Prevalence and Maintenance of Automated Functional Tests for Web Applications

Laurent Christophe*, Reinout Stevens*, Coen De Roover†*, Wolfgang De Meuter*
*Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
†Software Engineering Laboratory, Osaka University, Osaka, Japan
Email: lachrist—resteven—cderoove—wdmeuter@vub.ac.be

***Abstract*—Functional testing requires executing particular sequences of user actions. Test automation tools enable scripting user actions such that they can be repeated more easily. SELENIUM, for instance, enables testing web applications through scripts that interact with a web browser and assert properties about its observable state. However, little is known about how common such tests are in practice. We therefore present a cross-sectional quantitative study of the prevalence of SELENIUM-based tests among open-source web applications, and of the extent to which such tests are used within individual applications. Automating functional tests also brings about the problem of maintaining test scripts. As the system under test evolves, its test scripts are bound to break. Even less is known about the way test scripts change over time. We therefore also present a longitudinal quantitative study of whether and for how long test scripts are maintained, as well as a longitudinal qualitative study of the kind of changes they undergo. To the former's end, we propose two new metrics based on whether a commit to the application's version repository touches a test file. To the latter's end, we propose to categorize the changes within each commit based on the elements of the test upon which they operate. As such, we are able to identify the elements of a test that are most prone to change.**

I. INTRODUCTION

Testing is of vital importance in software engineering [22]. Of particular interest is functional GUI testing in which an application's user interface is exercised along requirement scenarios. Functional GUI testing has recently seen the arrival of test automation tools such as HP Quick Test Pro, SWT-BOT[1], ROBOTIUM[2] and SELENIUM[3]. These tools execute so-called *test scripts* which are executable implementations of the traditional requirements scenarios. Test scripts consist of commands that simulate the user's interactions with the GUI (e.g., button clicks and key presses) and of assertions that compare the observed state of the GUI (e.g., the contents of its text fields) with the expected one.

Although test automation allows repeating tests more frequently, it also brings about the problem of *maintaining test scripts*: as the system under test (SUT) evolves, its test scripts are bound to break. Assertions may start flagging correct behavior and commands may start timing out thus precluding the test from being executed at all. A study on Adobe's Acrobat Reader found that 74% of its test scripts get broken between two successive releases [20]. Worse, even the simplest individual GUI change can cause defects in 30% - 70% of a system's test scripts [12]. For diagnosing and repairing these defects, test engineers have little choice but to step through the script using a debugger [3] —an activity found to cost e.g., the Accenture company $120 million per year [12].

Several automated techniques for repairing broken test scripts have therefore been explored: [19], [15], [4], [12], [3]. These techniques apply straightforward repairs such as updating the arguments to a test command (e.g., its reference to a widget) or a test assertion (e.g., its comparison value). However, little is known about the kind of repairs that developers perform in practice. Insights about manually performed repairs are therefore of vital importance to researchers in automated test repair.

In this paper, we present an empirical study on the prevalence and maintenance of automated functional tests for web applications. More specifically, we study functional tests implemented in Java that use the popular SELENIUM library to automate their interactions with the web browser. Section II will outline the specifics of this library. Our study aims to answer the following research questions:

RQ1 How prevalent are SELENIUM-based functional tests for open-source web applications? To what extent are they used within individual applications?

RQ2 Do SELENIUM-based functional tests co-evolve with the web application? For how long is such a test maintained as the application evolves over time?

RQ3 How are SELENIUM-based functional tests maintained? Which parts of a functional test are most prone to changes?

The remainder of this paper is structured as follows. Section II identifies the components of a typical automated functional test that is implemented using SELENIUM. Examples include test commands (e.g., sending keystrokes to the browser) and test assertions (e.g., comparisons against the browser's observable state). Section III seeks to answer **RQ1** through a quantitative analysis. To this end, we gather a large-scale corpus of 287 Java-based web applications with references to SELENIUM. We classify the projects in this corpus and study the extent to which they use functional tests in terms of relative code sizes. Answering the other research questions requires a corpus that is representative of projects with an extensive usage of SELENIUM. We therefore refine the large-scale corpus into a smaller high-quality corpus of 8 projects that feature many functional tests. Section IV

[1] http://eclipse.org/swtbot/
[2] https://code.google.com/p/robotium/
[3] http://docs.seleniumhq.org/

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.36

Language	Keyword	# Repositories	# Files
Java	`openqa.selenium`	4287	113435
Python	`from selenium import webdriver`	1800	6207
Ruby	`require selenium webdriver`	1503	9169
C#	`using OpenQA.Selenium`	558	14473
JavaScript	`require selenium webdriver`	237	643

TABLE I: Search hits for SELENIUM on GitHub at 9/12/2013.

answers **RQ2** through a quantitative analysis of every commit in their version repository. We complement this analysis with a discussion of illustrative "Change History Views" [23] for a selection of projects. Finally, Section IV answers **RQ3** through an automated qualitative analysis of the changes to SELENIUM-based tests within each of these commits. To this end, we categorize each change based on the test component upon which it operates.

II. SELENIUM FOR AUTOMATED FUNCTIONAL TESTING

Several open-source frameworks for automating functional tests have become available. Examples include ROBOTIUM for testing mobile Android applications, SWTBOT for testing SWT-based desktop applications, and SELENIUM for testing web applications respectively. Apart from the targeted applications, these frameworks are very similar. Each provides an API for scripting interactions with an application's user interface and asserting properties about its observable state. Our study focuses on SELENIUM because its API is representative and it has grown into an official W3C standard called WebDriver[4].

Table I depicts the popularity of SELENIUM amongst different programming languages of projects available on GitHub. This table is populated by looking for projects that contain the appropriate import statements.

The fact extractors for our experiments will scope our results to web applications and tests that are implemented in Java. Moreover, we only consider tests that use the API directly and not through some third-party framework. Such tests require an import of the form `import org.openqa.selenium` at the top of the compilation unit in which they reside. In the remainder of this paper, we will refer to such files as SELENIUM *files*. Any project that contains at least one SELENIUM file will be called a SELENIUM *project*.

A. SELENIUM *By Example: a GitHub Scraper*

Listing 1 depicts a Java program that illustrates most of the SELENIUM WebDriver API for Java. This API provides functionality for interacting with a web browser (e.g., opening a URL, clicking on a button, or filling a text field), and functionality for inspecting the website that is currently displayed (e.g., retrieving its title or a particular element of its DOM). Table II gives an overview of the typical components of a SELENIUM-based functional test. Assertions are missing from the depicted program as it is not a functional test, but rather a scraper for the GitHub website.

