Web Application Model Generation through Reverse Engineering and UI Pattern Inferring

Clara Sacramento
Department of Informatics Engineering,
Faculty of Engineering of the
University of Porto
Porto, Portugal
ei09090@fe.up.pt

Ana C. R. Paiva
INESC TEC, Department of Informatics Engineering,
Faculty of Engineering of the
University of Porto
Porto, portugal
apaiva@fe.up.pt

***Abstract*—A great deal of effort in model-based testing is related to the creation of the model. The model itself, while a powerful tool of abstraction, can be a source of bugs. This paper presents a dynamic reverse engineering approach that aims to extract part of the model of an existing Web application through the identification of User Interface (UI) patterns. This approach explores the Web application via reverse engineering, records information related to the interaction (interaction history, HTML pages and their URLs), analyzes the gathered information, and infers the UI patterns via a set of heuristics rules. After complemented with additional information, the model extracted is the input for the Pattern-Based GUI Testing (PBGT) approach for testing existing applications.**

***Index Terms*—Reverse Engineering, Web Application, UI Patterns, Web Scraping**

I. Introduction

Web applications are getting more and more important, and can now handle tasks that before could only be performed by desktop applications [1], like editing images or creating spreadsheet documents. However, despite their growing relevance, they still suffer from a lack of standards and conventions [2], unlike desktop and mobile applications. This means that the same task can be implemented in many different ways, which makes automated Web application testing difficult to accomplish and inhibits reuse of testing code.

Graphical User Interfaces of all kinds are populated with recurring behaviors that vary slightly, an example being authentication (*login/password*). These behaviors (patterns) are called User Interface (UI) patterns [3] and are recurring solutions that solve common design problems. Due to their widespread use, UI patterns allow users a sense of familiarity and comfort when using applications.

However, while UI patterns are familiar to users, their implementation may vary significantly. Even a simple pattern like authentication can be implemented in many ways. Authentication failure can trigger the appearance of an error message; but some implementations simply erase the inserted data, with no error message visible. Despite this, it is possible to define generic and reusable test strategies (User Interface Test Patterns - UITP) to test those patterns. This requires a configuration process, in order to adapt the tests to different applications [4].

That is the main idea behind the PBGT (*Pattern-based GUI Testing*) project, in which this research work is included. In the PBGT approach, the user builds a test model of the Web application with instantiations of UI Test Patterns, and later uses that model to test the Web app. The goal of the work described in this paper is to continue the work done in [5] on the reverse engineering process (PARADIGM-RE), and its aim is to automatize the model construction: independently/automatically explore a Web application, infer the existing UI patterns in its pages, and finally produce a model with the UI Test Patterns that define the strategies to test the UI Patterns present in the web application.

The rest of the paper is structured as follows. Section II presents an overview of the PBGT project, setting the context for this work. Section III describes the developed approach, its components and their interrelations, and the results it produces. Section IV provides a practical example of the proposed system. Section V addresses the related work, as well as the available tools to perform the needed tasks. Section VI presents the conclusions, some of the problems encountered and a perspective on future work.

II. PBGT Overview

As mentioned before, the focus of this paper is a component of a research project named PBGT (*Pattern-based GUI Testing*) [6]. The goal of this project is to develop a model-based GUI testing tool and approach, usable as an industrial tool.

A. Architecture

This project has five parts: a DSL (*Domain Specific Language*) named **PARADIGM** to define GUI testing models based on UI Test Patterns; **PARADIGM-RE**, a Web application reverse engineering tool whose purpose is to extract UI patterns from Web pages without access to their source code, and use that information to generate a test model

written in PARADIGM; a modeling and testing environment, named **PARADIGM-ME**, built to support the creation of test models; an automatic test case generation tool, named **PARADIGM-TG**, that generates test cases from test models defined in PARADIGM; and finally, a test case execution tool, named **PARADIGM-TE**, which executes test cases, analyzes their coverage, and returns detailed execution reports. The architecture and workflow of the project is shown in Fig. 1.

