Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution

Maurizio Leotta, Diego Clerissi, Filippo Ricca, Paolo Tonella

Abstract:

There are several approaches for automated functional web testing and the choice among them depends on a number of factors, including the tools used for web testing and the costs associated with their adoption. In this paper, we present an empirical cost/benefit analysis of two different categories of automated functional web testing approaches: (1) capture-replay web testing (in particular, using Selenium IDE); and, (2) programmable web testing (using Selenium WebDriver). On a set of six web applications, we evaluated the costs of applying these testing approaches both when developing the initial test suites from scratch and when the test suites are maintained, upon the release of a new software version.

Results indicate that, on the one hand, the development of the test suites is more expensive in terms of time required (between 32% and 112%) when the programmable web testing approach is adopted, but on the other hand, test suite maintenance is less expensive when this approach is used (with a saving between 16% and 51%). We found that, in the majority of the cases, after a small number of releases (from one to three), the cumulative cost of programmable web testing becomes lower than the cost involved with capture-replay web testing and the cost saving gets amplified over the successive releases.

Digital Object Identifier (DOI):

http://dx.doi.org/10.1109/WCRE.2013.6671302

Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution

Maurizio Leotta[1], Diego Clerissi[1], Filippo Ricca[1], Paolo Tonella[2]
[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
maurizio.leotta@unige.it, diego.clerissi@gmail.com, filippo.ricca@unige.it, tonella@fbk.eu

***Abstract*—There are several approaches for automated functional web testing and the choice among them depends on a number of factors, including the tools used for web testing and the costs associated with their adoption. In this paper, we present an empirical cost/benefit analysis of two different categories of automated functional web testing approaches: (1) capture-replay web testing (in particular, using Selenium IDE); and, (2) programmable web testing (using Selenium WebDriver). On a set of six web applications, we evaluated the costs of applying these testing approaches both when developing the initial test suites from scratch and when the test suites are maintained, upon the release of a new software version.**

Results indicate that, on the one hand, the development of the test suites is more expensive in terms of time required (between 32% and 112%) when the programmable web testing approach is adopted, but on the other hand, test suite maintenance is less expensive when this approach is used (with a saving between 16% and 51%). We found that, in the majority of the cases, after a small number of releases (from one to three), the cumulative cost of programmable web testing becomes lower than the cost involved with capture-replay web testing and the cost saving gets amplified over the successive releases.

***Index Terms*—Test Case Evolution, Test Case Repair, Empirical Study, Selenium IDE, Selenium WebDriver.**

I. Introduction

Web applications provide critical services to our society, ranging from the financial and commercial sector, to the public administration and health care. The widespread use of web applications as the natural interface between a service and its users puts a serious demand on the quality levels that web application developers are expected to deliver. At the same time, web applications tend to evolve quickly, especially for what concerns the presentation and interaction layer. The release cycle of web applications is very short, which makes it difficult to accommodate quality assurance (e.g., testing) activities in the development process when a new release is delivered. For these reasons, the possibility to increase the effectiveness and efficiency of web testing has become a major need and several methodologies, tools and techniques have been developed over time [5], [4], [9], [12], [13].

Among the recently developed approaches to web testing, we can recognize two major trends, associated with a profoundly different way of facing the problem. On the one hand, *capture-replay (C&R) web testing* is based on the assumption that the testing activity conducted on a web application can be better automated by recording the actions performed by the tester on the web application GUI and by generating a script that provides such actions for automated, unattended re-execution. On the other hand, *programmable web testing* aims at unifying web testing with traditional testing, where test cases are themselves software artefacts that developers write by resorting to specific testing frameworks. For web applications, this means that the framework has to support an automated, unattended interaction with a web page and its elements, so that test cases can, for instance, automatically fill-in and submit forms or click on hyperlinks.

C&R test cases are very easy to obtain and actually do not require any advanced testing skill. Testers just exercise the web application under test and record their actions. However, during software evolution the test suites developed using a C&R approach tend to be quite fragile. A minor change in the GUI might break a previously recorded test case, whose script needs to be repaired manually, unless it is re-recorded from scratch, on the new version of the web application. Programmable test cases require non trivial programming skills, the involved effort being comparable to that required for normal code development. However, all benefits of modular programming can be brought also to the test cases, such as parametric and conditional execution, reuse of common functionalities across test cases, robust mechanisms to reference the elements in a web page.

In this paper, we empirically investigate the trade-off between C&R and programmable web testing. In addition to validating our initial hypothesis, that C&R test cases are cheaper to write from scratch than programmable test cases, but are also more expensive to maintain during software evolution, we want to estimate quantitatively the return on the investment. Specifically, we determine the number of software releases after which the savings of programmable test cases overcome the costs initially paid for their development. We have measured such trade-off on six real web applications. The findings reported in this paper include a detailed analysis of the features of each web application that make the investigated trade-off more or less favourable in each specific context, so as to derive some practical guidelines for developers who want to make an informed decision between the two alternatives.

The paper is organized as follows: Section II gives some background on test case development using the C&R vs. the programmable approach, by considering two specific tools that implement them. Section III describes the test case repair activities and provides a mathematical model of the investigated

trade-off. Section IV reports our experimental results and our detailed discussion of the pros and cons of the two approaches in specific scenarios. We then present the related works, followed by conclusions and future work.

II. Approaches to Web Testing

There are several approaches for functional web testing and the choice among them depends on a number of factors, including the technology used by the web application and the tools (if any) used for web testing. In this section, we consider two different categories of automated functional web testing approaches: C&R and programmable web testing.

A. Capture-Replay Web Testing

C&R web testing is based on capture/replay automated tools. Capture/replay tools have been developed as a mechanism for testing the correctness of interactive applications (GUI or Web applications). Using a capture/replay tool, a software tester can run a Web application and record the entire session. The tool records all the user's events on the navigated Web pages, such as the key presses and mouse clicks, in a script, allowing a session to be rerun automatically without further user interaction. Finally, a test case is produced by adding one or more assertions to the recorded script. By replaying a given script on a changed version of the Web Application Under Test (WAUT), capture/replay tools support automatic regression testing.

