# Web Testware Evolution

Filippo Ricca, Maurizio Leotta, Andrea Stocco, Diego Clerissi, Paolo Tonella

**Abstract:**

Web applications evolve at a very fast rate, to accommodate new functionalities, presentation styles and interaction modes. The test artefacts developed during web testing must be evolved accordingly. Among the other causes, one critical reason why test cases need maintenance during web evolution is that the locators used to uniquely identify the page elements under test may fail or may behave incorrectly. The robustness of web page locators used in test cases is thus critical to reduce the test maintenance effort. We present an algorithm that generates robust web page locators for the elements under test and we describe the design of an empirical study that we plan to execute to validate such robust locators.

**Digital Object Identifier (DOI):**

`http://dx.doi.org/10.1109/WSE.2013.6642415`

# Web Testware Evolution

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

***Abstract*—Web applications evolve at a very fast rate, to accommodate new functionalities, presentation styles and interaction modes. The test artefacts developed during web testing must be evolved accordingly. Among the other causes, one critical reason why test cases need maintenance during web evolution is that the locators used to uniquely identify the page elements under test may fail or may behave incorrectly. The robustness of web page locators used in test cases is thus critical to reduce the test maintenance effort. We present an algorithm that generates robust web page locators for the elements under test and we describe the design of an empirical study that we plan to execute to validate such robust locators.**

***Keywords—Web Testing, Test Case Evolution, Test Case Repair, Robust XPath.***

## I. Introduction

While test case generation has been investigated extensively in recent years, the problem of the evolution of web testware[1] has been somewhat neglected, despite its enormous industrial relevance. Web tests are broken quite frequently when a web application evolves and the effort involved in repairing them can be substantial. The analysis of the web testware evolution problem is in its early phases and no automated support exists to help developers dealing with it.

In order to save the input generation effort when a web application evolves to a new version, techniques have been studied to port the old input data to the new version of the code. Specifically, changes in the structure of the web application are mapped to repair actions that are applied to user session data [7]. This work has been among the first to give recognition to the importance of the web testware evolution problem. While it addresses the evolution of test data, it does not consider the problem of web page locators, that are heavily used in web testware and pose major challenges to testware evolution. After presenting the web testware evolution problem in more detail, this paper focuses on the robustness of web page locators used in testware specifically.

The paper is organized as follows: Section II provides background knowledge and describes existing works in web testing. Section III introduces testware evolution, by analysing how different kinds of changes executed on the web application under test impact on the testware maintenance activity. Section IV introduces the problem of finding robust web page locators with the goal of reducing the overall testware maintenance effort and presents our algorithm to address it. Section V sketches the design of an experimental study we plan to execute in order to evaluate the robustness of the locators created using our novel algorithm. We then present conclusions and future work in Section VI.

[1]*Testware* is an umbrella term to identify all the artefacts produced during the test process such as: documentation, scripts, inputs, expected results, files, databases and environment.

## II. Background

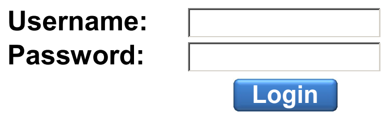
Around the year 2000, several researchers investigated the specific nature of testing when applied to web technologies and proposed methodologies, techniques, processes and tools to deal with the specific testing problems of web based systems [5], [15], [16]. These early works defined the foundations of web testing. Then, the area of web testing has expanded tremendously. Over the last 10 years, researchers have proposed several novel advanced approaches and tools, that build on top of the foundational works. The widespread diffusion of rich client languages and frameworks (e.g., ActionScript and Ajax) has posed additional challenges to Web testing.

The input data submitted through forms during the automated exploration of a web application affect to a major extent the subset of web pages that is visited (i.e., tested). The input generation problem has been dealt with by means of several, different approaches by various researchers [4], [5], [6], [7], [15], [17], [18]. In early works, most of the burden was on the testers' side and researchers expected the testers to provide input data manually or with the support of decision tables, still to be filled in manually [5], [15]. A very fruitful research direction was then followed, aimed at leveraging user-session data, so as to reuse them as test input data [6], [18].

Test adequacy criteria (e.g., coverage criteria) for web applications have been defined with reference to a *test model*. The straightforward test model for a web application is its navigation model [15], [19]. The problem with the construction of a navigation model is how to decide if a given page is already represented in the model or requires the addition of a new model element. In fact, slight variations in the content or structure of a page may be associated conceptually with the same entity in the model. This problem has been tackled by resorting to page similarity metrics [15] and to abstraction functions [19], that map a concrete web page to an abstract one.

Detailed modelling of the dynamic aspects (e.g., database access, client side scripts) can be achieved by resorting to code instrumentation [2]. Although the http protocol is stateless, web application are in reality stateful and employ various mechanisms to keep track of the current state. A finite state test model for web applications has been proposed [1], to ensure that all relevant states and state transitions involved in a web application are adequately tested.