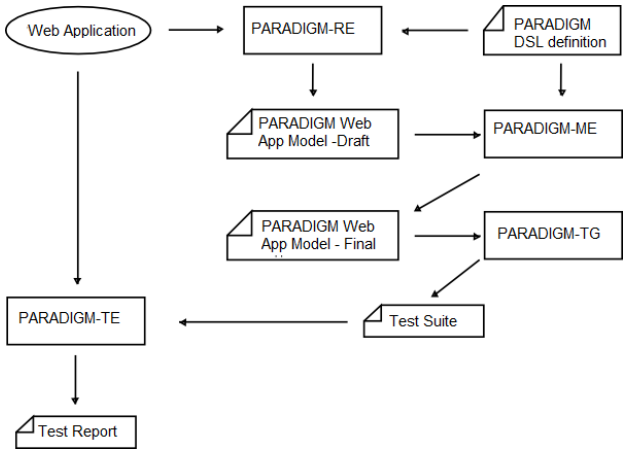

Fig. 1. An overview of the PBGT project

B. PARADIGM

The definition of the PARADIGM DSL has started by analyzing a set of existing UI Patterns [3]. This DSL contains a set of UI Test Patterns that specify generic test strategies to test those UI Patterns along their different implementations. Besides UI Test Patterns, the DSL also defines a set of connectors that allows the definition of sequences of test strategies to execute.

A UI Test Pattern defines a test strategy which is formally defined by a set of test goals (for later configuration) [6] with the form:

$$< Goal; V; A; C; P > \quad (1)$$

Goal is the ID of the test. *V* is a set of pairs [*variable*, *inputData*] relating test input data with the variables involved in the test. *A* is the sequence of actions to perform during test case execution. *C* is the set of possible checks to perform during test case execution, for example, check if it remains in the same page. *P* is a Boolean expression (precondition) defining the set of states in which it is possible to execute the test.

The UI Patterns that PARADIGM language is able to test through UI Test Patterns are:

Login
: This pattern is commonly found in Web applications, especially in the ones that restrict access to functionalities or data. Usually consists of two input fields (a normal input box for email or username, and a cyphered text for the password) and a submit button, with optionally a "remember me" checkbox. The authentication process has two possible outcomes: valid and invalid.

Search
: This pattern consists of one or more input fields, where the user inserts keywords to search, and a submit button to start the search. The search may be submitted via a submit button, or dynamically upon text insertion. When the search is successful, the website shows a list of results; upon failure, an error message may be shown.

Sort
: This pattern sorts a list of data by a common attribute (price, name, relevance, etc.) according to a defined criteria (ascending or descending, alphabetically, etc.).

Master Detail
: This pattern is present in a webpage when selecting an element from a set (called *master*) results in filtering/updating another related set (called *detail*) accordingly. For example, clicking on a checkbox associated to a brand may include (or exclude) products of that brand in a product search result list. Generally the only elements changed are the elements belonging to the *detail* set.

Menu
: This pattern is very common in webpages. It's usually defined as a tree structure with several navigational options, to provide easier access for users.

Input
: This pattern is any kind of input field that allows the user to insert text.

Call
: This pattern is any kind of element where a click triggers a change of page.

C. Produced Models

The models produced by the reverse engineering tool PARADIGM-RE consist of a XML file that contains information about the UI Test Patterns needed to test the UI Patterns found: their name and the input values for their variables. The PARADIGM model generated by the PARADIGM-RE tool does not contain the connectors between the UI Test Patterns, the checks to perform during test case execution, or the precondition determining the states where the test strategy will be run. This information needs to be complemented by the tester afterwards in order to generate test cases and execute them over a web application.