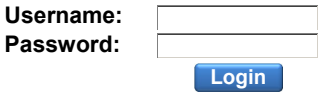
Building test cases using this approach is a relatively simple task. Even persons without programming expertise can easily build a complete test suite for a complex Web application. However, there are some drawbacks: (1) the scripts resulting from this method contain hard-coded values (the inputs), that have to be changed if anything changes during the evolution of the WAUT; (2) the test cases are strongly coupled with web pages, with the consequence that even a small change in the WAUT (e.g., in the layout of the page) leads to one or more broken test cases (e.g., test scripts fail to locate a link, an input field or a submission button because of the layout change).

A tool for Web application testing belonging to this category is Selenium IDE[1]. Selenium IDE is not only a capture/replay tool: it is a complete IDE, as suggested by its name. In particular, Selenium IDE provides several smart features such as: (1) it allows software testers to record, edit, and debug test cases expressed in the Selenese language[2]; (2) it supports smart field selection (e.g., using IDs, names, or XPath locators, as needed) and it offers a *locator assistance* function; (3) it can save test cases as HTML, Java code, or any other format; and (4) it suggests assertions.

Let us assume we have to test a portion of a web application that authenticates users. In a very simplified case, we would have a login page (e.g., called login.asp) that requires the users to enter their credentials, i.e., *username* and *password* (see Fig. 1). After having inserted the credentials and clicked on

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

[2] Each Selenese line is a triple: (command, target, value). See: http://release.seleniumhq.org/selenium-core/1.0.1/reference.html




```
<form name="loginform" action="homepage.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>
```

Fig. 1. login.asp – Page and Source

"Login" the application shows the home page (homepage.asp). If credentials are correct, the username (contained in a HTML tag with the attribute `ID="uname"`) and the logout button are reported in the upper right corner of the home page (e.g., *John.Doe*). Otherwise, the *Guest User* and the login button are shown.

As an example, we report a test case for this simple web application built using the capture/replay facility of Selenium IDE (see Fig. 2). The test script produced by Selenium IDE performs a valid login, using correct credentials (i.e., *username=John.Doe* and *password=123456*) and verifies that in the home page the user results to be correctly authenticated (`assertText, id=uname, John.Doe`).

B. Programmable Web Testing

Programmable web testing is based on manual creation of a test script. In web testing, a test script is a set of instructions/commands written by the developer and executed to test that the WAUT functions as expected. Web test script can be written using ad-hoc languages/frameworks or general purpose programming languages (such as, e.g., C++, Java, and Ruby) with the aid of specific libraries able to act as a browser. Usually, these libraries extend the programming language with user friendly APIs, providing commands to, e.g., click a button, fill a field and submit a form. Also in this case, scripts are then completed with assertions (e.g., JUnit assertions if the language chosen for implementing the test scripts is Java).

Test scripts built in this way are more flexible than test scripts built using capture/replay tools. In fact, programming languages allow developers to handle conditional statements (to program multiple test cases with different, condition-dependent behaviours), loops (to program repetitive web interactions, repeated multiple times), logging, and exceptions directly in the test script. They also support the creation of *parameterized* (a.k.a., *data-driven*) *test cases*. Parameterized test cases are test cases that are executed multiple times, each time passing them different arguments (i.e., inputs and expected values). Thus, in general, programmable web testing techniques can handle the complexity of web software better than C&R web testing techniques. Moreover, it is common opinion that programmable test cases can be modified more easily than

Command	Target	Value
open	login.asp	
type	id=UID	John.Doe
type	id=PW	123456
clickAndWait	id=login	
assertText	id=uname	John.Doe

Fig. 2. Selenium IDE test case

```
public class LoginPage {
 private final WebDriver driver;
 public LoginPage(WebDriver driver) {this.driver = driver;}
 public HomePage login(String UID, String PW) {
  driver.findElement(By.id("UID")).sendKeys(UID);
  driver.findElement(By.id("PW")).sendKeys(PW);
  driver.findElement(By.id("login")).click();
  return new HomePage(driver);
 }
}

public class HomePage {
 private final WebDriver driver;
 public HomePage(WebDriver driver) {this.driver = driver;}
 public String getUsername() {
  return driver.findElement(By.id("uname")).getText;
 }
}
```

Fig. 3. LoginPage.java and HomePage.java page objects

C&R test cases. However, "everything has a price". First, test script development, to be effective, has to be subjected to the same programming guidelines and best practices that are applied to traditional software development (e.g., verifying the script quality and using design patterns). Thus, to make an effective use of programmable web testing, technical skills and persons (i.e., code developers) are required. Second, a remarkable initial effort is necessary to develop and fine tune the test suite.

A tool for Web application testing belonging to this category is Selenium WebDriver[3]. Selenium WebDriver is a tool for automating web application testing that provides a comprehensive programming interface used to control the browser. WebDriver test cases are implemented manually in a programming language integrating Selenium WebDriver commands with JUnit or TestNG assertions (it is not tied to any particular test framework). A best practice often used by Selenium WebDriver testers is applying the *page object* pattern.

The *page object* pattern is used to model the web pages involved in the test process as objects, employing the same programming language used to write the test cases. In this way, the functionalities offered by a web page become methods exposed by the corresponding page object, which can be easily called within any test case. Thus, all the details and mechanics of the web page are encapsulated inside the page object. Adopting the *page object* pattern allows the test developer to work at a higher level of abstraction: the *page object* pattern is used to reduce the coupling between web pages and test cases. For these reasons, adopting the *page object* pattern is expected to improve test suite maintainability [7].

As an example, we report two simple WebDriver test cases: a successful authentication test case and an unsuccessful one. The first is equivalent to the one seen for Selenium IDE (Fig. 2). The second test case inserts invalid credentials and verifies that in the home page no user has been authenticated (i.e., *Guest* must be displayed in the home page).

