriver/

[2] http://code.google.com/p/sikuli-api/

rectangle is automatically positioned and sized around these web elements, and an image is automatically captured for each web element surrounded by such rectangles;

– an approach for the automatic transformation of DOM-based test suites to the Visual technology. Based on the captured images, the original test suites are automatically rewritten as visual test suites;

– an implementation of our approach in an open source tool called PESTO;

– an empirical evaluation to assess PESTO, demonstrating its effectiveness. In our experiments on existing web applications, we evaluated the level of automation and correctness offered by PESTO. Our conclusion is that, despite the limitations associated with our current research prototype, PESTO's performance is encouraging and indicates that the approach is viable.

The paper is organised as follows: Section 2 provides some background on the selected 2nd and 3rd generation web testing tools. Section 3 describes our approach. Section 4 presents the experimental results. Section 5 describes the current limitations of PESTO. Section 6 discusses the related works, followed by conclusions and future work.

2. BACKGROUND

Automated functional web testing is based on the creation of test cases that automate the interaction with a web page and its elements. Test cases can, for instance, automatically fill-in and submit forms or click on hyperlinks. Test cases can be programmed using general purpose programming languages (such as Java and Ruby) and specific libraries that offer user friendly APIs, providing commands to, e.g., click a button, fill a field and submit a form. Finally, test cases are completed with assertions, e.g., using xUnit.

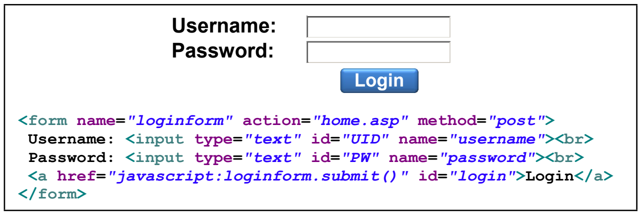
Nowadays, the most used web page elements localisation techniques are [11]:

- **DOM-based**: elements are located using the information contained in the Document Object Model (DOM). Tools belonging to this category (2nd generation) usually offer different localisation strategies (e.g., by ID, XPath and LinkText).
- **Visual**: tools belonging to 3rd generation use image recognition techniques to identify and control web elements.

Let us consider a running example, consisting of a web application including a login form shown in the upper right corner of the home page (home.asp). This web application requires the authentication of the users by entering their credentials, i.e., *username* and *password* (see Figure 1). If the credentials are correct, the username (e.g., John.Doe) and the logout button replace the login form in the upper right corner of the home page. Otherwise, the login form is still shown.

2.1 The Page Object and Factory Patterns

The *Page Object*[3] is a quite popular web test design pattern, which aims at improving the test case maintainability and reducing the duplication of code. A Page Object is a class that represents the web page elements as a series of objects and encapsulates the features of the web page in methods. All the functionalities to interact with or to make assertions about a web page are offered in a single place, the Page Object, and can be easily called and reused within any test case. Usually page objects are initialised by a Page Factory, a factory class that checks the correct mapping and initialisation of the web elements. The use of *Page Object* and *Page Factory* patterns reduces the coupling between web pages and test cases, promoting reusability, readability and maintainability of the test suites [9].



```
<form name="loginform" action="home.asp" method="post">
 Username: <input type="text" id="UID" name="username"><br>
 Password: <input type="text" id="PW" name="password"><br>
 <a href="javascript:loginform.submit()" id="login">Login</a>
</form>
```

Figure 1: home.asp – Page and Source of the Login form

2.2 DOM-based Web Testing

In this work, we consider Selenium WebDriver as a representative tool for implementing DOM-based web test suites (for short, WebDriver). WebDriver is a tool for automating web application testing that provides a comprehensive programming interface used to control the browser. WebDriver test cases are written in the Java programming language, by integrating WebDriver commands with JUnit or TestNG assertions. We chose WebDriver as the representative of this category, because: (1) it is a quite mature tool, (2) it is open-source, (3) it is one of the most widely-used open-source solutions for web test automation, (4) during a previous industrial collaboration, we gained a considerable experience in its usage [9].

In Figure 2 (top), we show an example of a simple WebDriver test case for our running example application corresponding to a successful authentication. This automated test case submits a valid login, using correct credentials (i.e., *username=John.Doe* and *password=123456*) and verifies that in the home page the user appears as correctly authenticated (the string "John.Doe" must be displayed in the top-right corner of the home page, i.e., it must be contained in the text of the HTML tag with ID="LoggedUser").

The first step for building this test case is creating the HomePage.java page object (see Figure 3), corresponding to the home.asp web page. The page object HomePage.java offers a method to log into the application. It takes in input username and password, inserts them in the corresponding input fields and clicks the Login button. Moreover, HomePage.java contains also a method that verifies the authenticated username in the application. As shown in Figure 3, web page elements are located using DOM-based locators (e.g., ID, LinkText, XPath).