III. Reverse Engineering Approach

The approach described in this paper aims to improve on the previous work [5] done on the PARADIGM-RE tool. The previous tool extracted information from an user's interaction with the Web application under analysis, analysed the information, produced some metrics (such as the total ratio of

the LOC (*lines of code*) length of all visited pages and the ratio of two subsequent pages), and finally used those metrics and the user interaction's information to infer UI patterns via a set of heuristic rules. The information saved of an user interaction is the source code and URLs of the visited pages, and the interaction's execution trace. An execution trace is the sequence of user actions executed during the interaction with a software system, such as clicks, text inputs and also some information of the system state (e.g., the information that is being displayed). An example of an execution trace file used by the tool can be seen in Section III-B, in Table I.

The approach identified all the available patterns, but produced a high number of false positives [5], and the exploration process was done by saving the user interaction with the Web application under test. The work described in this paper aims to identify UI patterns in the Application Under Test (AUT) as well, but also removes the need for user interaction with the AUT providing a reverse engineering process to explore automatically the web application under analysis, and attempts to improve the pattern inferring process. The architecture of the tool is seen in Figure 2.

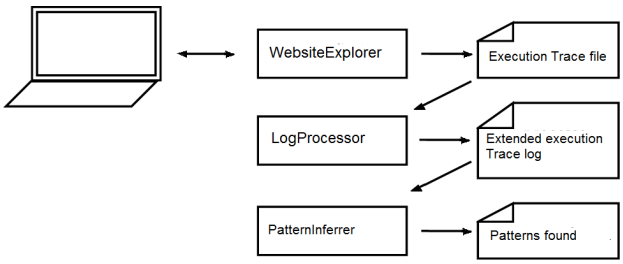

Fig. 2. The architecture of the approach.

The approach can be divided into three parts: **WebsiteExplorer**, **LogProcessor**, and **PatternInferrer**.

WebsiteExplorer interacts automatically with the AUT and produces an execution trace file with the actions taken (see Table I for an example, minus the rightmost column). **LogProcessor** is a text parser. It analyzes the previous file, parses each line, searches for important keywords, identifies the action contained in the line according to the results found, and produces an updated execution trace file (again, see Table I for an example). Finally, **PatternInferrer** analyzes the updated execution file, identifies the existent patterns, their location in the website and any existent parameters, and produces an XML file with the results.

A. AUT Exploration

The interaction process is described in a simplified manner in Algorithm 1.

Currently the elements explored are <*select*> elements, <*input*> elements, and <*a*> (link) elements. The way each element is visited is different: link elements are clicked; input elements get text (either a keyword or a number, depending on the type of input) and the containing form is submitted; and in the case of dropdown menus, a random option is selected and

```
Data: number_of_actions, number_of_iterations,
      website_base_url
Result: written execution trace file
current_action := 0; redirections := 0;
invalid_urls := parse robots.txt file;
while current_action < number_of_actions do
    list := extract valid non-visited elements;
    next_element := choose_next_element(list);
    if next_element == null then
        redirections++;
        if redirection < number_of_iterations then
            go to base URL;
            write to execution trace file;
        else
            end program;
        end
    else
        visit element;
        add element to visited elements;
        write to execution trace file;
        current_action++;
    end
end
```

Algorithm 1: Pseudo-code algorithm to explore a page.

the surrounding form is submitted. The exception to the *input* rule is when the element is identified as a *login* element, in which case all the sibling elements (elements inside the form where the selected element is located) get text and only then the form is submitted.

Information about these elements is extracted via XPath[1]. Before interacting with an element, the algorithm makes the following checks: if the element has not been visited in the current page, if the website's robots.txt[2] file allows it to be visited, and finally, if the element contains any unwelcome keywords that if explored may drive the explorer into unwanted ways. These keywords may be general to all elements (anything that edits information or contains the words 'buy'/'sell', for example, that would lead to purchase products) or element-specific (input elements cannot contain the attribute '*disabled*' or'*readonly*', if they are to be interacted with). All elements have the same probability of being chosen from a list of elements.