In rich client (so-called, *web 2.0*) web applications, a substantial portion of the computation and interaction is moved to



```
<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

the client, so as to improve the user experience, making it more similar to that of fully-fledged desktop applications. Technically, this is achieved by implementing the rich client as a single-page application, whose graphical elements are updated dynamically, in response to callbacks activated asynchronously by user interactions or by server messages. These features introduce new classes of faults, as compared to traditional web applications [11].

Recent works on testing of rich client web applications [12], [13] are based on a test model which is focused on the client side states. In fact, the behaviour of these applications is determined by the DOM (Document Object Model) state and by the asynchronous events processed in each DOM state. Events trigger the execution of client side code (e.g., Javascript), which in turn can affect the client state by directly manipulating the DOM elements. DOM states are modelled as abstract states in the test model. State abstraction can be based on user defined abstraction functions [12] or on the edit distance between concrete states [13]. Ajax states are then tested by checking whether they violate any invariant (generic DOM invariants, state machine invariants and application specific invariants) [13].

## III. Testware Evolution

When a Web application evolves to accommodate requirement changes, bug fixes, or functionality extensions, existing automated test cases may become broken (e.g., they may be unable to locate some links, input fields or submission buttons), and software testers have to repair them. This is a tedious and expensive task, which is usually performed manually by software testers (automatic evolution of test suites is far from being consolidated [14]). The strategy used by a software tester to repair a test case depends mainly on two factors: (1) the kind of change (logical or structural) and (2) the approach adopted by the tool used to build the test cases (capture-replay or programmable).

### A. Kind of Changes

Test case repair activities can be categorized, depending on the kind of maintenance task that has been performed, 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 modify 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). An example of a change request that necessitates of a logical repair activity is enforcing security in a Web application by means of stronger authentication.

The second kind (structural) refers to a change at level of page layout/structure only. For example, in a login web page the id of the password textbox may be changed from `PW` (see Fig. 1) 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 (a *locator* is a mechanism for uniquely identifying an element on the web page, i.e. in the Document Object Model (DOM), for instance based on an XPath query).

### B. Approaches to Web Testing

Among the recently proposed 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 [10]. A minor change in the web application 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 is 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

```
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. 2. LoginPage and HomePage page objects

functionalities across test cases (e.g., the *page object* pattern[2] used to model the web pages involved in the test process as reusable objects), robust mechanisms to reference the elements in a web page.

Different testing tools are associated with different testware. For instance a representative of the C&R category is Selenium IDE[3]. It uses capture-replay to store and reproduce the user interactions in a test case, defined in the Selenese scripting language. Instead, Selenium WebDriver[4] is a representative of the Programmable category. It is based on the page object pattern and on programmable test cases, which can be parameterized by testers and which can reuse/share functionalities, as done for the unit testing of traditional OO software.

## C. Programmable Approach & the Page Object Pattern

Let us consider the following running example, consisting of the portion of a web application that authenticates users. In a simplified case, we have a login page (e.g., called login.asp) that requires the user to enter her credentials, i.e., *username* and *password* (see Fig. 1).

The first step for testing our simple Web application is creating two page objects LoginPage and HomePage corresponding to the web pages login.asp and homepage.asp respectively (see Fig. 2). The page object LoginPage offers a method to log into the application. That method takes in input a username and a password, inserts them in the corresponding input fields, clicks on the Login button and returns a page object of kind HomePage (because the application moves to page homepage.asp). HomePage 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.

Let us consider a simple test case, testing a successful authentication. It logs in using valid credential and it verifies that in the resulting home page the user has been actually authenticated. Fig. 3 shows the WebDriver implementation of the successful authentication test case. First, a WebDriver of

[2]http://code.google.com/p/selenium/wiki/PageObjects

[3]http://docs.seleniumhq.org/projects/ide/

[4]http://docs.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();
}
```

Fig. 3. TestLoginOK test case

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 opens the specified URL and creates a page object that instantiates LoginPage, based on the information retrieved from 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 is checked, by resorting to method getUsername().

## D. Testware Evolution Patterns

Let us consider the various testware evolution patterns, associated with different kinds of changes and testing tools:

- *C&R + 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).
- *C&R + 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 re-executing the C&R tool from the last working command.
- *Programmable + structural change.* The tester modifies one or more page objects that the broken test case links to.
- *Programmable + 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.

## IV. Robust Web Page Locators

To execute a functional test case in the context of automated testing a test script has to interact with several web page elements such as links, buttons, and input fields, and to locate them different methods can be employed.

The first proposed C&R tools (1st generation tools) simply recorded the coordinates of the web page elements directly on the screen and then used this information to find the elements during replay activities. Unfortunately, the produced test cases are extremely fragile; they might break even with small changes in the layout of the Web pages.