The first step is to create two page objects (LoginPage.java and HomePage.java) corresponding to the web pages login.asp and homepage.asp respectively (see Fig. 3). The page object LoginPage.java offers a method to log into the application. That method takes in input a username and a password, inserts

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

```
public void testLoginOK() {
 WebDriver driver = new FirefoxDriver();
 // we start from the 'login.asp' page
 driver.get("http://www.....com/login.asp");
 LoginPage LP = new LoginPage(driver);
 HomePage HP = LP.login("John.Doe","123456");
 // we are in the 'homepage.asp' page
 assertEquals("John.Doe", HP.getUsername());
 driver.close();
}

public void testLoginKO() {
 WebDriver driver = new FirefoxDriver();
 // we start from the 'login.asp' page
 driver.get("http://www.....com/login.asp");
 LoginPage LP = new LoginPage(driver);
 HomePage HP = LP.login("Inexistent","Inexistent");
 // we are in the 'homepage.asp' page
 assertEquals("Guest", HP.getUsername());
 driver.close();
}
```

Fig. 4. TestLoginOK and TestLoginKO test cases

them in the corresponding input fields, clicks the Login button and returns a page object of kind HomePage.java (because after clicking the Login button the application moves to the page homepage.asp). HomePage.java contains a method that returns the username authenticated in the application or *Guest* when no user is authenticated. In these page objects, we have used the values of the id attributes to locate the HTML tags.

The second step is to develop the two test cases making use of those page objects (see Fig. 4). Specifically, we can see that in each test method: first, a WebDriver of type FirefoxDriver is created allowing to control the Firefox browser as a real user does (Selenium allows to instantiate also several other browsers); second, the WebDriver (i.e., the browser) opens the specified URL and creates the page object, that is an instance of LoginPage.java, relative to the page login.asp; third, using the method login(...) offered by the page object, a new page object (HP) representing the page homepage.asp is created; finally, the test case assertion can be checked using method getUsername().

III. Test Case Evolution

A. Test Repair

When a Web application evolves to accommodate requirement changes, bug fixes, or functionality extensions, test cases may become broken (e.g., test cases may be unable to locate some links, input fields and submission buttons), and software testers have to repair them. This is a tedious and expensive task since it has to be manually performed by software testers (automatic evolution of test suites is far from being consolidated [10]).

Depending on the kind of maintenance task that has been performed on the target web application, a software tester has to execute a series of test case repairment activities that can be categorized, for the sake of simplicity, in two types: *logical* and *structural*.

The first kind (logical) refers to major functional changes which involve the modification of the web application logic. This kind of change requires to change correspondingly the logic of one or more test cases (e.g., modifying a series of commands in a test case because the path to reach a certain

Web page has changed or the target functionality has changed). An example of a change request (CR1) that necessitates of a logical repairment activity is enforcing security by means of stronger authentication. In order to prevent possible brute force attacks, a new web page containing an additional question can be added after the login.asp page of Fig. 1.

The second kind (structural) refers to a change at level of page layout/structure only. For example (CR2), in the web page of Fig. 1 the id of the password textbox may be changed to `PWD`. Usually, the impact of a structural change is smaller than a logical change. Often, it is sufficient to modify one or more localization lines, i.e., lines containing locators that are enclosed in page objects (assuming they are adopted).

The strategy used by a software tester to repair a test case depends mainly on two factors: (1) the tool used to build the test cases (C&R, like Selenium IDE, or programmable, like Selenium WebDriver) and (2) the kind of change (logical or structural).

Selenium IDE + structural change. The tester modifies, directly in the Selenium IDE, the first broken action command (i.e., the Selenese command that is highlighted in red after test case execution), which can be a localization command or an assertion. Then, the tester re-executes the test case, possibly finding the next broken action command (if any). For example, if CR2 is implemented then the test case in Fig. 2 needs to be repaired. The tester has to replace `PW` with `PWD` in the command highlighted in blue in Fig. 2.

Selenium IDE + logical change. The tester keeps the portion of script up to the command that precedes the broken action command, deletes the rest and captures the new execution scenario by starting from the last working command. For example, if CR1 is implemented then the assertion of the test case shown in Fig. 2 fails and the tester has to delete it. Then, the tester has to complete the test case starting from the command `clickAndWait, id=login` and capturing the new scenario, that includes the new web page providing the additional authentication question.

Selenium WebDriver + structural change. The tester modifies one or more page objects that the broken test case links to. For example, if CR2 is implemented then the tester has to repair the line: `driver.findElement(By.id("PW")).sendKeys(PW)` in the `LoginPage.java` page object (see Fig. 3).

Selenium WebDriver + logical change. Depending on the magnitude of the executed maintenance task, the tester has to modify the broken test cases and/or the corresponding page objects. In some cases, new page objects have to be created. For example, if CR1 is implemented then the tester has to create a new page object for the web page providing the additional authentication question. Moreover, she has to repair the `testLoginOK` test case in Fig. 4, adding a new Java command that calls the method offered by the new page object.

B. Cost Benefit Analysis

We expect C&R test cases have a lower initial development cost than programmable test cases, while this relationship is reverted during test case evolution. Fig. 5 shows an example of such evolution costs. In the long run (after 5 releases, in Fig. 5), the lower maintenance cost of programmable test cases pays off, since the cumulative test case development and evolution cost becomes lower for programmable than for C&R test cases. The key point is to determine when such benefits are expected to occur, i.e., to estimate the value of n, the release number after which programmable test cases start to be cumulatively more convenient than C&R ones.

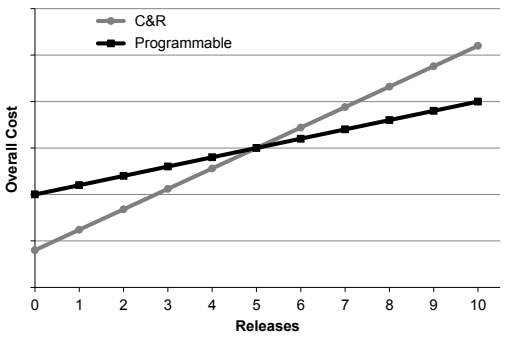


Fig. 5. Evolution costs for C&R and programmable test cases

Let us indicate by C_0 and P_0 the effort required for the initial development of C&R and programmable test cases, respectively, while $C_1, C_2, \ldots$ and $P_1, P_2, \ldots$ indicate the test case evolution costs associated with the successive software releases, numbered $1, 2, \ldots$ We are seeking the lowest value n such that:

$$\sum_{i=0}^{n} C_i \geq \sum_{i=0}^{n} P_i \tag{1}$$

Under the assumption that $C_i = C' \quad \forall i > 0$ and that $P_i = P' \quad \forall i > 0$, i.e., approximately the same test case evolution effort is required for the software releases following the initial one, either with C&R or with programmable test cases, we can find the following solution to the equation above:

$$n = \left\lfloor \frac{P_0 - C_0}{C' - P'} \right\rfloor \tag{2}$$

After more than n releases, the cumulative cost of initial development and evolution of programmable test cases is lower than that of C&R test cases.