The execution trace file produced is a CSV (*Comma-separated values*) file with three columns: **action**, the type of action executed (*click, type*, or *select* when selecting an option in a dropdown menu – it may also contain the suffix *AndWait*, which indicates a page change); **target**, the identifier of the visited element; and **value**, which has the parameter for the action and may be empty (for example, in the case of *type* actions, it is the inserted text).

[1]XPath: www.w3schools.com/XPath/
[2]robots.txt: http://www.robotstxt.org/

B. File Processing

This component is a lexical analyzer, whose role is to examine the execution trace file line by line and identify the type of each action written therein. It has a data structure (which serves as its lexical grammar) containing the rules it is going to search for, and each has the following attributes: **pattern_name**, the identifier for the rule; **identifying_regex**, a regex (Regular Expression[3]) that identifies an action of that type; and **garbage_removal_regex**, another regular expression that serves to erase unneeded words in case the pattern is discovered.

For every line, all rules are tested, and if a rule matches the line, first a camel-case token (composed by the sum of the action type and the rule's name) is produced; and afterwards all words that match the *garbage_removal_regex* are removed. This is done because a pattern may be a superclass of another pattern, and in that case it is better to just identify the super pattern. For example, if an element is identified as part of a *search* pattern, it is not necessary to identify it as *input* as well.

The identifiable patterns by default are: *login, search, sort*, and *input*. However, the identifiable patterns can also be overridden and added to via a text file.

An example of file processing may be seen in Table I.

Action	Target	Value	Processing Return
type	id=input_username	"user1"	typeUsername
type	id=input_password	"123pass"	typePassword
clickAndWait	css=input [type="submit"]	EMPTY	clickFormSubmit PageChange
type	id=searchInput	"coffee"	typeSearch
clickAndWait	id=mw-searchButton	EMPTY	clickSearch PageChange
select	id=sort	label=Price: Low to High	selectSort

TABLE I
EXAMPLE OF AN EXECUTION TRACE FILE, AND OF PROCESSED LINES.

C. UI Pattern Inferring

This component is a syntactical analyzer, that takes as input the extended execution trace file returned by the LogProcessor, runs it against a predefined grammar, and returns the patterns found. If during the process the tool detects the same UI Pattern instance several times, the tool is capable of ignoring it or add more configurations to the one already detected. One of the existent rules is defined on Listing 1.

Listing 1. The syntactic rule for all the identifiable patterns.

$$S \rightarrow FullLogin \mid FullSearch \mid FullSort \mid FullInput \mid Call$$

$$FullSearch \rightarrow Search\ Submit\ Refine$$
$$Search \rightarrow \gamma\ typeSearch\ \gamma$$
$$\gamma \rightarrow clickSearch \mid selectSearch \mid \varepsilon$$
$$Submit \rightarrow clickFormSubmit \mid clickButton \mid \varepsilon$$
$$Refine \rightarrow clickRefineSearch \mid \varepsilon$$

$$FullLogin \rightarrow Login\ Username\ Password\ Auth\ click$$
$$Login \rightarrow clickLogin\ PG$$
$$PG \rightarrow pageChange \mid \varepsilon$$
$$Username \rightarrow typeUsername \mid typeEmail$$
$$Password \rightarrow typePassword$$
$$Auth \rightarrow typeAuth \mid \varepsilon$$

$$FullSort \rightarrow Sort\ Submit$$
$$Sort \rightarrow clickSort \mid selectSort$$

$$FullInput \rightarrow Input\ Submit$$
$$Input \rightarrow clickInput \mid typeInput$$

$$Call \rightarrow Link\ pageChange$$
$$Link \rightarrow clickLink \mid clickHome \mid clickPreviousPage \mid clickNextPage \mid clickImageLink$$

After the file is processed, an XMI file is produced containing all the pattern occurrences found, and the values assigned to each variable, as per Equation II-B. As mentioned before, the checks to be done on each variable and any preconditions must be specified by the tester later on in the PARADIGM-ME tool.