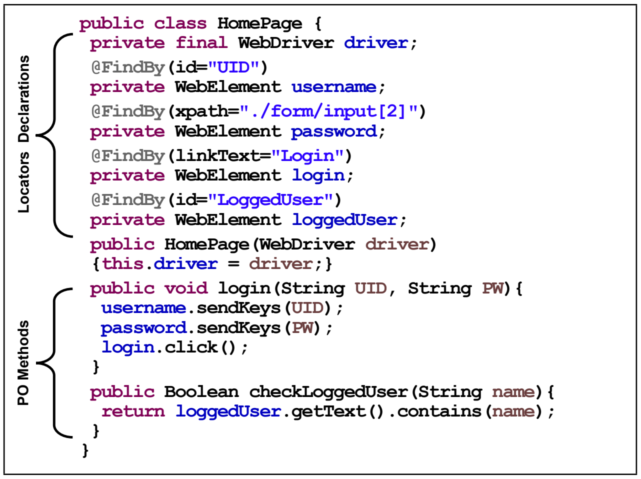
The second step requires to develop the test case making use of the page object methods (see Figure 2 (top)). In the test case, first, a WebDriver of type FirefoxDriver is created to control the Firefox browser as a real user does; second, WebDriver (i.e., the browser) opens the specified URL and creates a page object that instantiates HomePage.java; third, using method login(...) , the test tries to login in the application; finally, the test case assertion is checked.

```
public void testLogin() { //WebDriver
 WebDriver driver = new FirefoxDriver();
 driver.get("http://www.....com/home.asp");
 HomePage HP = new HomePage(driver);
 HP.login("John.Doe","123456");
 assertTrue(HP.checkLoggedUser("John.Doe"));
}

public void testLogin() { //Sikuli
 CommonPage.open("http://www.....com/home.asp");
 //WebDriver driver = new FirefoxDriver();
 //driver.get("http://www.....com/home.asp");
 HomePage HP = new HomePage();
 HP.login("John.Doe","123456");
 assertTrue(HP.checkLoggedUser());
}
```

Figure 2: TestLogin test case in Selenium WebDriver and in Sikuli API

[3] http://martinfowler.com/bliki/PageObject.html



```
public class HomePage {
 private final WebDriver driver;
 @FindBy(id="UID")
 private WebElement username;
 @FindBy(xpath="./form/input[2]")
 private WebElement password;
 @FindBy(linkText="Login")
 private WebElement login;
 @FindBy(id="LoggedUser")
 private WebElement loggedUser;
 public HomePage(WebDriver driver)
 {this.driver = driver;}
 public void login(String UID, String PW){
  username.sendKeys(UID);
  password.sendKeys(PW);
  login.click();
 }
 public Boolean checkLoggedUser(String name){
  return loggedUser.getText().contains(name);
 }
}
```

Figure 3: HomePage page object in Selenium WebDriver

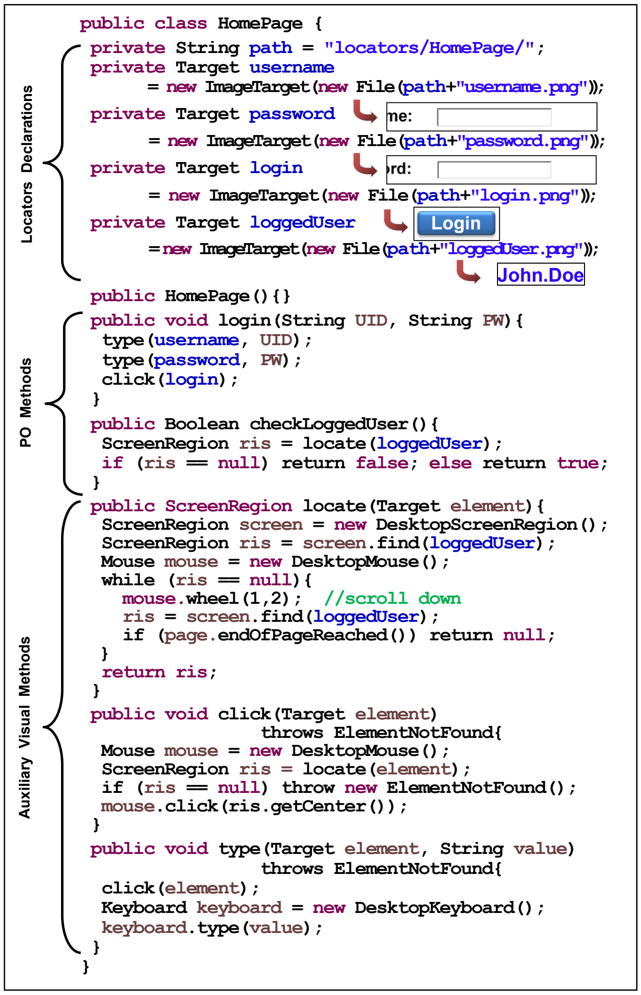
2.3 Visual Web Testing

The testing tool belonging to the Visual category that we used in this work is Sikuli API (for short, Sikuli). Sikuli is a visual tool able to automate and test graphical user interfaces using screenshot images. It provides image-based GUI automation functionalities to Java programmers. We chose Sikuli as the representative of this category mainly because: (1) it is open-source and (2) it is similar to WebDriver, thus, we can create test cases and page objects similar to the ones produced for WebDriver. Indeed, we can use the same programming environment: programming language (Java), IDE (Eclipse), and testing framework (JUnit). Sikuli allows testers to write scripts based on images that define the GUI widgets to interact with and the assertions to be checked.

As an example, the Sikuli version of the testLogin test case is shown in Figure 2 (bottom), while the related page object is given in Figure 4. The test case developed in Sikuli performs the same conceptual steps as the WebDriver test case. The first operation, `CommonPage.open(...)`, aims at opening the browser at a specified URL. In a purely visual test case, it involves identifying and clicking the Firefox icon on the desktop, inserting the URL into the address bar and then clicking on the "go" arrow (these operations are encapsulated in class `CommonPage`).