IV. Empirical Assessment

This section reports the design, objects, research questions, metrics, procedure, results, discussion and threats to validity of the empirical study conducted to compare C&R vs. programmable web testing. We follow the guidelines by Wohlin et al. [15] on design and reporting of empirical studies in software engineering.

A. Study Design

The *goal* of this study is to investigate the cost/benefit tradeoff of C&R vs. programmable test cases for web applications, with the purpose of assessing both the short term and long term (i.e., across multiple releases) effort involved in the two scenarios. The *cost/benefit focus* regards the effort required

for the creation of the initial test suites from scratch, as well as the effort required for their evolution across the successive releases of the software. The results of this study are interpreted according to two *perspectives*: (1) *developers* and *project managers*, interested in data about the costs and the returns of the investment associated with either C&R or programmable web testing; (2) *researchers*, interested in empirical data about the impact of different approaches to web testing. The *context* of the study is defined as follows: the *human subjects* are two Junior developers facing web testing, while the *software objects* are six open source web applications under test.

B. Software Objects

We randomly selected and collected six open-source web applications from *SourceForge.net*. We have included only applications that: (1) are quite recent – thus they can work without problems on the latest versions of Apache, PHP and MySQL (the technologies we have chosen[4]); (2) are well-known and used – some of them have been downloaded more than one hundred thousand times last year; (3) have at least two major releases (we have excluded minor releases because probably with small differences between the applications the majority of the produced test cases would work without problems, and there is no reason to believe that the direction of the results would be different although reduced in magnitude); (4) belong to different application domains. Table I reports some information about the selected applications. We can see that all of them are quite recent (ranging from 2009 to 2013). On the contrary, they are considerably different in terms of number of source files (ranging from 46 to 840) 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). The difference in lines of code (columns 4 and 8) gives a rough idea of how different the two chosen releases are. In the following, we report a short description for each of the selected applications.

MantisBT[5] is a web based bug tracking system. Over time it has matured and gained a lot of popularity, and now it has become one of the most popular open source bug tracking systems. MantisBT is developed in PHP, with support for multiple database back ends.

PHP Password Manager (PPMA)[6] is a web based password manager. Each password is (DES-)encrypted with an individual user password. It is based on the Yii Framework[7].

Claroline[8] is an Open Source software based on PHP/MySQL. It is a collaborative learning environment allowing teachers or education institutions to create and administer courses through the web. The system provides group management, forums, document repositories, calendar, chat, assignment areas, links, user profile administration on a single and highly integrated package.

TABLE I. OBJECTS: WEB APPLICATIONS FROM *SourceForge.net*

	1st Release				2nd Release			
	Version	Date	File[a]	kLOC[b]	Version	Date	File[a]	kLOC[b]
MantisBT	1.1.8	Jun-09	492	90	1.2.0	Feb-10	733	115
PPMA[c]	0.2	Mar-11	93	4	0.3.5.1	Jan-13	108	5
Claroline	1.10.7	Dec-11	840	277	1.11.5	Feb-13	835	285
Address Book	4.0	Jun-09	46	4	8.2.5	Nov-12	239	30
MRBS	1.2.6.1	Jan-08	63	9	1.4.9	Oct-12	128	27
Collabtive	0.65	Aug-10	148	68	1.0	Mar-13	151	73

[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)

PHP Address Book[9] is a simple, web-based address and phone book, contact manager, and organizer.

Meeting Room Booking System (MRBS)[10] is a system for multi-site booking of meeting rooms. Rooms are grouped by building/area and shown in a side-by-side view. Although the goal was initially to book rooms, MRBS can also be used to book any resource.

Collabtive[11] is a web based collaboration software. The software enables the members of geographically scattered teams to collaboratively work.

C. Research Questions and Metrics

To achieve the goal of this study, we formulate and address the following research questions:

RQ1: What is the initial development effort for the creation of C&R vs. programmable test suites?

RQ2: What is the effort involved in the evolution of C&R vs. programmable test suites when a new release of the software is produced?

RQ3: Is there a point in time when the programmable test suites become convenient with respect to the C&R test suites?

The first research question deals with the increased development cost that is expected to occur when programmable web tests are created. We want to verify that indeed there is an increased cost when programmable web tests are developed and, more importantly, we want to estimate the ratio between the two costs. This would give developers and project managers a precise idea of the initial investment (excluded tester training) to be made if programmable test suites are adopted, as compared to C&R test suites. The metrics used to answer research question RQ1 is the ratio between initial development effort of programmable test suites over the effort required by C&R test suite. Effort is measured by the time developers spent in creating the test suites.

The second research question involves a software evolution scenario. We consider the next (major) release of the web applications under test and we evolve the test suites so as to make them applicable to the new software release. The test case evolution effort for C&R and for programmable test suites is measured as the time developers spend to update the test suites to the new software version. The ratio between the two effort measures gives a precise, quantitative indication of the

[4]Since Selenium IDE and WebDriver implement a black-box approach, the server side technologies used by the applications do not affect the results of this study, hence considering only PHP web applications is not expected to bias the results in any way.

[5]http://sourceforge.net/projects/mantisbt/

[6]http://sourceforge.net/projects/ppma/

[7]http://www.yiiframework.com/

[8]http://sourceforge.net/projects/claroline/

[9]http://sourceforge.net/projects/php-addressbook/

[10]http://sourceforge.net/projects/mrbs/

[11]http://sourceforge.net/projects/collabtive/

benefits provided by the more maintainable test suites. We expect the programmable test suites to be more maintainable than the C&R ones, hence this research question is about the cost saving achieved in the long term, when programmable test cases are expected to be more easily evolved than C&R test cases.

The last research question is about the return on the initial investment. Assuming that programmable test cases are initially more expensive to create than C&R test cases, but are also more easily maintained when the software evolves, we estimate the number of releases after which C&R testing costs become higher than programmable testing costs (see Fig. 5). To obtain such estimate, we apply Equation (2).