IV. EVALUATION

The RE tool was initially experimented iteratively over a number of Web applications, a training set, with the goal of refining and fine-tuning the strategies used to find UI Patterns.

After the training phase, the RE tool was used to detect UI Patterns in several public known and widely used Web applications, in an evaluation set. This time, the purpose was to evaluate the RE tool, i.e., determine which UI patterns occurrences the tool was able to detect in each application execution trace (ET) and compare them to the patterns that really exist in such trace. The results are presented in tables that show the number of instances of each UI pattern that exist in the ET, the ones that the tool correctly found and the ones that the tool mistakenly found (false positives). Five applications were chosen from the top 30 most popular Websites[4]: Amazon, Wikipedia, Ebay, Youtube and Yahoo. The numerical results were combined and can be found in Table II.

As we can see in Table II, during the case study the tool found no false positives, and found 89% of the patterns present in the ET. However, the case study doesn't mention how many patterns were present in the web applications that weren't visited, which we haven't verified. Only one path produced by the application was considered for each Web application, and the results don't consider any duplicate patterns found.

[3]Regex: http://www.regular-expressions.info/

[4]according to: en.wikipedia.org/wiki/List_of_most_popular_websites

Pattern	Present in ET	True Positive	False Negative	False Positive
Login	2	2	0	0
Search	14	11	3	0
Sort	2	1	1	0
Input	6	4	2	0
Call	61	58	3	0
Total	**85**	**76**	**9**	**0**
Total	**100%**	**89%**	**11%**	**0%**

TABLE II
EVALUATION SET RESULTS.

V. RELATED WORK

Reverse engineering is "the process of analyzing the subject system to identify the system components and interrelationships and to create representations of the system in another form or at a higher level of abstraction" [7]. There are four methods of applying reverse engineering to a system: the dynamic method, in which the data are retrieved from the system at run time without access to the source code, the static method, which obtains the data from the system source code [8], the hybrid method, which combines the two previous methods, and the historical method, which extracts information about the evolution of the system from version control systems, like SVN[5] or GIT[6] [9]. These approaches follow the same main steps: collect the data, analyze it and represent it in a legible way, and in the process allow the discovery of information about the system's control and data flow [10].

There are plenty of approaches that extract information from Web applications [11], [12], [13]. ReGUI [14], [15] is a dynamic reverse engineering tool made to reduce the effort of modeling the structure and behavior of a software application GUI. It was also developed to reduce the effort of obtaining models of the structure and behaviour of a software application's GUI, however it only works in desktop applications. Duarte, Kramer and Uchitel defined an approach for behavior model extraction which combines static and dynamic information [16].

There are also plenty of approaches that explore Web applications for analysis and processing. Ricca and Tonella's ReWeb [17] dynamically extracts information from a Web application's server logs to analyze its structure and evolution, and so aims to find inconsistencies and connectivity problems. Benedikt *et al.* introduced a framework called VeriWeb [18] that discovers and explores automatically Website execution paths that can be followed by a user in a Web application. Bernardi *et al.* [19] presents an approach for the semi-automatic recovery of user-centered conceptual models from existing web applications, where the models represents the application's contents, their organization and associations, from a user-centered perspective. Marchetto *et al.* proposed a state-based Web testing approach [20] that abstracts the Document Object Model (DOM) into a state model, and from the model derives test cases. Crawljax [21] is a tool that obtains graphical site maps by automatically crawling through a Web application. Memon presented an end-to-end model-based Web application automated testing approach [22] by consolidating previous model development work into one general event-flow model, and employs three ESESs (event space exploration strategies) for model checking, test-case generation, and test-oracle creation. Mesbah *et al.* proposed an automated technique for generating test cases with invariants from models inferred through dynamic crawling [23]. Artzi *et al.* developed a tool called Artemis [24] which performs feedback-directed random test case generation for Javascript Web applications. Artemis triggers events at random, but the events are prioritized by less covered branch coverage in previous sequences. Amalfitano *et al.* developed a semi-automatic approach [25] that uses dynamic analysis of a Web application to generate end user documentation, compliant with known standards and guidelines for software user documentation. Another approach by Mesbah *et al.*, named FeedEx [26] is a feedback-directed Web application exploration technique to derive test models. It uses a greedy algorithm to partially crawl a RIA's GUI, and the goal is that the derived test model capture different aspects of the given Web application's client-side functionality. Dallmeier *et al.*'s Webmate [27], [28] is a tool that analyzes the Web application under test, identifies all functionally different states, and is then able to navigate to each of these states at the users request.

