Inferring User Interface Patterns from Execution Traces of Web Applications

Miguel Nabuco[1], Ana C. R. Paiva[1,2], João Pascoal Faria[1,2]

[1]Department of Informatics Engineering, [2] INESC-TEC
Faculty of Engineering of University of Porto
Porto, Portugal

Abstract. This paper presents a dynamic reverse engineering approach to extract User Interface (UI) Patterns from existent Web Applications. Firstly, information related to user interaction is saved, in particular: user actions and parameters; the HTML source pages; and the URLs. Secondly, the collected information is analysed in order to calculate several metrics (e.g., the differences between subsequent HTML pages). Thirdly, the existent UI Patterns are inferred from the overall information calculated based on a set of heuristic rules. The overall reverse engineering approach is evaluated with some experiments over several public Web Applications.

1 Introduction

Web Applications are no longer solely used to display information. They have been used more and more to handle tasks that before could only be performed by desktop applications [11]. Also, due to this era of globalization, there is a constant need for communication and interaction with different locations around the world, and Web Applications play a key role in this process.

However, opposite to desktop and mobile applications, Web Applications suffer from a lack of standards and conventions [6]. For example, distinct Web sites may use different tags for the same HTML elements, the same functionality may be implemented with dissimilar code, and the same task may be handled in diverse ways. This characteristic has a negative impact over Web testing because it inhibits the reuse of test code.

Despite this diversity in Web Applications, there are some elements that developers frequently use to provide the users a sense of comfort and easiness, such as User Interface (UI) Patterns [27]. UI Patterns are recurring solutions that solve common UI design problems. They provide an easy way to perform certain tasks, such as searching for information (Search pattern) or authenticating to access private data (Login pattern). When a user sees an instance of a UI Pattern, he can easily infer how the UI is supposed to be used.

Despite a common behavior, UI Patterns may have slightly different implementations along the Web. However, it is possible to define generic and reusable test strategies to test them after a configuration process to adapt the tests to those different possible applications [16]. That is the main idea behind the

Pattern-Based GUI Testing (PBGT) project in which context this research work is developed. In the PBGT approach, the user builds a test model containing instantiations of the aforementioned test strategies (UI Test Patterns) for testing occurrences of UI Patterns on Web Applications.

The goal of the work described in this paper is to develop a reverse engineering (RE) process to automatically identify the presence of UI Patterns on existent Web Applications, and facilitate the building of test models in the context of PBGT. Such test models need to be completed and configured manually in order to generate test cases for testing the Web Application.

The rest of the paper is structured as follows. Section 2 presents an overview of the PBGT approach, setting the context for this work. Section 3 describes the UI patterns that our approach is able to detect. Section 4 describes the reverse engineering process and heuristics. Section 5 demonstrates the effective use of the approach in several existent Web Applications. Section 6 addresses related work. Section 7 provides the conclusions, sums up the positive points and the limitations of this approach, and points out the future work.

2 PBGT Overview

Pattern Based GUI Testing (PBGT) is a testing approach that aims to increase the level of abstraction of test models and to promote reuse. PBGT [15] has the following components (see also Figure 1):

- **PARADIGM** — A domain specific language (DSL) for building GUI test models based on UI patterns;
- **PARADIGM-ME** — A modeling and testing environment to support the building of test models;
- **PARADIGM-TG** — A test case generation tool that builds test cases from PARADIGM models;
- **PARADIGM-TE** — A test case execution tool to execute the tests, analyze the coverage and create reports;
- **PARADIGM-RE** — A dynamic reverse engineering approach and tool to extract UI Patterns from existent Web Applications without access to their source code. This tool can also generate test models to use in PARADIGM-ME.

The focus of this paper is on this extracting process (PARADIGM-RE). In Figure 1, the activities (rounded corner rectangles) with the human figure mean that they are not fully automatic requiring manual intervention. The activities with the cog mean that part (or all) of that activity is automatic. The numbers within the activities define their sequencing.

The definition of the PARADIGM DSL has started by analyzing a set of existing UI Patterns [27]. 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 has also a set of connectors that allows defining the sequencing of test strategies execution.