D. Experimental Procedure

The experiment has been performed as follows:

– Six open-source web applications have been selected from *SourceForge.net* as explained in Section IV-B.

– For each selected application, two equivalent test suites (Selenium IDE and Selenium WebDriver) have been built by the first two authors of this paper working in pair-programming and adopting a systematic approach. They can be considered junior developers: one is a PhD student and the other one is a master student with 1-year industrial experience as software tester [7]. We measured the development effort for the creation of the two test suites as clock time (in minutes). To balance as much as possible learning effects, we decided to alternate the order of test suite production and repairment

To produce the test cases we followed a systematic approach consisting of the following two steps: (1) we discovered the main functionalities from the available documentation of the applications; (2) we covered each discovered functionality with at least one test case (we assigned a meaningful name to it, so as to keep the mapping between test cases and functionalities). For both approaches, we followed the proposed best practices[12], i.e., for Selenium IDE we used the suggested locators and took advantage (when appropriate) of the suggested assertions; for Selenium WebDriver we used the page object pattern and the ID locators when possible (i.e., when HTML tags are provided with IDs), otherwise Name, LinkText, CSS and XPath locators were used. The two test suites are equivalent because the included test cases are developed to test exactly the same functionalities using the same locators, the same sequences of actions and the same input data. Column 1 of Table II reports the number of test cases in both test suites.

– Each test suite has been executed against the second release of the Web applications. First, we recorded the failed test cases and then, in a second phase, we repaired them. We measured the repair effort as clock time (in minutes).

– The results obtained for each test suite are compared (Selenium IDE vs. Selenium WebDriver), with the purpose of answering our research questions. On the results, we conduct both a quantitative analysis (e.g., computing n as reported in Equation (2)) and a qualitative analysis (e.g., reporting the lessons learned during test suite evolution, the specific traits of the testing techniques when applied to each Web application, the advantages of the page object pattern, etc.).

[12]see http://seleniumhq.org/projects/ide/ and http://seleniumhq.org/projects/webdriver/

TABLE II. TEST SUITES DEVELOPMENT

	Number of Test Cases	Time[a]			Code					
		IDE	Web Driver	p-value	IDE		WebDriver			
					Sel[b]	Java[c]	Test[c]	PO[c]	Total[c]	# PO
MantisBT	41	181	383	< 0.01	536	2825	1577	1054	2631	30
PPMA	23	68	98	0.01	475	1780	867	346	1213	6
Claroline	40	161	239	< 0.01	619	2932	1564	1043	2607	22
Address Book	28	77	153	< 0.01	482	2074	1078	394	1472	7
MRBS	24	79	133	< 0.01	476	1946	949	372	1321	8
Collabtive	40	291	383	< 0.01	555	2787	1565	650	2215	8

[a] Minutes
[b] Selenese LOC
[c] Java LOC - Comment and Blank lines are not considered

E. Results

RQ1. Table II reports some general information about the developed test suites. For each application, it reports: the number of test cases composing the test suites (column 1), the time required to develop them (columns 2 and 3), the statistical difference of the two distributions test suite development time IDE and test suite development time WebDriver (column 4) computed using the Wilcoxon paired test and their size in lines of code (columns 5-9)[13]. The last column contains the number of page objects (PO) for each WebDriver test suite[13]. The development of the Selenium IDE test suites required from 68 to 291 minutes, while the Selenium WebDriver suites required from 98 to 383 minutes. In all the six cases, the development of the Selenium WebDriver test suites required more time than the Selenium IDE test suites. The first column of Table IV shows the ratio between the time required to develop a WebDriver test suite and the corresponding time for the IDE test suite. A value greater than one means that the Selenium WebDriver test suite required more development time than the corresponding Selenium IDE test suites. The extreme values are: Collabtive with 1.32 and MantisBT with 2.12. According to the Wilcoxon paired test (see column 4 of Table II), the difference in test suite development time between IDE and WebDriver is statistically significant (with $\alpha = 0.05$) for all test suites. Summarizing, with regards to the research question RQ1 we can say that, for all the considered applications, the initial development effort for the creation of C&R test suites was lower than the one for programmable test suites.

For what concerns the size of the test suites, we report in Table II (column 5) the size of the Selenium IDE test suites, ranging from 475 (PPMA) to 619 (Claroline) Selenese LOCs[2] and the number of corresponding Java lines (column 6) obtained using the "export in Java" functionality provided by the Selenium IDE tool. We can notice that, the generated Java code does not adopt the *page object* pattern that, among other benefits, helps to reduce the amount of duplicated code. The size of the Selenium WebDriver test suites, ranging from 1213 (PPMA) to 2631 (MantisBT) Java LOC, is reported in column 9 (Total) of Table II. We can notice that, in all the cases, the

[13]The number of LOC and page objects refer to the test suites built for the newer release of each application.

TABLE III. TEST SUITES REPAIRING

	IDE				WebDriver				p-value
	Time[a]	Test Repaired	Logical Changes	Structural Changes[b]	Time[a]	Test Repaired	Logical Changes	Structural Changes[b]	
MantisBT	113	33 / 41	18	67 / 479	95	32 / 41	36	29 / 106	0.05
PPMA	71	23 / 23	12	168 / 388	55	17 / 23	21	24 / 42	< 0.01
Claroline	94	40 / 40	3	129 / 535	46	20 / 40	3	30 / 126	< 0.01
Address Book	92	28 / 28	42	100 / 375	54	28 / 28	38	14 / 54	< 0.01
MRBS	120	24 / 24	32	189 / 514	72	23 / 24	23	29 / 51	< 0.01
Collabtive	114	32 / 40	1	74 / 444	79	23 / 40	1	36 / 108	0.10

[a] Minutes
[b] Number of Locators changed over the total number of Locators in the test suite

TABLE IV. EVOLUTION COSTS OF THE SELECTED APPLICATIONS

	WebDriver / IDE (Time)		n[a]	Inter-release Time	Inter-release Time * n[a]	# PO per Test Case
	Development	Repair				
MantisBT	2,12	0,84	11,22	0 Y, 8 M	8 Y, 0 M	0,73
PPMA	1,44	0,77	1,88	1 Y, 9 M	3 Y, 4 M	0,26
Claroline	1,48	0,49	1,63	1 Y, 2 M	1 Y, 10 M	0,55
Address Book	1,99	0,59	2,00	3 Y, 4 M	6 Y, 9 M	0,25
MRBS	1,68	0,60	1,13	4 Y, 8 M	5 Y, 3 M	0,33
Collabtive	1,32	0,69	2,63	2 Y, 6 M	6 Y, 7 M	0,20

[a] We have not rounded **n** to show exact values

majority of the code is devoted to the test case logics, while only a smaller part is devoted to the implementation of the page objects. Finally, it is interesting to notice that the number of page objects, with respect to the number of test cases, varies considerably depending on the application (see column 10). For instance, MantisBT required 30 page objects for its 41 test cases (that corresponds to 0.73 page objects per test case), while Collabtive required only 8 page objects for 40 test cases (i.e., 0.20 page objects per test case). We analyse in depth the reasons and the consequences of having a different number of page objects per test case in Section IV-F.