The program's `main` method opens a connection to the web browser on line 47. Line 54 invokes method `fetchUrl` with this connection and the URL of a GitHub page with search results. Line 32 directs the web browser to this URL. Table II refers to such navigation requests and user actions as *commands*. Assuming the page rendered correctly, line 33 retrieves all `h1` elements in its DOM. We refer to such calls as *locator* expressions. Through a similar *inspector* expressions, line 36 verifies that there is no text `"Whoa there!"` within these elements. In this way, the program checks that it has not exceeded GitHub's requests per minute threshold. It then proceeds to locate all `div` elements that include the class `code-list`. Those elements correspond to the hits shown on each search page; a matching source code fragment, and the file and repository it belongs to. Lines 24 and 25 extract this information from each element.

Driver-related	Expressions opening and closing a connection to the web browser; e.g., `new ChromeDriver()`, `driver.close()`
Locators	Expressions locating specific DOM elements; e.g., `driver.findElements(By.ByName("..."))`, `webElement.findElement(By.ById("..."))`
Inspectors	Expressions retrieving properties of a DOM element; e.g., `element.getAttribute("...")`, `element.isDisplayed()`
Commands	Navigation requests and interface interactions; e.g., `driver.get("...")`, `navigation.back()`, `element.click()`, `element.sendKeys("...")`
Demarcators	Means for demarcating actual tests and setting up or tearing down their fixtures, typically provided by a Unit Testing framework; e.g., `Test`, `BeforeClass`
Assertions	Predicates over values, typically provided by a Unit Testing framework ; e.g., `assertTrue(element.isDisplayed())`, `assertEquals(element.getText(), value)`
Exception-related	Means for handling exceptions that stem from interacting with a separate process. e.g., `StaleElementReferenceException`, `TimeOutException`
Constants	Constants specific to web pages such as identifiers and classes of DOM element.

TABLE II: Components of a SELENIUM-based functional test.

Note that our SELENIUM-based scraper is strongly coupled to the user interface of GitHub. Small changes to GitHub web pages can easily break the scraper. For instance, a change to the `"Whoa there!"` text will cause it to keep on exceeding the threshold and miss hits. Likewise, changes to the element class that labels search hits will also cause our scraper to miss hits. SELENIUM-based functional tests are prone to the same problem. The problem arises every time the interface of the system under test is changed.

III. PREVALENCE OF SELENIUM TESTS

RQ1 *How prevalent are* SELENIUM*-based functional tests for open-source web applications? To what extent are they used within individual applications?*

Table I gives an informal account of the popularity of SELENIUM among the projects that are hosted on GitHub. In this section, we provide more in-depth insights about the usage of SELENIUM within these projects.

A. *Large-Scale Corpus of Projects that use* SELENIUM

We will answer **RQ1** in an exploratory manner using a large-scale corpus of Java-based SELENIUM projects. We gather this corpus using a variant of the GitHub scraper explained above. This variant not only respects the threshold on requests per minute, but also accounts for the fact that GitHub search pages show at most 1000 results (100 pages of 10 hits). To overcome this limitation, the variant implements a sliding window that limits all search requests to files of a window-wise increasing size.

Scraping GitHub for the initial 4287 candidate repositories for our corpus (cf. Table I) took most of the week of December

[4] http://www.w3.org/TR/2013/WD-webdriver-20130117/

```java
import java.util.Iterator;
import java.util.List;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class Scraper {

  public static class RateLimitException extends Exception {
    public RateLimitException(String msg) {
      super(msg);
    }
  }

  private static void printHits(WebDriver driver) {
    String xpath = "//div[@class='code-list']/*";
    List<WebElement> hits = driver.findElements(By.xpath(xpath));
    Iterator<WebElement> iter = hits.iterator();
    while(iter.hasNext()) {
      xpath = "./p[@class='title']/a";
      List<WebElement> links = iter.next().findElements(By.xpath(xpath));
      String repositoryURL = links.get(0).getAttribute("href");
      String fileURL = links.get(1).getAttribute("href");
      System.out.println(repositoryURL + " " + fileURL);
    }
  }

  private static void fetchURL(WebDriver driver, String url)
      throws RateLimitException {
    driver.get(url);
    List<WebElement> h1s = driver.findElements(By.xpath("//h1"));
    Iterator<WebElement> iter = h1s.iterator();
    while(iter.hasNext()) {
      if (iter.next().getText().contains("Whoa there!")) {
        throw new RateLimitException("GitHub rate Exceeded!");
      }
    }
  }

  public static void main(String[] args)
      throws RateLimitException, InterruptedException {
    int minSize = 1000;
    int maxSize = 1050;
    long delay = 5000;
    WebDriver driver = new FirefoxDriver();
    String url = "https://github.com/search?ref=searchresults";
    url = url + "&type=Code";
    url = url + "&l=java";
    url = url + "&q=\"import org.openqa.selenium\"";
    url = url + "+size:" + minSize + ".." + maxSize;
    for (int i = 1; i <= 100; i++) {
      fetchURL(driver, url + "&p=" + i);
      Thread.sleep(delay);
      printHits(driver);
    }
    driver.quit();
  }
}
```

Listing 1: Java program for scraping GitHub using Selenium.

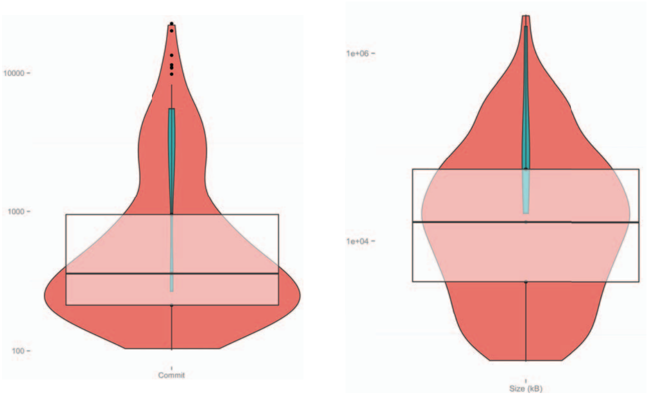

Fig. 1: Distribution of the number of commits in (left) and total size in kilobytes (right) of each repository in the large-scale corpus (red) and high-quality corpus (green).

9, 2013. From these candidate repositories, we systematically selected repositories that: (i) were created before 2013, (ii) have over 100 commits in the last year, and (iii) are larger than 500 KBytes. We assume that these criteria result in a selection of relatively mature project repositories. Only 287 of the initial 4287 candidate project repositories satisfy all criteria. These 287 repositories constitute our *large-scale corpus* of SELENIUM projects.