The following steps are basically the same in Sikuli and WebDriver, the only differences being that in Sikuli `driver` is not a parameter of the `HomePage` constructor and the assertion checking method does not need any string parameter (see the explanation below). On the contrary, Sikuli's page object is quite different from WebDriver's. As shown in Figure 4, command `locate` is invoked to search for the portion of a web page that looks like the image representing the rendering of the web element to be located. The image must have been previously saved in the file system as a file or must be available online. Once the web element has been located, a `ScreenRegion` is returned by method `locate`, which can be used to perform operations such as clicking and typing into it (see, e.g., method `type` in Figure 4).

Thus, in Sikuli *locators are always images*. While using DOM-based tools it is possible to verify whether an HTML element contains textual information (see the last line in Figure 3), with visual tools it is necessary to check that the page contains an image displaying such text (see Figure 4, method `checkLoggedUser`). Moreover, some useful and quite general WebDriver methods are not natively available in Sikuli (e.g., click() and sendKeys()). Thus, when using Sikuli, they must be implemented explicitly in the page object class (e.g., the methods click() and type()). The source code reported in Figure 3 and Figure 4 is very similar to the one we have used/obtained respectively as source/target of our transformation with PESTO.



```
public class HomePage {
 private String path = "locators/HomePage/";
 private Target username
       = new ImageTarget(new File(path+"username.png"));
 private Target password
       = new ImageTarget(new File(path+"password.png"));
 private Target login
       = new ImageTarget(new File(path+"login.png"));
 private Target loggedUser
       =new ImageTarget(new File(path+"loggedUser.png"));
 public HomePage(){}
 public void login(String UID, String PW){
  type(username, UID);
  type(password, PW);
  click(login);
 }
 public Boolean checkLoggedUser(){
  ScreenRegion ris = locate(loggedUser);
  if (ris == null) return false; else return true;
 }
 public ScreenRegion locate(Target element){
  ScreenRegion screen = new DesktopScreenRegion();
  ScreenRegion ris = screen.find(loggedUser);
  Mouse mouse = new DesktopMouse();
  while (ris == null){
    mouse.wheel(1,2);  //scroll down
    ris = screen.find(loggedUser);
    if (page.endOfPageReached()) return null;
  }
  return ris;
 }
 public void click(Target element)
                throws ElementNotFound{
  Mouse mouse = new DesktopMouse();
  ScreenRegion ris = locate(element);
  if (ris == null) throw new ElementNotFound();
  mouse.click(ris.getCenter());
 }
 public void type(Target element, String value)
                throws ElementNotFound{
  click(element);
  Keyboard keyboard = new DesktopKeyboard();
  keyboard.type(value);
 }
}
```

Figure 4: HomePage page object in Sikuli API

3. APPROACH AND TOOL DESCRIPTION

The aim of PESTO (PagE object tranSformation TOol) is converting a DOM-based web test suite, created using WebDriver and adopting the page object (PO) pattern, into a visual web test suite based on the Sikuli image recognition capabilities and still adopting the PO pattern. While we developed PESTO to transform WebDriver test suites to Sikuli, the techniques and architectural solutions adopted for its implementation are quite general and can be easily used within any web test transformation activity involving abstractions similar to the ones provided by the PO.

PESTO executes the transformation by means of two main modules (see Figure 5):

– **Visual Locators Generator (Module 1)** This module generates a visual locator for each web element used by the DOM-based test suite. The details about this step are reported in Section 3.2.

– **Visual Test Suite Transformer (Module 2)** This module transforms the source code of the DOM-based test suite in order to adopt the visual approach. In particular, the majority of the changes are concentrated in the page objects code, since page objects are responsible for the interaction with the web pages. The details about this step are reported in Section 3.3.

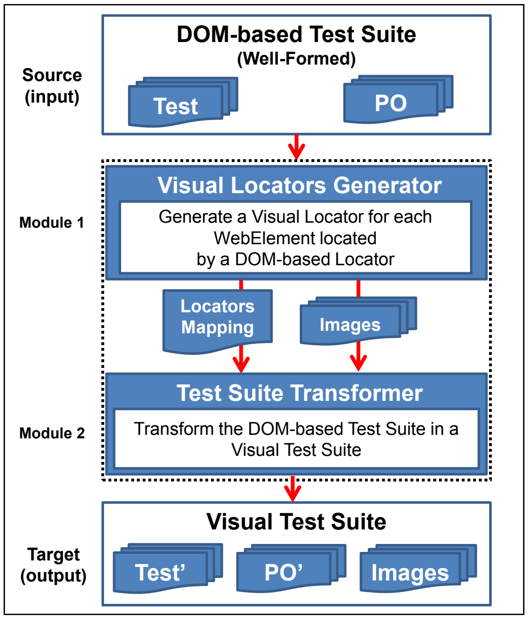


Figure 5: High Level Architecture of PESTO

For the interested reader, a demo video of PESTO and the source code can be found at http://sepl.dibris.unige.it/2014-PESTO.php. This paper extends our previous work [13] providing a deeper description of the approach, more details on the tool implementation and an empirical evaluation of PESTO on four applications.

3.1 Choosing the Commands to Transform

Test suites developed with Selenium WebDriver make use of a wide selection of commands, to control the browser and perform operations on the web elements (e.g., clicking a button or writing in a text field). Even if our future goal is to deal with all commands offered by WebDriver, in this initial work we chose to focus on the automatic transformation of the locator instructions and of the most used web element interaction commands [4]. We based the analysis of the commands occurrences in an open source test suite[4] created for a real size web application: *Moodle*.