User Interaction (UI) patterns, in particular the ones supported by the tool, are well-documented in a various number of sources [29], [3], [30], [31]. Lin and Landay's approach [32] uses UI patterns for Web applications that run on PCs and mobile phones, and prompt-and-response style voice interfaces. Pontico *et al.*'s approach [33] presents UI patterns common in eGovernment applications.

Despite the fact that there are plenty of approaches to mine patterns from Web applications, no approaches have been found that infer UI patterns from Web applications beside the work extended in this paper [5], [4]. The approaches found deal mostly with Web mining, with the goal of finding relationships between different data or finding the same data in different formats. Brin [34] presented an approach to extract relations and patterns for the same data spread through many different formats. Chang [35] proposes a similar method to discover patterns, by extracting structured data from semi-structured Web documents.

VI. CONCLUSIONS

This paper presented a dynamic reverse engineering approach to extract UI Patterns from Web applications, by extracting information from an execution trace and afterwards inferring the existing UI Patterns and their configurations from that information. The result is then exported into a PARADIGM model to be completed in PARADIGM-ME. Then, the model can be used to generate test cases that are

[5]SVN: svn.apache.org
[6]GIT: git-scm.com

performed on the Web application. This reverse engineering tool is used in the context of the PBGT project that aims to build a model based testing environment to be used by companies.

The steps followed by the approach have been explained in detail, including the components responsible for the automatic exploration of the Web application, the lexical and syntactical analysis of execution trace files, pattern discovery, and the production of the final model.

The evaluation of the overall approach was conducted in several worldwide used Web applications. The result was quite satisfactory, as the RE tool found most of the occurrences of UI patterns present in each application as well as their exact location (in which page they were found).

Despite the satisfactory results obtained, the approach still needs some improvement. The tool doesn't handle dynamic pages very well, and the website explorer component visits the same elements more than once, which leads to the discovery of the same patterns many times. As so, the features planned for future versions of the RE tool include the support for a larger set of UI patterns, the definition of more precise methods to identify patterns, based on HTML page analysis and/or manipulation of the DOM tree, and better Javascript handling.

REFERENCES

[1] J. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.

[2] L. L. Constantine and L. A. Lockwood, "Usage-centered engineering for web applications," *Software, IEEE*, vol. 19, no. 2, pp. 42–50, 2002.

[3] M. Van Welie, G. C. Van Der Veer, and A. Eliëns, "Patterns as tools for user interface design," in *Tools for Working with Guidelines*. Springer, 2001, pp. 313–324.

[4] I. C. Morgado, A. C. Paiva, J. P. Faria, and R. Camacho, "Gui reverse engineering with machine learning," in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*. IEEE, 2012, pp. 27–31.

[5] M. Nabuco, A. C. Paiva, R. Camacho, and J. P. Faria, "Inferring ui patterns with inductive logic programming," in *Information Systems and Technologies (CISTI), 2013 8th Iberian Conference on*. IEEE, 2013, pp. 1–5.

[6] R. M. Moreira, A. C. Paiva, and A. Memon, "A pattern-based approach for gui modeling and testing," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 288–297.