Figure 1 and the following table summarize our large-scale corpus in terms of the distribution of the number of commits in and the total size in kilobytes of each git repository:

	Min	1st Quartile	Median	3rd Quartile	Max
Commit	101	211	358	937	22714
Size	512	3545	15884	57831	2584284

A logarithmic scale is used because the values for these metrics vary greatly. While the `xwiki-platform` and `neo4j` repositories have 22714 and 20139 commits respectively (i.e., the top-most data points on the left violin plot), there are several repositories with barely 100 commits in total (i.e., the data points at the bottom of the left violin plot). Note that the subset of the large-scale corpus that corresponds to our *high-quality corpus* (cf. Section IV-A) is depicted in green. We use this corpus to answer the other research questions.

B. Research Method

As illustrated by our GitHub crawler, SELENIUM can be used for other purposes than automating functional tests. We therefore manually inspect and categorize the repositories in the large-scale corpus as follows:

APP Web service providers (e.g., `tedeling/ehour`).
FMW Framework for building web applications (e.g., `juzu/juzu`).
EXT SELENIUM extensions, mostly testing frameworks (e.g., `FluentLenium/FluentLenium`) or web crawlers.
EXP Web application examples for demonstration or learning purposes (e.g., `photon-infotech/php-blog`, `tomcz/example-webapp`).
MISC Unclassified projects (e.g., `xmx0632/deliciousfruit`).

APP projects use SELENIUM for automating functional GUI tests. FMW projects use SELENIUM more for unit testing purposes, verifying that the framework generates mock web applications correctly. EXP projects correspond to small experiments involving SELENIUM for demonstration or learning purposes. EXT projects are testing frameworks, crawlers or code coverage analyzers that build upon SELENIUM. MISC projects defeat this classification due to insufficient or non-English documentation.

Next, we perform an exploratory analysis of extent-related metrics of SELENIUM usage in our large-scale corpus. Concretely, we checked out a snapshot of the "main" branch of each repository at December, 13 2013 and compute the following metrics:

Files Number of `.java` files in the last snapshot.
LoC Number of lines in every `.java` file, after removing comments and empty lines.
Sel Files Number of `.java` files with an import containing the string ".selenium.".
Sel LoC Number of lines in every SELENIUM file, after removing comments and empty lines.

C. Results

Table III summarizes the corpus-wide results for each extent-related metric. The largest category of projects are the web frameworks FMW (28%), closely followed by actual web applications APP (26%). As expected, EXT projects extending SELENIUM have the largest number of SELENIUM files and highest amount of SELENIUM lines-of-code. EXP is the category with the least of them. **Although 75% of web applications APP have fewer than 19 SELENIUM files, the scatterplots in Figure 2 reveal that several use SELENIUM to a much larger extent.**

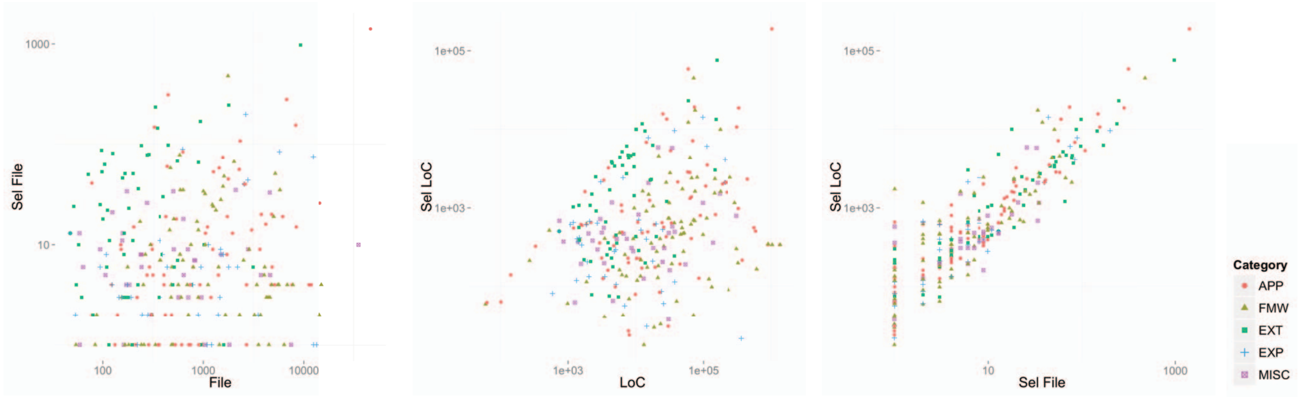

Fig. 2: Relations between the extent-related metrics for SELENIUM by category.

Scope	Count	Files	LoC	Sel Files	Sel Loc
APP	76 (26%)	305-804-2642	4876-19203-72275	3-6-19	229-600-2010
FMW	81 (28%)	484-973-3254	11873-31793-73840	2-4-14	167-421-1055
EXT	54 (19%)	108-196-404	3612-5843-10133	4-18-53	357-1428-4338
EXP	38 (13%)	221-903-1806	1180-5139-23831	2-4-9	134-406-674
MISC	38 (13%)	162-518-971	3012-9143-30409	3-7-10	206-384-578
Corpus	287	231-600-1781	4407-13734-49062	3-6-19	217-497-1619

TABLE III: Extent-metrics of the large-scale corpus by category. Each cell contains the quartiles Q1-Q2-Q3.

Figure 2 depicts different scatterplots involving these extent-related metrics. Again, a logarithmic scale has been used in order to show all data points. **No clear correlation can be discerned between the size of a web application (red dot) and the extent to which it uses SELENIUM in terms of test files (first plot), nor in terms of lines of test code (second plot)**. We attribute this to the difference in testing strategy adopted by web applications. Additionally, the complexity of the user interface of a web application — and hence the corresponding tests— cannot be gauged from size-related metrics alone.

Independently from the category, the right-most plot exhibits **a rather clear exponential relation between the number of SELENIUM files and the number of SELENIUM lines of code.** Its non-linear nature means that projects that include a lot of SELENIUM files tend to have larger SELENIUM files as well. For the categories APP, FMW and EXP ; a possible explanation lies in the seemingly widespread usage of the so-called "PageObjects" pattern[5]. This pattern advocates modeling the user interface with separate classes, of which the methods correspond to the services offered on each web page such as displaying the details for an order or navigating to another page. Implementing a test in terms of interactions with these services rather than commands ought to render them less brittle to low-level page changes. Projects `Zimbra/zimbra-source` and `Mifos/head` are examples of projects that contain many and very complex page classes. Here, every page class seems to model one part of the user interface in great detail. Of course, detailed user interface models add to the code required for the test logic itself.

D. Threats to Validity

Projects can demarcate their functional tests in different ways (i.e., see category "demarcators" in Table II). We have not attempted to measure how many functional tests a single SELENIUM file defines. Projects with few SELENIUM files might therefore not be using SELENIUM extensively, or be grouping all of their functional tests in but a few files. The same goes for the lines of code metric.