We discovered that in this test suite there are 154 click(), 49 sendKeys(), 20 getText(), 25 selections of an option in a drop-down list, 2 clear() used to clear text boxes and areas already filled. We noticed that 223 over a total of 250 command calls used in this DOM-based test suite (i.e., 89,2%) can be managed by means of only three commands (i.e., click(), sendKeys(), and getText()). This is not surprising since the most common interactions with a web application are actually: clicking (e.g., a link, a button), typing (e.g., in a text box, area), and reading (e.g., the string displayed by a web element). We obtained similar results in a test suite we developed during a previous industrial work [9]. In that case the click(), sendKeys(), and getText() commands represented 74,9% of all the commands used. Thus, we decided to create a preliminary version of PESTO able to deal with these three most used commands.

3.2 Visual Locators Generator (Module 1)

To capture the images corresponding to the target web elements, as rendered by the browser at run time, we need to intercept the commands of the test cases while they are executing. For this reason, for the implementation of Module 1 of PESTO, we adopted the aspect-oriented programming paradigm (in particular, the AspectJ[5] language). In this way, PESTO is able to intercept calls to the three most used WebDriver commands and create the corresponding visual locators. Figure 6 summarizes the activities carried out by Module 1. The detailed steps of Module 1 are as follows:

1) During the execution of the test suite, WebDriver interacts with the web elements of the application under test. For instance, it can click on a link or a button using the click() command, fill a textBox or a textArea using the sendKeys(...) command, or retrieve the text shown by a web element using getText().

2) The AspectJ sub-module intercepts these calls (e.g., the call to sendKeys(...), used to fill a textbox) before they are carried out, by means of a *before advice* in AspectJ), and for each of them it performs steps 3–6, indicated with numbered arrows in Figure 6 and detailed below, in order to generate a visual locator for the web element of interest, e.g., the username textBox. If a visual locator for the current web element already exists, the locator generation process for this web element ends (e.g., the visual locator for the username textBox is reused by each test case upon login).

3) The AspectJ sub-module calls a WebDriver method that returns a screenshot of the entire web page (e.g., the login page) containing the web element of interest (e.g., the username textBox).

4) The AspectJ sub-module calls a WebDriver method that returns the following information about the rendered web element: *(i)* the coordinates of the top left-hand corner and *(ii)* its size (i.e., width and height).

5) The AspectJ sub-module invokes the Visual Locator Creator that is able to generate a visual locator for the web element of interest. As shown in Figure 6, often an image representing only the web element cannot be considered a locator, since it cannot uniquely locate it. For instance, this happens with forms, in which usually all text boxes have the same size and appearance. In these cases, multiple matches would be found for the perfectly cropped image representing the web element. For this reason, in such cases the Visual Locator Creator expands the size of the rectangle image until a unique locator is found (n expansion steps are indicated in Figure 6). The only requirement is that the web element of interest must be kept at the geometric centre of the visual locator, since Sikuli executes the click at the centre of the area that matches the visual locator.

6) The AspectJ module saves the association between DOM-based locator and visual locator. In detail, for each web element the saved mapping consists of a triple:

(PO Name, Locator Type and Value, Image Path)

like for instance:

(HomePage, id="UID", HomePage/username.png)

7) When the test suite execution is completed, the Visual Locators Generator provides two outputs: (1) a set of folders, one for each page object, containing the saved images (i.e., visual locators), and (2) a mapping file that associates the DOM-based locators to their corresponding visual locators.

3.3 Test Suite Transformer (Module 2)

Module 2, starting from the information provided by the first module, transforms the DOM-based test suite into a visual one as summarized in Figure 7.

The source code transformation has been implemented using the JavaParser[6] library. The *Test Cases Transformation* module modifies the test cases source code in order to use the new visual page objects (PO') instead of the original DOM-based page objects (PO). This