RQ2. Table III shows some information about the test suites repairing process. In detail, the table reports, for each application, the time required to repair the test suites (IDE and WebDriver), the number of repaired test cases over the total number of test cases and (last column) the statistical difference of the two distributions test suite repairing time IDE and test suite repairing time WebDriver computed using the Wilcoxon paired test. Selenium IDE test suites required from 71 to 120 minutes to be repaired, while Selenium WebDriver test suites required from 46 to 95 minutes. For all the applications: (1) the repairing time of the Selenium IDE test suites was longer than the repairing time of the Selenium WebDriver test suites; and, (2) the number of repaired Selenium IDE test cases is greater or equal to the number of repaired Selenium WebDriver test cases. The second column of Table IV shows the ratio between the time required to repair a WebDriver test suite and the time required to repair an IDE test suite. The extreme values are: Claroline (0.49) and MantisBT (0.84). According to the Wilcoxon paired test, the difference in test suite evolution time between IDE and WebDriver is statistically significant (with $\alpha = 0.05$) for all test suites except for Collabtive (see Table III). Summarizing, with regards to the research question RQ2 we can say that, for five out of six considered applications, the effort involved in the evolution of C&R test suites, when a new release of the software is produced, is greater than for programmable test suites.

Globally, in the six test suites, we have approximately the same number of modifications made to address logical changes (i.e., 108 and 122 respectively in Selenium IDE and WebDriver), but we can observe a huge difference in terms of modified locators to repair the broken test cases due to structural changes (respectively 727 out of 2735 locators changed with IDE vs. 162 out of 487 locators changed with WebDriver). In fact, adopting the *page object* pattern avoids the duplication of locators as well as the need for their repeated, consistent evolution. Moreover, we observed that for the six Selenium IDE test suites less than 2% of the 459 ID locators were broken, while roughly 12-20% of the 655 Name, 473 LinkText and 357 CSS locators had to be repaired and 60% of the 791 XPath locators (percentages are very similar for the Selenium WebDriver test suites).

RQ3. In Table IV we computed n as reported in Equation (2) (see Section III-B). In five cases out of six, the cumulative cost of the initial development and evolution of programmable test cases (i.e., using Selenium WebDriver) is lower than that of C&R test cases (i.e., using Selenium IDE) after a small number of releases (more precisely between 1 and 3, see column 3 in Table IV). In the case of MantisBT, the same results can be obtained after about 11 releases. We discuss on the reasons behind these results in Section IV-F. Summarizing, with regards to the research question RQ3 we can say that in most cases adopting a programmable approach (as Selenium WebDriver) is convenient after a small number of releases.

F. Discussion

The key findings of our empirical study indicate that, for web applications, programmable test cases are more expensive to write from scratch than C&R test cases, with a median ratio between the two costs equal to 1.58. During software evolution, test suite repair is substantially cheaper for programmable test cases than for C&R test cases, with a median ratio equal to 0.65. Such cost/benefit trade-off becomes favourable to the programmable test suites after a small number of releases (estimated using Equation (2)), the median of which is 1.94. The most important practical implication of these results is that for any software project which is expected to deliver 2 or more releases over time, programmable test cases offer an advantageous return of the initial investment. In fact, after 2 or more releases, the evolution of the test suites will be easier and will require less effort if a programmable approach (such as WebDriver) is adopted. However, specific features of a given web application might make the trade-off more or less favourable to programmable tests. In particular, the possibility to capture reusable abstractions in page objects plays a major role in reducing the test evolution effort for programmable test cases. In the following, we analyse each factor which may affect the trade-off between C&R and programmable test cases.

Summary: In general, programmable test cases are more expensive to write but easier to evolve than C&R ones, with an advantage after 2 releases (in the median case).

1) Number of Page Objects per Test Case: As seen in Section IV-E, the number of page objects per test case varies considerably among the considered applications (from 0.20 to 0.73 page objects per test case). This number gives an indication

of the degree of reuse that page objects have across test cases, and a higher reuse is of course expected to be associated with a lower maintenance effort, since reused page objects will be maintained once for all their clients. The variability observed for the objects used in our study is due to different characteristics of these applications. From this point of view, the worst case is MantisBT with 0.73 page objects per test case. This could explain the high value of *n* (i.e., 11) estimated for this application. Often, when a test case (e.g., test case: *"create a new user"*) is executed, MantisBT shows several different and function-specific pages (e.g., a page to insert the new user data, a page to confirm the user creation, etc.), with almost no page reuse when similar functionalities are exercised. Thus, in MantisBT, each test case is required to create often a test-specific page object, associated with the functionality being exercised. In this application, the low degree of reuse of web pages corresponds to the highest observed number of pages objects per test case.

Analysing in more detail the dependencies between test cases and page objects, we can notice that: (1) in the MantisBT Selenium WebDriver test suite there are few page objects (i.e., 23%) used by a large number of test cases (only 7 page objects are used by more than 10 test cases); and, (2) a lot of page objects (i.e., 56%) are used by only 1 or 2 test cases (respectively 6 and 11 page objects). This is an aspect that can reduce the benefits of adopting the *page object* pattern. For instance, in the case of MantisBT we have that 14 page objects (among 17) are used by only one or two test cases that have been repaired. In these cases, we have few advantages in terms of maintenance effort reduction from adopting the *page object* pattern, since each repair activity on a page object, done just once, affects only one or at most two test cases.