Some repositories consist of test files only (e.g., `Wikia/selenium-tests` and `exoplatform/ui-test`). Some projects therefore seem to version their functional tests and the system under test in separate repositories. We categorized those test-exclusive projects following their system under test. This explains the unusual concentration of SELENIUM code within some repositories.

Finally, our observations might differ for non-Java web applications and functional tests implemented using a non-Java binding for the SELENIUM WebDriver API. However, we do expect to see the same trends as the density of API calls within these tests is typically high. The same goes for tests that use the APIs of other frameworks for automating functional tests. Although intended for web applications, the WebDriver API is fairly representative.

IV. QUANTITATIVE ANALYSIS OF TEST CHANGES

RQ2 *Do SELENIUM-based functional tests co-evolve with the web application? For how long is such a test maintained as the application evolves over time?*

A. High-quality Corpus of Projects that use SELENIUM

Answering **RQ2** requires a high-quality corpus of projects that use the framework extensively. We therefore use the number of SELENIUM files as an additional selection criterion. Inspired by the descriptive statistics from the previous section, we set this criterion to a minimum of 40 for the high-quality corpus. It is satisfied by 47 of the original 287 repositories in the large-scale corpus. The high-quality corpus consists of a manually verified selection of these 47. This selection excludes web frameworks and test frameworks built on top of SELENIUM. It also excludes test-only project repositories (cf. Section III-D). As such, the high-quality corpus consists of repositories that version true web applications and their SELENIUM-based functional tests. Table IV describes the repositories in our high-quality corpus.

[5] https://code.google.com/p/selenium/wiki/PageObjects

GitHub Repository	Project	Description	# Commit	# Sel. Commit	Java LoC	Sel. LoC
gxa/atlas	Gene Expression Atlas	Portal for sharing gene expression data	2118	358	32375	5374
INCF/eeg-database	EEG/ERP portal	Portal for sharing EEG/ERP portal clinical data	854	17	68262	7158
mifos/head	MiFos	Portfolio management for microfinance institutions	7977	505	338705	18735
motech/TAMA-Web	Tama	Front office application for clinics	2358	239	62034	2815
OpenLMIS/open-lmis	OpenLMIS	Logistics management information system	4714	1153	72275	19195
xwiki/xwiki-enterprise	XWiki Enterprise	Enterprise-level wiki	688	164	28405	13506
zanata/zanata-server	Zanata	Software for translators	3430	81	111698	3509
Zimbra-Community/zimbra-sources	Zimbra	Enterprise collaboration software	377	243	1025410	189413

TABLE IV: The 8 repositories in the high-quality corpus.

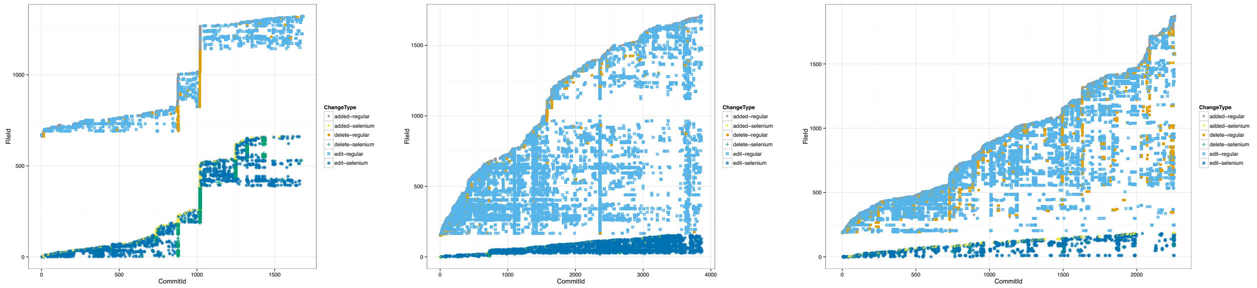

Fig. 3: Change histories of the XWIKI-ENTERPRISE (left), OPEN-LMIS (middle), and ATLAS projects (right).

B. *Project-specific Change Histories*

Before answering **RQ2** quantitatively with two new corpus-wide metrics, we provide some visual insights into the commit histories of individual projects from our high-quality corpus. Figure 3 depicts a variant of the *Change History Views* introduced by Zaidman et al. [23] for three randomly selected projects. For each commit in a project's history, our variant visualizes whether a SELENIUM (rather than a unit test file) or application file was added, removed or edited. The X-axis depicts the different commits ordered by their timestamp. The Y-axis depicts the files of the project. To this end, we assign a unique identifier to each file. We ensure that SELENIUM files get the lowest identifiers by processing them first. As a result, they are located at the bottom of the graph.

Figure 3 clearly demonstrates that **SELENIUM tests are modified as the projects evolve**. However, the modifications do not appear to happen in a coupled manner. This is to be expected as SELENIUM tests concern the user interface of an application, while application commits also affect the application logic underlying the user interface. Any evolutionary coupling will therefore be less outspoken than, for instance, between a unit test and the application logic under test.

Note that a large number of files is removed and added around revision 1000 of the `xwiki-enterprise` project. The corresponding commit message[6] reveals that this is due to a change in the project's directory structure. We see this occur in several other projects. In providing a more quantitative answer to **RQ2**, the remainder of this section will therefore take care to track files based on their content rather than their path alone.

C. *Corpus-Wide Commit Activity Metrics*

We first aim to provide quantitative insights into the *pace at which* SELENIUM-based functional tests are changed as the web application under test evolves.

1) Research method: To this end, we evaluate the high-quality corpus against several metrics that are based on the following categorization of commit activities:

Repository	$\#S$	$\overline{AS_C}$	$\overline{SS_C}$	$\overline{AS_D}$	$\overline{SS_D}$
gxa/atlas	258	6.67	1.39	1.5	0.33
INCF/eeg-database	11	75.36	1.55	98.59	2.8
mifos/head	381	19.58	1.33	6.7	0.4
motech/TAMA-Web	170	11.52	1.41	3.33	0.44
OpenLMIS/open-lmis	704	5.05	1.64	0.45	0.16
xwiki/xwiki-enterprise	96	5.33	1.71	6.94	1.95
zanata/zanata-server	51	65.65	1.59	35.82	0.72
.../zimbra-sources	66	1.74	3.73	1.81	7.09

TABLE V: Averaged commit activity metrics for the high-quality corpus. The first column denotes either $\#SS$ or $\#AS$ (cannot diverge by more than 1).

SC SELENIUM commit: commit that adds, modifies or deletes at least one SELENIUM file.

AC Application commit: commit that does not add, modify or delete any SELENIUM file.

The same categorization transposes to commit sequences:

SS SELENIUM span: maximal sequence of successive SC.

AS Application span: maximal sequence of successive AC.

Finally, this categorization enables computing the following metrics about each kind of span:

$AS_{D,C}$ Length of an AS in days and in commits.

$SS_{D,C}$ Length of a SS in days and in commits.