[4] https://github.com/moodlehq/functional-test-suite

[5] http://eclipse.org/aspectj/

[6] https://code.google.com/p/javaparser/

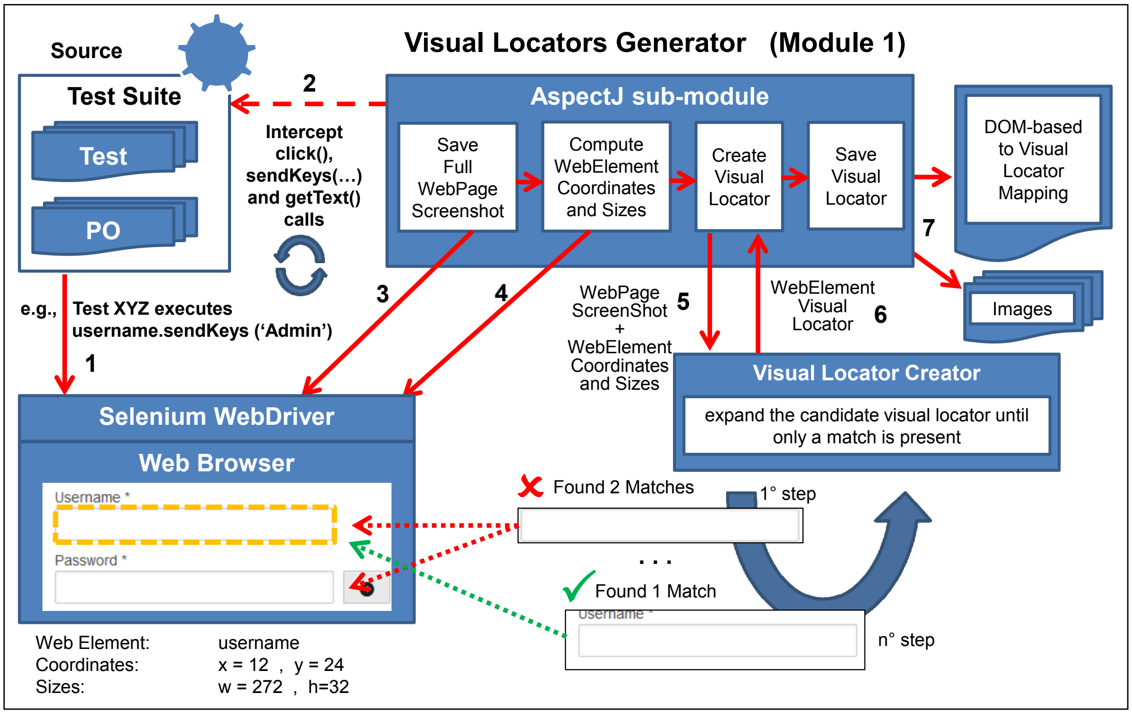

Source
Test Suite
Test
PO
2
Intercept click(), sendKeys(…) and getText() calls
Visual Locators Generator (Module 1)
AspectJ sub-module
Save Full WebPage Screenshot
Compute WebElement Coordinates and Sizes
Create Visual Locator
Save Visual Locator
DOM-based to Visual Locator Mapping
7
Images
3
4
WebPage ScreenShot + WebElement Coordinates and Sizes
5
WebElement Visual Locator
6
e.g., Test XYZ executes username.sendKeys ('Admin')
1
Visual Locator Creator
expand the candidate visual locator until only a match is present
Selenium WebDriver
Web Browser
Username *
Password *
Found 2 Matches
1° step
. . .
Found 1 Match
Username *
n° step
Web Element: username
Coordinates: x = 12 , y = 24
Sizes: w = 272 , h=32

Figure 6: Module 1: Visual Locators Generator

step requires to change a few import instructions, so as to replace the old POs with the new ones. Then, the *Page Objects Transformation* module transforms the DOM-based page objects into new visual page objects, by executing the following three steps.

– **Replace DOM-based WebElements Declarations.** In this step, each declaration of a WebElement located by a DOM-based locator is removed from the page object and replaced with the declaration of an ImageTarget that points to the image representing the visual locator. For example, in the case of `username` (see Figure 3 and Figure 4) we have the following transformation (the arrow means "is transformed to"):

```
@FindBy(id="UID")
private WebElement username;
                              →
private Target username
        = new ImageTarget(new File(path+"username.png"));
```

using the information found in the mapping file:

```
(HomePage, id="UID", HomePage/username.png)
```

The result of this transformation is also apparent in our running example, Figure 3 and Figure 4 (*Locators Declarations* portion).

– **Insert Auxiliary Visual Methods.** As already mentioned, Sikuli provides only methods to simulate low-level mouse and keyboard operations, through its *mouse* and *keyboard* interfaces. For instance, it is not possible to write directly inside an input field using something similar to the WebDriver command sendKeys(...). To address this issue, we automatically generate some auxiliary methods (locate(), click(), and type()) that simulate the ones provided by WebDriver. Figure 4 (*Auxiliary Visual Methods* portion) shows a simplified output of this transformation, where minor details, such as, e.g., the Thread.sleep(...) calls — necessary to wait for page loading — are omitted. The following three auxiliary methods are automatically inserted into the new visual page objects (PO'):

locate(). Sikuli interacts with a web element only if it is currently displayed on the screen. The locate() method searches the target

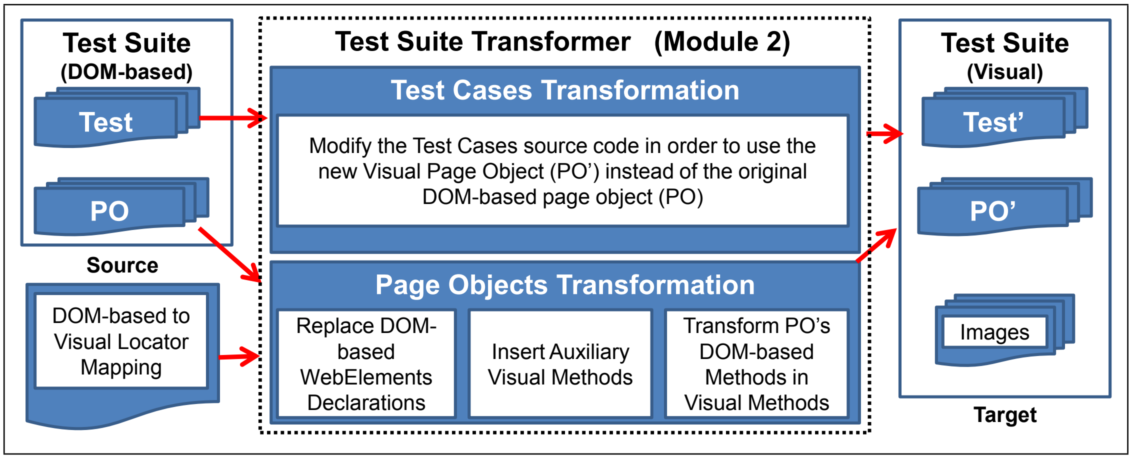

Test Suite (DOM-based)
Test
PO
Source
DOM-based to Visual Locator Mapping
Test Suite Transformer (Module 2)
Test Cases Transformation
Modify the Test Cases source code in order to use the new Visual Page Object (PO') instead of the original DOM-based page object (PO)
Page Objects Transformation
Replace DOM-based WebElements Declarations
Insert Auxiliary Visual Methods
Transform PO's DOM-based Methods in Visual Methods
Test Suite (Visual)
Test'
PO'
Images
Target