```
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.xpath("/html/body/form/input[2]")).sendKeys(PW);
  driver.findElement(By.linkText("Login")).click();
  return new HomePage(driver);
 }
}
```

Fig. 4. LoginPage page object (using ID, XPath and LinkText locators)

A better mechanism producing more robust test cases consists of locating the web page elements using the information contained in the Document Object Model (DOM). Selenium IDE and WebDriver (examples of 2nd generation tools) employ this approach and offer several different ways to locate the elements composing a web page. The most efficient one, according to Selenium developers[5], is localization by ID value (e.g., the password input field is located by searching for the value PW among the id values, see Fig. 1). In case the id attributes are not present, a CSS or XPath locator can be alternatively used (e.g., the password input field is located by the following XPath expression: /html/body/form/input[2]; see Fig. 1). Finally, the LinkText locator allows for the selection of a hyperlink in a web page by making use of its displayed text. Fig. 4 presents an equivalent version of the LoginPage page object, where XPath and LinkText locators are used. Moreover, Selenium WebDriver offers also other locators not considered here.

Recently, a new kind of tools (3rd generation tools) emerged using image recognition techniques to identify and control GUI components. Even if these tools can be used to interact with anything visible on the screen, specific versions focused on web application testing have been released (e.g., SikuliFirefoxDriver[6]).

While coordinates based approaches are now considered obsolete, the recent image recognition approaches are only useful in specific contexts, such as testing of complex visual applications like Google Maps, where the DOM is not available. In all the other cases, 2nd generation tools are preferred (and used in the industry) because image processing requires high computational resources and their technology is not mature enough. For this reason, we focused our attention mainly on 2nd generation tools, such as Selenium IDE and WebDriver.

In a previous work [10], we discovered that structural changes have a strong impact on test suites' maintenance. We found that in 196 test cases developed for six different open-source and heterogeneous applications, the maintenance effort due to structural changes was very high and overcame the effort due to logical changes. In the considered applications, we performed 108 modifications associated with logical changes and 727 with structural changes (727 locators out of 2735 required a fix). In a companion work [9], where an industrial Web application has been tested with Selenium WebDriver, we found that only structural changes occurred during the evolution of the application to another release. Thus, the main lesson we learnt from these works is that having a robust method to locate web page elements is essential to limit the maintenance effort.

With 2nd generation tools, different locators can be used to locate the same web page element. A key question is *how to identify the most robust locator*. On a real industrial case [8], we compared the maintenance costs of two equivalent Selenium WebDriver test suites, differing only in the used locators (similarly to the two versions of the page object LoginPage discussed above). We compared IDs vs. absolute XPaths and IDs turned out to be the most robust, even when they are auto-generated. Hence, absolute XPaths should be replaced by more robust XPath expressions when IDs cannot be used. In another previous study [10], we observed that for six test suites, less than 2% of the 459 ID locators were broken, while 60% of the 791 XPath locators required to be fixed, which again shows the need for robust XPath expressions.

In the literature on web data extraction, there are several proposals to build robust XPath expressions, for wrapping and retrieving information. However, the problem has never been investigated in the context of Web testing, where quite specific locators are needed for the web page elements under