2) Results: The next table depicts the results for the commit activity metrics for the repositories in the high-quality corpus. The most revealing entries are related to the average length of the non-SELENIUM spans measured in commits ($\overline{AS_C}$). **It takes on average about 11.23 non-SELENIUM commits (or 4.33 days) before a commit affects a SELENIUM file.** However, this mean is largely influenced by outliers since the third quartile is even lower with only 9 non-SELENIUM commits (2.05 days). These results suggest that SELENIUM-based functional tests do co-evolve with the web application under test.

	AS_C	SS_C	AS_D	SS_D
Mean	11.23	1.59	4.33	0.66
Std Deviation	73.07	1.36	33.66	4.9
1st Quartile	2	1	0.05	0.02
Median	4	1	0.54	0.06
3rd Quartile	9	2	2.05	0.36

[6] Commit `74feec18b81dec12d9d9359f8fc793587b4ed329`

Table V scopes the same results for each project of the high-quality corpus. Repositories `gxa/atlas` and `xwiki-enterprise`, for instance, change their test scripts as often as every 6.67 and 5.33 AC respectively. Others, such as `eeg-database` and `zanata-server`, merely update their test scripts every 75.36 and 65.65 AC respectively. In fact, their fairly low number of spans $\#S$ indicates that Selenium activity and application activity does not alternate often. It turns out that `eeg-database` made little use of its test scripts and eventually removed all of them in March 2014 (i.e., after we collected our data). The stated reason is that the tests have become obsolete[7]. According to Alex Eng, a developer of `zanata-server`, Selenium tests are updated on release basis and not when new features are committed.

For all but one project in our corpus, the AS last much longer than the SS (i.e., about a 13-fold on average). This suggests that, as expected, developers spend the vast majority of their time on the web application rather than its automated functional tests. Repository `zimbra-source` forms the exception. Investigation revealed that this GitHub repository is a mirror of another Perforce one from which a cumulative commit is transferred on a daily basis [8]. Given the project's size, chances are great that such a cumulative commit affects a Selenium file.

On average, Selenium activities span up to 1.59 commits (i.e., $\overline{SS_C}$). Again, the `zimbra-source` repository is an outlier for the reasons explained above. This suggests that Selenium spans are rather short; one or two commits seem to suffice for bringing the tests up-to-date. Few supplementary commits are needed. Note that this is even the case for projects with fairly lengthy application spans. Section V investigates how scripts are maintained.

3) Threats to Validity: Extensively engineered test suites might correspond to a class hierarchy of which only the root tests import the Selenium WebDriver API. Our implementation is oblivious to maintenance efforts on such leaf tests. This might have resulted in an over-approximation of the application commits and under-approximation of the Selenium commits.

Lengths of Selenium spans, measured in commits as well as in days, are a crude substitute for information about the actual time spent and the difficulty of the maintenance efforts.

Our interpretation of the metrics assumes that the Selenium test updated by an SC is actually testing the functionality affected by a preceding AC. There is no straightforward, automated means to verify this relation. Such verification requires knowledge about which application logic is invoked by each scripted user interaction. As functional tests aim to exercise an interface along extensive scenarios, a lot of application logic is executed at once.

Finally, our findings are once again specific to the Java bindings for the Selenium WebDriver API. It therefore remains to be seen whether they hold for other automated functional tests (cf. Section III-D).

Repository	$\#B$	$\#D$	$\overline{Sv_{Day}}$	$\overline{Sv_{Mod}}$	$\overline{Sv_{AC}}$	$\overline{Sv_{SC}}$
gxa/atlas	202	45	45.16	3.89	173.64	33.31
INCF/eeg-database	58	1	0.72	3	0	3
mifos/head	430	65	316.34	3.55	1113.2	76.22
motech/TAMA-Web	203	87	47.91	1.09	260.22	42.72
OpenLMIS/open-lmis	371	140	73.06	4.71	633.96	167.31
xwiki/xwiki-enterprise	306	120	183.89	1.23	127	52.85
zanata/zanata-server	217	62	653.99	2.42	1180.02	28.47
.../zimbra-sources	1740	163	198.86	0.94	49.64	118.36

TABLE VI: Averaged maintenance metrics for the high-quality corpus. $\#B$ and $\#D$ are the number of Selenium files that are born and that die over time.

D. Corpus-wide Maintenance Metrics

We now complement the above metrics about commit activities with metrics about the lifespan of individual Selenium test files. These aim to provide insights about *how long such tests are maintained before they are abandoned* (or changed beyond recognition).

1) Research Method: We represent the lifespan of a Selenium file as a sequence of similar blobs from successive commits such that: (i) the first blob corresponds to a newly added file, (ii) the next blobs correspond to successive modifications of that file (i.e., they are similar content-wise), and (iii) the last blob corresponds to the last occurrence of the file.

This relies on a definition for the similarity of blobs. We consider two blobs from successive git commits similar if the younger one contains at least 66% of the same lines of the older one. Successive blobs exceeding this threshold correspond to the death (deletion) of and the birth (addition) of a newly added file —regardless of their file name. This way, our lifespans account for test files that are changed beyond recognition.

These definitions give rise to the following maintenance metrics about a single Selenium file:

Sv_{Day} Number of days the test survived.
Sv_{Mod} Number of modifications to the test.
Sv_{AC} Number of application commits the test survived.
Sv_{SC} Number of Selenium commits the test survived.

For simplicity's sake, we do not consider blobs with an ill-defined lifespan. We therefore exclude Selenium files that were still alive when we cloned the git repositories for our high-quality corpus (cf. Section IV-A). This excludes about 80% of Selenium files.

2) Results: Table VI depicts the results for the above maintenance metrics for individual repositories in the high-quality corpus. The following table summarizes the results for the entire corpus:

	Sv_{Day}	Sv_{Mod}	Sv_{AC}	Sv_{SC}
Mean	193.29	2.36	421.76	89.31
Std Deviation	338.1	4.46	896.11	144.77
1st Quartile	9	0	20	9
Median	85.96	1	72	44
3rd Quartile	251.92	3	263	126

The most striking result is that **75% of the Selenium lifespans in our corpus include at most three modifications before the corresponding file was deleted or modified beyond recognition** (i.e., measures for Sv_{Mod}). This indicates that either user interfaces or the tested interactions with them evolve drastically. The Selenium files in project `open-lmis` survive the longest with an average of 4.71 modifications.