Figure 7: Module 2: Test Suite Transformer

element in the visible portion of the page by applying the Sikuli image recognition algorithm. If the target element is not present, the method automatically scrolls the page down and repeats the search. If the end of the page is reached (i.e., the scroll does not modify the page visualization anymore) and no match is found, the search fails and null is returned.

click(). In this case it is necessary to: (1) locate the web element; and, (2) click on it. The first step is done by calling the above defined method locate(). Then, a click is triggered at the central point where the match is found. If no match is found, an ElementNotFound exception is raised.

type(...). In this case it is necessary to: (1) locate the web element; (2) click on it; and, (3) type into it. The first two steps are done by calling the above defined method click(). Then, it is sufficient to use the *keyboard* interface of Sikuli.

– **Transform PO's DOM-based Methods into Visual Methods.** Each method of the page object is transformed in order to adopt the visual approach. For instance, each call to the sendKeys method provided by WebDriver (e.g., username.sendKeys("Admin")) becomes a call to the new visual method (type(username, "Admin")). Note that, in the case of the WebDriver test suite, username is a WebElement (a specific WebDriver type) located using a DOM-based locator, while in the Sikuli test suite, username is an ImageTarget representing the visual locator (i.e., an image). In detail, the transformed method calls are:

```
webElement.click() → click(webElement)
webElement.sendKeys(...) → type(webElement,...)
return webElement.getText().contains(name);
                              →
ScreenRegion ris = locate(webElement);
if (ris == null) return false; else return true;
```

The input/output of this transformation can be seen in Figure 3 and Figure 4 (*PO Methods* portion), for our running example.

For the test case assertions we adopt a specific transformation template. In particular, PESTO assumes that each assertion contains a call to a page object method that takes in input the expected textual value (the string to match with the text contained in a web element) and returns a boolean value (in practice we used the JUnit assertTrue statement). Thus, the comparison between the expected value and the actual text contained in the web element is executed in the page object (e.g., see the assertion implementation in the WebDriver test case shown in Figure 2 and the corresponding PO method shown in Figure 3). Our transformation produces a visual page object where each textual match in the original assertions becomes an image match. In this way, we maintain a similar semantics as that of getText(...), even if the command execution is not exactly the same, since an image is provided as expected value instead of a string. Indeed, the assertions of the DOM-based test cases check that the value of the expected string is contained in the target web element, while in the case of the visual test cases the assertions check that the target web element is displayed in the web page with the expected visual appearance. Generation of the images that replace the strings used in the assertions is performed automatically by Module 1.

4. PRELIMINARY EVALUATION

This section describes the experimental results obtained by applying PESTO to four test suites developed to test the most relevant functionalities of four open-source web applications. In particular, our empirical study aims at answering the following research questions:

RQ1 (automation): *Is the migration process fully automated? Is any manual intervention required to compile and execute the migrated test suites?*

RQ2 (correctness): *Do the migrated test suites locate and interact with the web elements under test correctly? Do assertions check each test case outcome correctly?*

4.1 Web Applications

We selected and downloaded four open-source web applications from *SourceForge.net*. We have included only applications that: (1) are quite recent, so that they can work without problems on the latest versions of Apache, PHP and MySQL, technologies we are familiar with (actually, since WebDriver and Sikuli implement a black-box approach, the server side technologies do not affect the results of our study); (2) are well-known and used (some of them have been downloaded more than one hundred thousand times last year); and (3) belong to different application domains.

Table 1 reports some information about the selected applications. We can see that all of them are quite recent (ranging from 2008 to 2013). They are considerably different in terms of number of source files (ranging from 63 to 835) and number of lines of code (ranging from 4 kLOC to 285 kLOC, considering only the lines of code contained in the PHP source files, comments and blank lines excluded).

4.2 Experimental Procedure

For each of the four selected applications, the following steps have been performed:

– **DOM-based Test Suite Development.** A DOM-based test suite has been developed by two of the authors in the context of prior empirical work on web testing [10] and independently of the present work on test suite migration, when PESTO was not even planned to exist. A systematic approach was adopted for test suite creation, consisting of three steps: (1) the main functionalities of the target web application have been identified from the available documentation; (2) each discovered functionality has been covered by at least one test case (a meaningful name was assigned to each test case, so as to keep the mapping between test cases and functionalities); (3) each test case has been implemented with WebDriver. The test suite adopts the *Page Object* and *Page Factory* patterns, and contains only assertions on web elements that are rendered visually (indeed, assertions on hidden fields would be quite problematic to migrate to any visual testing tool).