However, a high number of page objects per test case is not strictly correlated with high values of *n*. In fact, in the Claroline test suite we have 0.55 page objects per test case (a value slightly lower than MantisBT) but here the value of *n* is low (i.e., 1.63). This different result can be explained by analysing the dependencies among test cases and page objects: (1) in Claroline we have 13 page objects repaired out of a total of 20 (i.e., 65%) compared to 24 page objects repaired out of 30 of MantisBT (i.e., 80%); (2) the page objects repaired in Claroline are usually more used (on average, each of them is used by 12 test cases) than the ones of MantisBT (on average, each of them is used by 7 test cases); and, (3) in the Claroline test suite, only 4 page objects used by no more than two test cases have been repaired (in MantisBT they are 14).

Summary: The web page modularity of the web application under test affects the benefits of programmable test cases. Web applications with well modularized functionalities, implemented through reusable web pages, are associated with reusable pages objects that are maintained just once during software evolution.

2) Number of Test Cases Repaired: From Table III we can notice that in 5 cases out of 6, the number of repaired test cases is lower when Selenium WebDriver is used. At first sight, this result could appear strange since each pair of test suites (IDE and WebDriver) has been developed equivalently, using the same locators. Actually, the number of broken test cases is equal for each pair of test suites, but the number of repaired test cases is lower with Selenium WebDriver because of the adoption of the *page object* pattern. With the page object pattern an offered method can be reused more times in a test suite. Thus, a change at the level of the page object can repair more than one test case at once. Clearly, the reduction of the number of repaired test cases is correlated with the number of times a method in a page object is (re-)used.

Let us consider a specific example. In Claroline, between the two considered releases a modification of the part of the application managing the login process occurred. Since this modification involved also the attribute used to locate the user credentials submission button, all the test cases were impacted (since all of them start with the authentication). In the Selenium WebDriver Test suite we repaired only the page object offering the method `DesktopPage login(String user, String pass)`. In this way, we automatically resolved the problem for the entire test suite. On the contrary, in the Selenium IDE test suite, we had to modify all the test cases (i.e., 40 test cases). A similar problem occurred also in MRBS and PasswordManager.

Summary: Page object reuse reduces dramatically the test repair effort.

3) Mapping n to Calendar Time: In Section IV-E we have estimated the value of n. Since we know the dates of each release (see Table I), we can easily calculate the inter-release time (see Table IV). By multiplying the inter-release time by n we obtain a value expressed in years and months that approximates the time point in which the cumulative cost for using a programmable Web testing approach becomes lower than the one for adopting a C&R approach. As reported in Table IV), for the six considered applications, this value ranges from 1 year and 10 months to 8 years, with a median of 6 years. Even if these values could appear quite high, it is important to notice that they correspond to small values of n and high values of inter-release time. This is due to the fact that the selected Web applications are quite mature and consolidated, and thus, they receive major updates rarely. If a testing approach is adopted from the very beginning of a software project, when the inter-release time is low, the benefits of the programmable test cases will be observed much earlier, since a value of n equal to 2 will correspond to a relatively small calendar time. To choose the best approach between programmable and C&R web testing, the project manager should estimate the expected number of major releases that will be produced in the future and their supposed inter-release time.

Summary: The benefits of programmable test cases are maximized if they are adopted in the early stages of a software project, when the inter release time is low.

4) Other Selenium WebDriver Advantages: As already mentioned, Selenium WebDriver offers a comprehensive programming interface and so a higher flexibility as compared

to Selenium IDE. For example, it allows developers to create test cases enriched with functionalities that natively Selenium IDE does not provide, such as: conditional statements, loops, logging, exception handling, and parameterized test cases. In this experimental work we have not used these features to render fairer the comparison between the two approaches and thus having equivalent test suites, but a project manager should consider also these features when selecting between the two testing approaches. In a previous work [7], we found that these features are indeed quite useful in industrial projects.

Summary: Additional benefits of programmable test cases (e.g., parametric test cases) should be taken into account when choosing between programmable and C&R web testing.

G. Threats to Validity

The main threats to validity that affect our results are internal, construct and external validity threats [15].

Internal validity threats concern factors that may affect a dependent variable (development and repair time of the test suites and value of *n*) and were not considered in the study. One such factor is associated with the systematic approach used to produce the test cases (i.e., the chosen functional coverage criteria). Moreover, the variability involved in the selection of input data and of the used locators could have played a role. To mitigate this threat, we have applied all known good-practices in the construction of the IDE and WebDriver test suites. Finally, learning effects may have occurred during the test suites development/repairment phases even if we tried to limit them with the specific procedure described in Section IV-D.

Construct validity threats concern the relationship between theory and observation. It is possible that measuring the time spent during production and repair of the test suites, and measuring the cumulative cost of initial development and evolution of test cases, does not provide the best means to compare the two approaches. In particular, Equation (2) is based on a simple linear model, whose parameters have been estimated using only two consecutive releases. To reduce this threat, we have carefully chosen the delta between the base release and the next one to be representative of the typical changes between releases for the selected objects. Moreover, since the manual effort available for testware evolution was limited, we preferred to allocate it on a higher number of applications than on a higher number of releases of a smaller set of applications. The development/repair time of the test suites was reported on time sheets. Even if this way to measure the time dependent variable can be considered questionable, this practice is very common in the empirical software engineering community. Finally, the test suites have been developed/repaired by two authors of this paper. Thus, researcher bias is a potential threat to the validity of this study. However, it is important to highlight that, since both tools are open source and we were not involved in their development, the authors have no reasons to favour any particular approach or interest to obtain any particular result (biasing the results). To reduce this threat, we have carefully designed and followed the experimental procedure described in Section IV-D. We chose to conduct a case study with only two software testers instead of a controlled experiment with many students (which would have overcome the researcher bias threat), because employing people without experience in Web testing would have probably benefited the simpler and easier to learn approach, C&R.

Conclusion validity concerns the relationship between the treatment and the outcome. We chose to use a non-parametric test (Wilcoxon paired test) due to the size of the sample and because we could not safely assume normal distributions.