[7] Commit `c723e9a373b9374d71c05ec3e605ba469d5903e8`

[8] http://www.zimbra.com/forums/developers/56176-zimbra-sourcecode-mirror-browseable-repository.html

The measures Sv_{Day}, Sv_{AC} and Sv_{SC} vary greatly and are very much project-dependent. Indications are the disparities among the project-scoped averages as well as the large corpus-wide standard deviations. Projects `eeg-database` and `zanata-server` have exceptionally high values for $\overline{Sv}_{AC}$, corresponding to very infrequent updates to SELENIUM files. The data for `eeg-database` is meaningless as its commit history includes only one death of a SELENIUM file (i.e., all but one files were still alive at the end). Assessing this project's maintenance activities therefore requires a more refined metric. Possible hypotheses for the longevity of SELENIUM files in `zanata-server` include that the project has a static user interface that does not change often, or that the project's SELENIUM files are no longer in sync with the user interface but still kept in the repository.

According to Table V, the other six projects update their test scripts on much shorter and comparable intervals. Yet we observe great differences: SELENIUM files from `mifos` survive 1113 application commits on average, while SELENIUM files from `gxa/atlas` do so for only 173 application commits. We attribute such disparities to differences in maturity between these projects. `mifos` is very large and mature (as of May 2014: 12000+ commits, 38 releases and 55 contributors), while `gxa/atlas` is much smaller (as of May 2014: 3600 commits, 4 branches and 6 contributors).

3) Threats to Validity: Our similarity threshold of 66% is somewhat more relaxed than the default 80% used by git to detect renamed files. This is because a manual investigation revealed the 80% threshold to be too strict for detecting obviously related SELENIUM blobs. Extending a small test file with a new case easily exceeds the maximum difference allowed by this threshold. We settled on 66% through experimentation.

Discarding all SELENIUM lifespans that were not yet completed at the time we cloned the repository removes about 80% of the total lifespans started in a commit history. These files are likely among the most robust and oldest. As such, our maintenance metrics are somewhat biased towards younger and more fragile SELENIUM files. To better understand this bias, we also produced the same statistics for all the SELENIUM lifespans in our commit histories. We observed a slight increase for Sv_{Day}, Sv_{AC} and Sv_{SC}. However, the measures for Sv_{Mod} remained the same, which only strengthens our most outspoken observation.

The external threats to validity for the "commit activity" metrics also hold for this experiment (cf. Section IV-C3).

V. QUALITATIVE ANALYSIS OF TEST CHANGES

RQ3 *How are* SELENIUM*-based functional tests maintained? Which parts of a functional test are most prone to changes?*

A. Research Method

We answer **RQ3** with a qualitative study of the changes that SELENIUM-based functional tests undergo throughout the evolution of the system under test. The repositories from the high-quality corpus (cf. Section IV-A) serve as subjects. More concretely, we will compute all changes in their commit history and categorize each change according to the component of the test upon which it operates. Table II lists the test component categories that form the basis for this categorization.

For our study, changes correspond to tree edit operations on the Abstract Syntax Tree (AST) of a SELENIUM-importing source file. A child of a node can either be a primitive value (such as a string), a single node or a sequence of nodes. We discern the following tree edit operations:

Insert A new node is inserted into a sequence of node children.
Update A single child or primitive value of a node is updated to a different node or value.
Move A node is moved to a different location in the AST.
Delete A node is removed from the AST.

Combining these changes with the test component categories of Section II-A results in a component-wise categorization of the changes in a commit. For instance, one change can *update* the argument of a *locator* expression. Another change can *insert* a new test *demarcator*. Yet another change may *delete* a test *command*.

We compute for each project in the high-quality corpus the total number of changes made to SELENIUM files throughout its history, as well as how frequently a change is classified as affecting a specific test component category. Combining both provides us insights into which components of a SELENIUM file are most prone to change.

B. Automating the Categorization of Test Changes

The long commit histories within the high-quality corpus call for a form of automatization. To this end, we extend the general-purpose history querying tool QWALKEKO [21] with support for the aforementioned tree edit operations. Before detailing this new extension called CHANGENODES, we recapitulate the pre-existing EKEKO and QWAL components.

1) EKEKO*, a Program Query Language:* EKEKO [6] is a tool for answering program queries about Java programs such as "where does my code implement a double dispatching idiom?". Its specification language is based on the CORE.LOGIC port to Clojure of KANREN [10]. Source code characteristics are therefore specified as logic conditions. Queries are launched using the `ekeko*` special form which takes a vector of logic variables as its first argument, followed by a sequence of logic conditions:

```
(ekeko* [?s ?e]
  (ast :ReturnStatement ?s) (has :expression ?s ?e))
```

Solutions to a query consist of bindings for its variables such that all conditions succeed. For the above query, the solutions consist of a return statement `?s` and an expression `?e` such that the latter is the former's expression part. Binary predicate `ast/2` quantifies over AST nodes of a particular type, while ternary predicate `has/3` quantifies over the values of their properties. In addition to such AST-related predicates, Ekeko provides predicates that quantify over the structure, control flow and data flow of a Java program.

2) QWAL*, a Graph Query Language:* QWAL enables querying graphs using regular path expressions [5]. Regular path expressions are an intuitive formalism for quantifying over the paths through a graph. They are akin to regular expressions, except that they consist of logic conditions to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their conditions holds.

In the context of QWALKEKO, graphs represent a program's history. Nodes correspond to program versions, while edges connect consecutive versions. Applied to such a version graph, QWAL's regular path expressions match sequences of successive versions. The logic conditions within such an expression specify source code characteristics of a single version through EKEKO predicates. Version characteristics can be specified through predicates provided by QWALKEKO itself.

```
1 (qwal graph start ?end [?left-cu ?right-cu ?change]
2  (in-current
3    (ast :CompilationUnit ?left-cu))
4  q=>
5  (in-current
6    (compilationunit|corresponding ?left-cu ?right-cu)
7    (change ?change ?left-cu ?right-cu)))
```

The above query quantifies over all changes `?change` between two successive versions of `?left-cu` and `?right-cu` of the same Java compilation unit. The first line configures the QWAL engine: it specifies the graph, start node and end node of the path expression, and introduces three logic variables `?left-cu`, `?right-cu` and `?change`. Note that the end node is an unground logic variable, and will be bound to the end node of the path expression. Lines 2–3 specify that there must be an AST node of the type `CompilationUnit` present in the current version, which is bound to `?left-cu`. Line 4 uses the QWAL primitive `q=>`. This primitive moves the current version to one of its successors. Other primitives are available as well, such as `q=>*`, which skips an arbitrary number of versions (i.e., including none). For each primitive a counterprimitive is available that traverses to a predecessor instead of a successor. For example `q<=` traverses to a direct predecessor of the current version. Lines 5–6 bind `?right-cu` to the corresponding compilation unit of `?left-cu`. This is done by looking at the package name and type declaration of `?left-cu`. Finally, changes between these two compilation units are computed and bound to `?change`.