Table 1: WEB APPLICATIONS FROM *SourceForge.net*

	Description	Web Site	Release Info			
			Vers.	Date	File[a]	kLOC[b]
PPMA[c]	password manager	sourceforge.net/projects/ppma/	0.2	2011	93	4
Claroline	learning environment	sourceforge.net/projects/claroline/	1.11.5	2013	835	285
Address Book	address/phone book	sourceforge.net/projects/php-addressbook/	8.2.5	2012	239	30
MRBS	meeting rooms manager	sourceforge.net/projects/mrbs/	1.2.6.1	2008	63	9

[a] Only PHP source files were considered

[b] PHP LOC - Comment and Blank lines are not considered

[c] Without considering the source code of the framework used by this application (Yii framework - http://www.yiiframework.com/)

Table 2: DOM-BASED TEST SUITES

	Test Cases	Page Objects	LOC	WebDriver Web Element	WebDriver Command Calls			
					click()	sendKeys()	getText()	others
PPMA	18	4	687	52	23	29	12	2
Claroline	19	4	645	65	46	25	17	4
Address Book	16	5	576	52	23	14	13	3
MRBS	15	7	648	44	25	13	16	1
	68	**20**	**2556**	**213**	**117**	**81**	**58**	**10**

[a] Java LOC - Comment and Blank lines are not considered

– **DOM-based Test Suite Transformation.** The DOM-based test suite has been executed in order to allow the Visual Locators Generator (Module 1) to create the visual locators and the mapping file. Then, the Visual Test Suite Transformer (Module 2) has generated the visual test suite.
– **Visual Test Suite Evaluation.** The freshly generated visual test suite has been executed to check its correctness.

4.3 Experimental Results and Discussion

Table 2 summarizes the most important information about the four DOM-based test suites. In particular, each test suite has from 15 to 19 test cases and from 4 to 7 page objects. The sizes of the four test suites, measured as lines of Java code, are quite similar, ranging from 576 to 687 LOCs. The number of WebDriver WebElements, used to declare the web elements the test cases interact with ranges from 44 in the case of MRBS to 65 in the case of Claroline. Columns 5-7 report, respectively, the number of click(), sendKeys(), and getText() command calls used in each test suite. The last column reports the number of other command calls employed to interact with various kinds of elements, currently not handled by PESTO (only 10 out of 266). These are: select an element of a drop-down list (two cases), click the OK button of an alert window (seven cases), clear an already filled text box (one case).
– **RQ1.** All the 256 handled command calls (click() (117), sendKeys() (81), and getText() (58)) used in the target DOM-based test suites have been transformed automatically into their corresponding visual versions. Moreover, for all the 213 WebDriver WebElements, located by DOM-based locators, PESTO was able to automatically generate a corresponding visual locator. It should be noticed that the total number of command calls is higher than the number of WebElements since the same WebElement can be used in different methods by different calls. In our experiments, PESTO was not able to transform only 10 command calls (last column of Table 2) out of 266. In these ten cases, PESTO simply copied the commands from the original test suite to the output test suite, hence creating a *hybrid test suite*. This means that the resulting test suite can be compiled and executed without requiring any manual intervention. However, in such cases the generated test suite contains also DOM-based commands. To turn it into a fully visual, 3rd generation test suite, the ten commands that are currently not handled by PESTO should be transformed manually. The migrated test suites contain 96.2% of visual command calls and just 3.8% of residual DOM-based command calls.

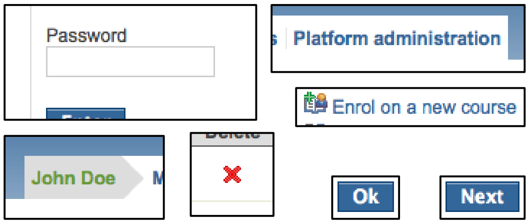


Figure 8: Examples of visual locators automatically generated by PESTO for the Claroline web application

– **RQ2.** According to our manual analysis, there were no web element localisation, interaction or assertion errors in the resulting test suites. When we executed the generated test suites we had the same results obtained with the original DOM-based ones. In summary, PESTO was able to automatically produce four compilable, executable and fully working test suites.
Locators Readability: The Visual Locators Generator algorithm of PESTO has been designed to mimic the visual selection strategy of a human tester, i.e., crop a rectangle area around the element of interest, together with specific visual features. The authors analysed manually the visual locators generated by PESTO, finding them highly understandable, since it was easy to retrieve the corresponding target elements on the web page (see some examples in Figure 8).

4.4 Threats of Validity of the Study

The main threat to validity that affect this study is authors' bias. The authors' bias concerns the involvement of the authors in manual activities conducted during the empirical study and the influence of the authors' expectations about the empirical study on such activities. In our case, two of the authors developed the input test suites before the tool PESTO was conceived, as part of another research work. It might be the case that other testers, asked to follow the procedure described in Section 4.2, would have developed test suites resulting in a different percentage of residual DOM-based commands left in the migrated test suites (3.8% in our case). However, the answers to RQ1 and RQ2 would be largely unaffected, since most locators that are handled by PESTO would be probably the same as those used by the two authors.

5. UPCOMING IMPROVEMENTS

Although PESTO is able to handle the three most widely used WebDriver commands, it is not yet complete and it has some limitations that we plan to address in the near future. In particular, current limitations concern: *(i)* the set of handled WebDriver commands, *(ii)* the prerequisites required by PESTO about the input test suites (e.g., the structure of assertions, see Section 3.3), *(iii)* web elements that require a complex visual interaction and *(iv)* web elements that change their visual appearance during the execution of a test suite. We are gradually extending the set of commands and assertions supported by PESTO, hence addressing *(i)* and *(ii)*. The other two issues (*(iii)* and *(iv)*) are more demanding:

Elements with Complex Interaction. PESTO is currently not able to handle complex web elements, such as drop-down lists and multilevel drop-down menus. For instance, let us consider a registration form that asks for the nationality of the submitter. This can be implemented using a drop-down list containing a list of countries. A DOM-based tool like WebDriver can provide a command to select directly an element from the drop-down list (only one locator is required). On the contrary, when adopting a visual approach, the task is much more complex. One has to: (1) locate the drop-down list (more precisely the arrow that expands the list) using an image locator; (2) click on it; (3) if the required list element is not shown, locate and move the scrollbar (e.g., by clicking the arrow); (4) locate the required element using another image locator; and, finally, (5) click on it. Actually, in this case the visual approach performs exactly the same steps that a human tester would do. PESTO will be improved to automatically generate such complex interaction sequence.