External validity threats are related to the generalization of results. The selected applications are real open source Web applications belonging to different domains. This makes the context quite realistic, despite further studies with other applications are necessary to confirm or confute the obtained results. The test suites are small but realistic and have been built using a systematic approach. In the discussion of the experimental results, we have analysed in depth the factors which may affect our findings when applied to a different context. Another threat to external validity is that the results are limited to Selenium IDE and WebDriver, and different results could be obtained with other Web testing frameworks/tools.

V. Related Works

We will focus our related work on empirical studies about test suite evolution and partially on automatic repairing of test cases.

Berner *et al.* [1] describe their experiences on testing automation, gained participating in several industrial projects. In particular, they focus on: advantages of automated testing w.r.t. manual testing and difficulties encountered by practitioners to automatize test cases. The main findings of this work can be summarized in a number of interesting lessons learned about test automation. Two of them are: (1) the number of times a test suite is executed in its life span is correlated with cost effectiveness of automated tests and this number can help in the decision between automated and manual testing, and (2) maintenance tends to have a much bigger impact on the overall cost for testing than the initial implementation of automated tests. Concerning the first lesson learnt, it is obvious that the initial cost is higher when test automation is adopted, since it is necessary to develop the automated test suite. On the contrary, with manual testing it is sufficient to prepare the test's steps. But every time the test cases are re-executed the costs are higher if this is done manually. The authors report that, generally, each test case that is expected to be executed more than ten times is a potential candidate for automation. Concerning the second lesson learnt, we completely agree with it. Indeed, our results confirm that, after a small number of releases (i.e., n), the smaller overall cost for test automation is obtained by adopting the approach that, as much as possible, minimize the maintenance cost (i.e., the programmable approach).

Collins and de Lucena [3] describe their experience in test automation during the agile development of a web application.

They built the automated test suite using Selenium IDE. In the first phases of their project, they tried to automate the testing process as much as possible. However, as often happen at the beginning of a new project, the web pages were frequently updated because they were not able to meet users' needs. As a consequence, the test team had to re-record and re-write the test cases very often. In this way, the testing process was too time consuming, so they decided to limit the usage of automated test cases to only "stable" web pages, with the drawback of reducing the automated test suites coverage. Considering the results we obtained, we believe that the adoption of Selenium WebDriver (plus *page object* pattern) would have limited this problem since it allows to reduce the effort of repairing the test suites.

It is well-known that maintaining automated test cases is expensive and time consuming (costs are more significant for automated than for manual testing [14]), and that often test cases are discarded by software developers due to huge maintenance costs. For this reason, several researchers proposed techniques and tools for automatically repairing test cases. For instance, Mirzaaghaei *et al.* [11] presents TestCareAssistant (TcA), a tool that combining data-flow analysis and program differing, automatically repairs test compilation errors caused by changes in the declaration of method parameters. Other tools for automatically repairing GUI test cases or reducing effort during maintenance have been presented in literature [16], [6], [8]. Choudhary *et al.* [2] extend these proposals to Web applications, presenting a technique able to automatically suggest repairs for web application test cases.

VI. Conclusions and Future Work

We have conducted an empirical study to compare the costs and benefits of two alternative approaches to web testing: C&R web testing vs. programmable web testing. Results collected on six real web applications indicate that programmable tests involve higher development (between 32% and 112%) but lower maintenance effort (with a saving between 16% and 51%) than C&R tests. We have estimated the number of releases after which the maintenance cost savings overcome the initially higher development investment. According to our linear estimate, after two major releases, programmable test cases become more convenient than C&R ones. However, the actual benefits depend on specific features of the web application, including its degree of modularity, which maps to reusable page objects that need to be evolved only once, when programmable test cases are used. Another relevant factor is the time when programmable test cases are adopted, with early adoption associated with the highest benefits. It should be finally noticed that the benefits we measured do not include useful features of programmable test cases, such as the possibility to define parametric and repeated test scenarios, which might further amplify the advantages of programmable test cases.

In our future work, we plan to investigate the possibility to join the advantages of C&R web tests (ease of initial development) with those of programmable test cases (ease of evolution). On the one hand, we will reduce the effort for the definition of the page objects, by trying to generate them automatically. On the other hand, we will refactor the C&R test cases so as to make them similar to the programmable ones by, e.g., introducing the use of page objects in the interactions with the web application under test. We plan also to replicate our study on visual web testing tools based on image recognition (in particular, Sikuli Script[14] and Sikuli API[15], respectively as C&R and programmable tool). Finally, we intend to replicate our empirical study as a controlled experiment with professional developers, in order to confirm (or confute) the obtained results.

References

[1] S. Berner, R. Weber, and R. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE 2005, pages 571–579. IEEE, 2005.

[2] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, pages 24–29. ACM, 2011.

[3] E. Collins and V. de Lucena. Software test automation practices in agile development environment: An industry experience report. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST 2012, pages 57–63. IEEE, 2012.

[4] G. A. Di Lucca, A. R. Fasolino, F. Faralli, and U. de Carlini. Testing web applications. In *Proceedings of the 18th International Conference on Software Maintenance*, ICSM 2002, pages 310–319, 2002.

[5] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering (TSE)*, 31(3):187–202, 2005.

[6] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE 2009, pages 408–418. IEEE, 2009.

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

[8] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):4:1–4:36, Nov. 2008.

[9] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE 2009, pages 210–220, 2009.

[10] M. Mirzaaghaei. Automatic test suite evolution. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering*, ESEC/FSE 2011, pages 396–399. ACM, 2011.

[11] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *Proceedings of the 26th International Conference on Software Maintenance*, ICSM 2010, pages 1–5. IEEE, 2010.

[12] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE 2001, pages 25–34, 2001.

[13] F. Ricca and P. Tonella. Detecting anomaly and failure in web applications. *IEEE MultiMedia*, 13(2):44–51, 2006.

[14] M. Skoglund and P. Runeson. A case study on regression test suite maintenance in system evolution. In *Proceedings of the 20th International Conference on Software Maintenance*, ICSM 2004, pages 438–442. IEEE, 2004.

[15] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[16] Q. Xie, M. Grechanik, and C. Fu. Rest: A tool for reducing effort in script-based testing. In *24th International Conference on Software Maintenance*, ICSM 2008, pages 468–469. IEEE, 2008.

[14] http://www.sikuli.org/

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