3) CHANGENODES, a JDT Change Distiller: To answer **RQ3**, we extended QWALKEKO with support for reasoning about the changes between two ASTs in terms of tree edit operations. We call this extension CHANGENODES[9]. At its heart lies an algorithm by Chawathe et al. [2] that computes a minimal script of tree edit operations (cf. Section V-A) that, when applied, transforms a source AST into a target AST.

The same algorithm forms the foundation for CHANGEDISTILLER [9]. Our implementation differs in its use of language-specific nodes to represent ASTs, while CHANGEDISTILLER uses generic and language-agnostic nodes. To integrate with QWALKEKO, CHANGENODES uses the nodes provided by the Eclipse JDT for Java programs. We conjecture that this difference results in a better, shorter output of change operations compared to CHANGEDISTILLER due to a better matching of nodes.

4) Classifying Changes using the Extended QWALKEKO: Figure 4 depicts our query to classify changes. It detects which SELENIUM files have been modified between two revisions through predicates `fileinfo|edit/1` and `fileinfo|selenium/1`. These predicates make use of metadata that is stored for each project in QWALKEKO. Next, the query binds the compilation unit of the changed SELENIUM file to the variable `?right-cu`. We need to classify the changes made to this compilation unit. To this end, we use `q<=` to move the current version to one of its predecessors. In this version we retrieve the corresponding compilation unit, compute the changes between both units and pass this change to the predicate `classify-change/2`. This predicate succeeds when the change can be classified in a particular category.

```
(ekeko* [?change ?info ?end ?type]
  (qwal graph version ?end [?left-cu ?right-cu]
    (in-current
      (fileinfo|edit ?info)
      (fileinfo|selenium ?info)
      (fileinfo|compilationunit ?info ?right-cu))
    q<=
    (in-current
      (compilationunit|corresponding ?right-cu ?left-cu)
      (change ?change ?left-cu ?right-cu)
      (classify-change ?change ?category))))
```

Fig. 4: Classifying the changes between two SELENIUM scripts

The classification of one particular change is done using the surrounding context of the node affected by the change. For example, a change is classified as a modification to an assert statement if that change occurs within an invocation of a method that starts with "assert". To this end, the predicate `change|affects-node/2`, depicted in figure 5, takes a bound change and binds `?node` to any node that is affected by that change. An affected node is any parent node of the changed node in both the source and the target AST of the change. This predicate is used by `change|affects-assert/2`, which succeeds if a change affects an assert statement. Detecting whether an AST node is an assert statement is done by the predicate `methodinvocation|assert/1`. This predicate succeeds if a node is an method invocation with the prefix "assert".

```
(defn change|affects-node
  [change ?node]
  (conde
    [(change|affects-node|original change ?node)]
    [(change|affects-node|target change ?node)]))

(defn change|affects-node|original
  [change ?node]
  (fresh [?original]
    (change|original change ?original)
    (conde
      [(== ?original ?node)]
      [(ast-parent+ ?original ?node)])))

(defn methodinvocation|assert [?x]
  (fresh [?strname]
    (ast :MethodInvocation ?x)
    (methodinvocation|named ?x ?strname)
    (string|starts-with ?strname "assert")))

(defn change|affects-assert [change ?assert]
  (all
    (change|affects-node change ?assert)
    (methodinvocation|assert ?assert)))
```

Fig. 5: Detecting whether a change affects an assert statement.

The other change classification queries are similar. For example, a change is classified as changing an element locator if the change either affects a `@FindBy` annotation or an invocation of a method named `findBy`.

C. Results

Figure 6 depicts the ratio of changes that are classified in a specific category for our high-quality corpus. The Y-axis has a box plot summarizing the number of changes that were classified in each category, divided by the total number of

[9] https://github.com/ReinoutStevens/ChangeNodes

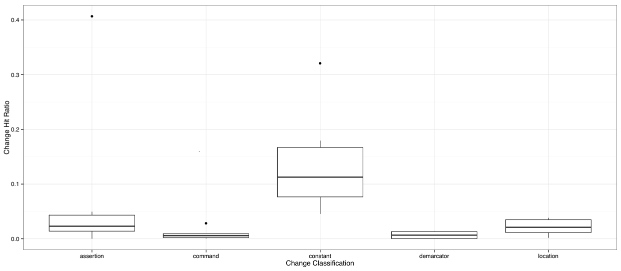

Fig. 6: Summary of the corpus-wide change classification.

Project	Total	Locator	Command	Demarcator	Asserts	Constants
Atlas	8068	90	3	104	3282	2586
XWiki	68665	115	154	24	1490	3114
Tama	31821	95	89	43	36	571
Zanata	12959	497	119	0	1	906
EEG/ERP	248	3	0	0	6	24
OpenLMIS	69792	2550	401	8	3454	8972

TABLE VII: Project-scoped change classification.

changes. Table VII lists project-scoped results. Our tool timed out on two projects of the corpus with an extensive history.

The most frequently made changes are those to constants and asserts. These are the two test components that are most prone to changes. Constants occur frequently in locator expressions to retrieve DOM elements from a web page and in assert statements as the values compared against.[10] Focusing future tool support for test maintenance on these areas might therefore benefit test engineers most. Existing work about repairing assertions in unit tests [4], and about repairing references to GUI widgets in functional tests for desktop applications [12] suggests that this is not infeasible. Note that existing work also targets repairing changes in test command sequences [15], but such repairs do not seem to occur much in practice.

Both outliers in our results stem from the ATLAS project. Its test scripts contain hardcoded genome strings inside assert statements that are frequently updated.

D. Threats to Validity

The edit script generated by CHANGENODES is not always minimal. It may incorrectly output a sequence of redundant operations for nodes that are not modified. This is due to some of the heuristics used by the differencing algorithm. These unneeded operations only form a small set of the total operations. We have performed random validations of distilled changes to ensure the correctness of our results.

Several changes are classified by looking at names of methods, without using type information. Computing this information would be too expensive to do our experiments on multiple large-scale projects. As a result some changes may be incorrectly classified.

We have been unable to find examples of some of the change categories from Section II. This is either due to our change query being too strict, the patterns not being present in the examined projects or due incorrectly distilling the changes made to the SELENIUM scripts.

VI. Related Work

Little is known about how automated functional tests are used in practice. A notable exception is Berner et al. [1]'s account of their experiences with automated testing in industrial applications. Their observation "The Capability To Run Automated Tests Diminishes If Not Used" underlines the importance of test maintenance. In interviews with experts from different organizations, an industrial survey [16] found that the main obstacles in adopting automated tests are development expenses and upkeep costs. Finally, Leotta et al. [17] recently reported on an experiment in which the authors manually created two distinct sets of SELENIUM-based automated functional tests for 6 randomly selected open-source PHP projects. While the first set of tests corresponds to the test scripts studied in this paper, the second set of tests is created using a capture-and-replay functionality of the SELENIUM IDE. The authors find that the latter are less expensive to develop in terms of time required, but much more expensive to maintain. To the best of our knowledge, ours is the first large-scale study on the prevalence and maintenance of SELENIUM-based functional tests —albeit on open-source software.