Web Elements Changing their State. When a web element changes its state (e.g., a check box is checked or unchecked, or an input field is emptied or filled), a visual locator must be created for each state, while with the DOM-based approach only one locator is required. Since PESTO currently associates only one visual locator to each DOM-based locator, it is not able to interact more than once with a web element that changes its visual appearance upon interaction. A solution to this limitation consists in associating multiple visual locators to each DOM-based locator in case the target web element changes its appearance. This can be done by using, for instance, the Sikuli MultiStateTarget[7] construct.

6. RELATED WORK

To the best of our knowledge, no automated transformation tool exists for the creation of 3rd generation web test suites from 2nd generation. PESTO is the first attempt to assist the tester in the thorny task of transforming a DOM-based test suite towards one employing visual recognition capabilities. Even though there is no strictly related work, some related papers have been published in the context of refactoring. *Deiß* [6] describes TTtwo2three, an automatic tool for the conversion of TTCN-2 test systems to TTCN-3 at Nokia. The tool realises semantic and syntactic transformations, but some manual refactoring is needed to ensure that the test cases behave as expected. TTtwo2three has been used to convert two industrial test suites, a Bluetooth Serial Port Profile and a UMTS network element, consisting of about 2,500 test cases. *Chu et al.* [5] propose a tool to guide test case refactoring after having applied well-established pattern-based code refactoring. While refactoring the application, using a series of patterns, the plugin records all the useful steps and information. This allows the plugin to create a mapping relationship between pattern refactoring and test case refactoring. Such mapping is used to transform the test case source code automatically, in accordance with the source code refactorings. *Ricca et al.* [12], automate the reengineering of web applications adopting the DMS Reengineering toolkit, a program transformation system, and evaluate its applicability to a real world case study. *Ding et al.* [7] propose a black-box approach for testing of web applications after migration towards a new technology (e.g., a cloud system) without manually creating test cases. Responses of the migrated application are automatically compared against those from the original production one. Possible mismatches due to migration problems can be detected automatically.

7. CONCLUSIONS AND FUTURE WORK

This work proposed and experimented PESTO, a tool able to automatically transform DOM-based web test suites developed using Selenium WebDriver into visual test suites relying on the usage of Sikuli API. Current version of PESTO handles the three most widely used DOM-based commands: click(), sendKeys() and getText(). If present, any other DOM-based command is simply copied to the target test suite, which becomes a hybrid DOM-based / visual test suite. In our empirical study, PESTO was able to migrate 96.2% of the commands used in existing test suites. We are extending PESTO in order to cover the remaining 3.8% residual DOM-based commands.

PESTO has been validated on four DOM-based test suites, used to test four different web applications. The visual test suites produced automatically by PESTO have been compiled and executed with no need for manual interventions or adjustments, since the automatically transformed test cases exhibited the correct, expected behaviour. The visual locators automatically generated by PESTO have been checked for readability and they appeared easy to understand. In our future work, we intend to improve PESTO in order to handle: (1) web elements with complex interaction (e.g., drop-down list), (2) web elements changing their state.

8. REFERENCES

[1] E. Alegroth, M. Nass, and H. H. Olsson. JAutomate: A tool for system- and acceptance-test automation. In *Proc. of 6th International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 439–446. IEEE, 2013.

[2] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proc. of 28th Conference on Human Factors in Computing Systems (CHI 2010)*, pages 1535–1544. ACM, 2010.

[3] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proc. of 18th Conference on Computer and Communications Security (CCS 2011)*, pages 263–274. ACM, 2011.

[4] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Proc. of 30th Int. Conference on Software Maintenance and Evolution (ICSME 2014)*. IEEE, 2014.

[5] P.-H. Chu, N.-L. Hsueh, H.-H. Chen, and C.-H. Liu. A test case refactoring approach for pattern-based software development. *Software Quality Journal*, 20(1):43–75, 2012.

[6] T. Deiß. Refactoring and converting a ttcn-2 test suite. *International Journal on Software Tools for Technology Transfer*, 10(4):347–352, 2008.

[7] X. Ding, H. Huang, Y. Ruan, A. Shaikh, B. Peterson, and X. Zhang. Splitter: A proxy-based approach for post-migration testing of web applications. In *Proc. of EuroSys 2010*, pages 97–110. ACM, 2010.

[8] L. Hayes. *The Automated Testing Handbook*. Software Testing Institute, 2004.

[9] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Proc. of 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*, pages 108–113. IEEE, 2013.

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

[11] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In S. Casteleyn, G. Rossi, and M. Winckler, editors, *Proc. of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *LNCS*, pages 322–340. Springer, 2014.

[12] F. Ricca, P. Tonella, and I. D. Baxter. Web application transformations based on rewrite rules. *Information & Software Technology*, 44(13):811–825, 2002.

[13] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. PESTO: A tool for migrating DOM-based to visual web tests. In *Proc. of 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 65–70. IEEE, 2014.

[7] https://code.google.com/p/sikuli-api/wiki/MultiStateTarget