```
$specialize("//*", e, newFifoQueue());$
proc $specialize(w : XPath, e : Element, q : Queue)$ ≡
  do
     if $(uniquelyLocate(w, e))$ then **return** $w$; fi
     $xpath1 := transf1(w);$
     if $(xpath1 \neq null)$ then $add(q, xpath1);$ fi
     $xpath2 := transf2(w);$
     if $(xpath2 \neq null)$ then $add(q, xpath2);$ fi
     $xpath3 := transf3(w);$
     if $(xpath3 \neq null)$ then $add(q, xpath3);$ fi
     $xpath4 := transf4(w);$
     if $(xpath4 \neq null)$ then $add(q, xpath4);$ fi
     $xpath := null;$
     while $(\neg empty(q) \wedge xpath = null)$
          do
             $w := getFirst(q);$
             $xpath := specialize(w, e, q);$
     od
     **return** $xpath;$
  od.
```

Fig. 5. Pseudocode to generate robust locators

[5]http://docs.seleniumhq.org/docs/03_webdriver.jsp
[6]http://code.google.com/p/sikuli-api/wiki/SikuliWebDriver

test. Starting from the approach of Dalvi *et al.* [3], we intend to apply their wrapper generation algorithm to the specific problem of page element localization in a web test case. The algorithm (see Fig. 5) generates expressions in top-down style, starting from the most general XPath expression matching all the nodes ("//*") and specializing it by applying a minimum number of transformations, until it matches only the target node. The result is a general XPath locator that identifies the target node uniquely. We expect it to be a robust locator.

More specifically, transformations work as follows:

– `transf1` converts a * to a tag name
(e.g., $//table/*/td/ \rightarrow //table/tr/td/$)

– `transf2` adds a predicate to some nodes in $w$
(e.g., $//table/*/td/ \rightarrow //table[@bgcolor = \text{'}red\text{'}]/*/td/$)

– `transf3` adds child position information to some node in $w$ (e.g., $//table/*/td/ \rightarrow //table/*/td[2]/$)

– `transf4` adds a $//*$ at the top of $w$
(e.g., $//table/*/td/ \rightarrow //*//table/*/td/$)

## V. Empirical Study Design

Goal of the study is evaluating the robustness of the XPaths created by means of the above presented algorithm. We are interested in understanding if automatically generated XPaths are able to limit the fragility problem. The results of this study will be interpreted according to two perspectives: (1) developers and project managers, interested in reducing the costs of maintaining web test suites; (2) researchers, interested in empirical data about the impact of using different locators in the context of Web testing. The context of the study is defined as follows: the human subjects of the study will be developers facing web testing, while the software objects will be heterogeneous open source web applications under test.

We plan to compare the following kinds of XPath locators (*treatments*):

- *Absolute XPath:* the element under test is identified by the complete path (from the root) in the DOM;
- *FirePath XPath:* the element under test is identified by the (possibly relative) XPath automatically generated by FirePath. FirePath is a Firebug extension that adds a development tool to edit, inspect and generate XPath expressions[7];
- *Auto-XPath:* the element under test is located by the XPath expression produced by the algorithm in Fig. 5.

We will apply the three treatments described above to publicly available web applications. We will consider two versions of each web applications and we will manually define the correct correspondence between elements under test before and after web evolution, so as to have a ground truth to be used to assess the effectiveness of the various approaches.

[7] https://code.google.com/p/firepath/

The study aims at answering the following research questions:

- **RQ1:** What kind of XPath locator is more robust when the web application under test evolves?
- **RQ2:** What kind of XPath locator has the minimum distance (in terms of number of transformation steps) between the broken locators and the correct ones?

The empirical results of the study will be collected by measuring the following metrics:

- **TP (true positives)**: elements under test that are correctly reported by locators after web evolution;
- **FP (false positives)**: elements under test that are reported by locators after web evolution but do not correspond to the elements to be located;
- **FN (false negatives)**: elements under test that should be reported by locators, but are actually no longer reported after web evolution.

We will also compute derived metrics that facilitate the comparison between different locators, such as:

- *precision* (TP/(TP+FP))
- *recall* (TP/(TP+FN))
- *F-measure* (2 * precision * recall / (precision + recall))

These metrics address directly RQ1, while for RQ2 we need an additional metrics:

- **RP**: minimum number of repair transformations, similar to the four used by the algorithm in Fig. 5, to be applied to a broken locator to fix it when the web application evolves.

With TP, FP, FN and RP we expect to be able to collect enough empirical evidence to answer research questions RQ1 and RQ2.

## VI. Conclusions and Future Work

We have discussed the importance of page element locators during web testware evolution and we have proposed an algorithm to generate robust locators. We intend to evaluate the proposed algorithm empirically, by following the experimental design described in this paper. We also plan to improve and extend the proposed algorithm based on the results of the empirical study, once available.

Other research directions that we will investigate in the near future include an empirical evaluation of the web testware maintenance costs incurred by developers when different web testing approaches and tools (e.g., Selenium IDE, Selenium WebDriver, SikuliFirefoxDriver) are adopted.

## References

[1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with fsms. *Software and System Modeling*, 4(3):326–345, 2005.

[2] G. Antoniol, M. D. Penta, and M. Zazzara. Understanding web applications through dynamic analysis. In *Proceedings of the 12th International Workshop on Program Comprehension*, IWPC 2004, pages 120–131, 2004.

[3] N. Dalvi, P. Bohannon, and F. Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD 2009, pages 335–348, New York, NY, USA, 2009. ACM.

[4] Y. Deng, P. G. Frankl, and J. Wang. Testing web database applications. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[5] 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.

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

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

[8] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In *Proceedings of the 1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013, pages 53–58. ACM, 2013.

[9] 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.

[10] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Submitted to the 19th Working Conference on Reverse Engineering*, WCRE 2013, 2013.

[11] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to ajax web applications. *International Journal of Software Tools for Technology Transfer*, 10(6):477–492, Oct. 2008.

[12] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, ICST 2008, pages 121–130, 2008.

[13] 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.

[14] 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.

[15] 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.

[16] F. Ricca and P. Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, 2002.

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

[18] S. Sampath, S. Sprenkle, E. Gibson, L. L. Pollock, and A. S. Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering (TSE)*, 33(10):643–658, 2007.

[19] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM 2009, pages 211–220, 2009.