Unit tests have received more attention in the literature. The work of Zaidman et al. [23] on the co-evolution of unit tests with the application under test is seminal. Apart from "Change History Views" (cf. Section IV-B), they also proposed to plot the relative growth of test and production code over time in "Growth History Views". This enables observing whether their development proceeds synchronously or in separate phases. Fluri et al. follow a similar method in their study on co-evolution of code and comments [8]. The same goes for a study on co-evolution of database schema and application code by Goeminne et al. [11]. Our metrics from Section IV aim to provide quantitative rather than visual insights into this matter, based on commit activities rather than code growth.

Method-wise, there are several works tangential to ours in mining software repositories. Germàn and Hindle [18], for instance, classify metrics for change along the dimensions of whether the metric is aware of changes between two distinct program versions, of whether the metric is scoped to entities or commits, and of whether the metric is applied at specific events or at evenly spaced intervals. The commit activity and maintenance metrics from Section IV are scoped to commits and SELENIUM files, unaware and aware of program changes, and applied at every and specific types of commits respectively. Several techniques and metrics have been proposed for detecting co-changing files in a software repository (e.g., [13], [24]). We expect such fine-grained evolutionary couplings to be less outspoken in our setting because test scripts exercise an implemented user interface along extensive scenarios, rather than the implementation itself. More research is needed.

Section V distilled and subsequently categorized changes within each commit to a SELENIUM file. Similar analyses have been performed using the CHANGEDISTILLER [9] tool. The aforementioned study by Fluri et al. [8], for instance, includes a distribution of the types of source code changes (e.g., return type change) that induce a comment change. Another fine-grained classification of changes to Java code has also been used to better quantify evolutionary couplings of files [7]. In contrast to these general-purpose change classifications, ours is specific to automated functional tests. More coarse-grained

[10] Our tool classifies such changes also in the locator or assertion category.

change classifications have been explored as well. Hindle et al. [14], for instance, successfully categorize large commits as relating to a specific maintenance activity (e.g., adaptive or corrective) using machine learning techniques on commit messages alone.

VII. Conclusions and Future Work

This paper presents the first extensive study of the prevalence and maintenance of SELENIUM-based functional tests for web applications. Its contributions include: (i) a cross-sectional quantitative analysis of the prevalence of such tests using extent-related metrics (on a large-scale corpus of 287 open-source GitHub repositories that use SELENIUM), (ii) a longitudinal quantitative analysis of their co-evolution with the application under test, using both metrics derived from commit activities and metrics derived from the lifespan of individual tests (on a high-quality corpus of 8 projects that use SELENIUM extensively), and (iii) a longitudinal qualitative analysis of the kind of changes tests undergo (on the same corpus). To automate this analysis, we presented a significant QWALKEKO [21] extension that distills and classifies changes within each commit based on the elements of the test upon which they operate. Our corpora, tools and results are included in the paper's replication package available from `http://soft.vub.ac.be/~lachrist/icsme14/`.

As our immediate future work, we consider extending our analysis from classifying individual changes to detecting known change patterns (i.e., coordinated sets of changes) such as the addition or removal of pages to the SELENIUM-specific "PageObjects" pattern. Their detection might prove useful for maintaining the traceability between a functional test and the web application under test. Mining unknown change patterns, on the other hand, might uncover interesting avenues for researchers in test repair. In general, we believe the prevalence of automated functional testing to warrant additional research in techniques and tools that support test engineers.

Acknowledgments

This work has been supported, in part, by the *Cha-Q* SBO project of the Flemish agency for Innovation by Science and Technology (IWT), by a PhD scholarship of the same agency, by the *Cognac* project of the Research Foundation - Flanders (FWO), by the Japan Society for the Promotion of Science, Kakenhi Kiban (S), No.25220003, and by the Osaka University Program for Promoting International Joint Research.

References

[1] S. Berner, R. Weber, and R. K Keller. Observations and lessons learned from automated testing. In *Proc. of the 27th Int. Conf. on Software Engineering (ICSE05)*, 2005.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD96)*, 1996.

[3] S. Roy Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proc. of the 1st Int. Workshop on End-to-End Test Script Engineering*, 2011.

[4] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM Int. Conf. on*, 2009.

[5] O. de Moor, D. Lacey, and E. Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 2002.

[6] C. De Roover and R. Stevens. Building development tools interactively using the ekeko meta-programming library. In *Proc. of the CSMR-WCRE Software Evolution Week (CSMR-WCRE14)*, 2014.

[7] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. of the 14th Int. Conf. on Program Comprehension (ICPC06)*, 2006.

[8] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proc. of the 14th Working Conf. on Reverse Engineering (WCRE07)*, 2007.

[9] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11), 2007.

[10] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.

[11] M. Goeminne, A. Decan, and T. Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *Proc. of the CSMR-WCRE Software Evolution Week (CSMR-WCRE14)*, 2014.

[12] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proc. of the 31st Int. Conf. on Software Engineering (ICSE09)*, 2009.

[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. of the Int. Conf. on Software Maintenance (ICSM98)*, 1998.

[14] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt. Automatic classification of large changes into maintenance categories. In *Proc. of the 17th IEEE Int. Conf. on Program Comprehension (ICPC09)*, 2009.

[15] S. Huang, M. B Cohen, and A. M Memon. Repairing gui test suites using a genetic algorithm. In *Proc. of the 3rd Int. Conf. on Software Testing, Verification and Validation (ICST), 2010* , 2010.

[16] J. Kasurinen, O. Taipale, and K. Smolander. Software test automation in practice: Empirical observations. *Advances in Software Engineering*, January 2010.

[17] 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 the 20th Working Conf. on Reverse Engineering (WCRE13)*, 2013.

[18] D. M. Germán and A. Hindle. Measuring fine-grained change in software: Towards modification-aware change metrics. In *11th IEEE Int. Symp. on Software Metrics (METRICS05)*, 2005.

[19] 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, 2008.

[20] A. M Memon and M. Lou Soffa. Regression testing of guis. In *ACM SIGSOFT Software Engineering Notes*, volume 28, ACM, 2003.

[21] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers. A history querying tool and its application to detect multi-version refactorings. In *Proc. of the 17th European Conf. on Software Maintenance and Reengineering (CSMR13)*, 2013.

[22] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), 2002.

[23] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proc. of the 2008 Int. Conf. on Software Testing, Verification, and Validation (ICST08)*, 2008.

[24] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller Mining Version Histories to Guide Software Changes In *Proc. of the 26th Int. Conf. on Software Engineering (ICSE04)*, 2004.