[7] E. J. Chikofsky, J. H. Cross *et al.*, "Reverse engineering and design recovery: A taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[8] T. Systä, "Dynamic reverse engineering of java software," in *ECOOP Workshops*, 1999, pp. 174–175.

[9] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[10] M. J. Pacione, M. Roper, and M. Wood, "A comparative evaluation of dynamic visualisation tools," in *10th Working Conference on Reverse Engineering*, 2003, pp. 80–89.

[11] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald, "Applying concept analysis to user-session-based testing of web applications," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 643–658, 2007.

[12] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Rich internet application testing using execution trace data," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. IEEE, 2010, pp. 274–283.

[13] I. Andjelkovic and C. Artho, "Trace server: A tool for storing, querying and analyzing execution traces," in *JPF Workshop, Lawrence, USA*, 2011.

[14] I. Coimbra Morgado, A. Paiva, and J. Pascoal Faria, "Reverse engineering of graphical user interfaces," in *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, 2011, pp. 293–298.

[15] I. Coimbra Morgado, A. C. Paiva, and J. Pascoal Faria, "Dynamic reverse engineering of graphical user interfaces," *International Journal On Advances in Software*, vol. 5, no. 3 and 4, pp. 224–236, 2012.

[16] L. M. Duarte, J. Kramer, and S. Uchitel, "Model extraction using context information," in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 380–394.

[17] F. Ricca and P. Tonella, "Understanding and restructuring web sites with reweb," *Multimedia, IEEE*, vol. 8, no. 2, pp. 40–51, 2001.

[18] M. Benedikt, J. Freire, and P. Godefroid, "Veriweb: Automatically testing dynamic web sites," in *In Proceedings of 11th International World Wide Web Conference (WW W2002*. Citeseer, 2002.

[19] M. L. Bernardi, G. A. Di Lucca, and D. Distante, "Reverse engineering of web applications to abstract user-centered conceptual models," in *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*. IEEE, 2008, pp. 101–110.

[20] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 121–130.

[21] D. Roest, "Automated regression testing of ajax web applications," Master's thesis, Delft University of Technology, February 2010.

[22] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[23] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 35–53, 2012.

[24] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of javascript web applications," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 571–580.

[25] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Using dynamic analysis for generating end user documentation for web 2.0 applications," in *Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on*. IEEE, 2011, pp. 11–20.

[26] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models," in *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE). IEEE Computer Society*, 2013, p. 10.

[27] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools*. ACM, 2012, pp. 11–15.

[28] ——, "Webmate: Generating test cases for web 2.0," in *Software Quality. Increasing Value in Software and Systems Development*. Springer, 2013, pp. 55–69.

[29] J. Tidwell, *Designing interfaces*. O'Reilly, 2010.

[30] T. Neil. 12 standard screen patterns. Accessed: 2014-01-22. [Online]. Available: http://designingwebinterfaces.com/designing-web-interfaces-12-screen-patterns

[31] D. Sinnig, A. Gaffar, D. Reichart, P. Forbrig, and A. Seffah, "Patterns in model-based engineering," in *Computer-Aided Design of User Interfaces IV*. Springer, 2005, pp. 197–210.

[32] J. Lin and J. A. Landay, "Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1313–1322.

[33] F. Pontico, M. Winckler, and Q. Limbourg, "Organizing user interface patterns for e-government applications," in *Engineering Interactive Systems*. Springer, 2008, pp. 601–619.

[34] S. Brin, "Extracting patterns and relations from the world wide web," in *The World Wide Web and Databases*. Springer, 1999, pp. 172–183.

[35] C.-H. Chang, C.-N. Hsu, and S.-C. Lui, "Automatic information extraction from semi-structured web pages by pattern discovery," *Decision Support Systems*, vol. 35, no. 1, pp. 129–147, 2003.