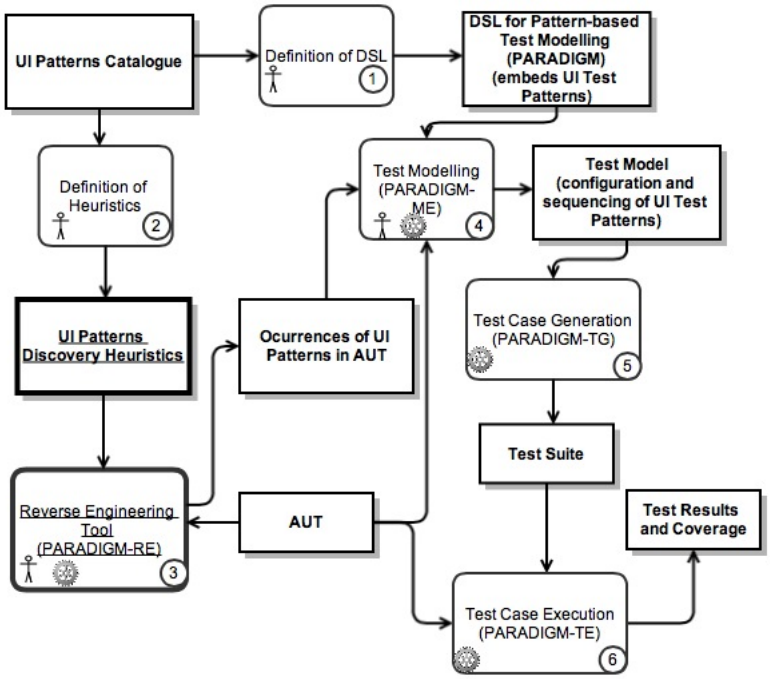

Fig. 1. PBGT overview

In order to diminish the modelling effort, the PARADIGM-RE tries to infer the existing UI Patterns within an Application Under Test (AUT) based on a set of heuristics. PARADIGM-RE will also automatically create a PARADIGM model, with the appropriate UI Test Patterns to test the identified UI Patterns. This model consists of a XML file that contains all the information about the UI Patterns found: their definition and configurations. The PARADIGM model generated by the RE tool does not contain the connectors between the UI Patterns, only the UI Patterns and their input data.

Afterwards, the extracted model may be completed manually in PARADIGM-ME. In particular, the connectors need to be added to define the sequence of execution. Afterwards, the model needs to be configured. During this configuration process, the tester provides preconditions (defining when the corresponding test strategy can be executed) [14] and checks to verify if a particular element is behaving as expected. Then, PARADIGM-TG can generate test cases based on the model, creating a test suite. PARADIGM-TE will then execute the test cases over the AUT to find errors and generate test coverage reports to assess the quality of the test suite.

3 Supported UI Patterns

The UI Patterns that the RE approach is able to detect are described in the sequel. These UI Patterns were chosen because they are the ones used by the modeling environment (PARADIGM-ME). To provide maximum compatibility between these two tools, the UI Pattern set chosen was the same. Also, most of the Web Applications can be almost entirely modeled using this set of UI Patterns.

- **Login UI Pattern** The login UI pattern is commonly found in Web Applications, especially in the ones that restrict the access to some functionalities/data. Besides two input fields (usually password is a cyphered text box, meaning that the characters input are only visible as symbols) and a submit button, sometimes this pattern has a "remember me" button that saves the authentication data for the next visit to the Website. The authentication process has two possible outcomes: valid and invalid.

- **Search UI Pattern** The search UI pattern consists of one or more input fields where the user inserts the subjects he wants to look for; and a submit button to start the search, sending the data to the server, retrieving the results and showing them to the user. When the search succeeds, it shows the results; when it fails, it may show a message explaining that no results were found.

- **Sort UI Pattern** The sort UI pattern sorts a list of data, for example price, name and relevance, according to some defined criteria, e.g., ascending or descending. For example, in a Web store a user can sort a list of a specific type of products according to their price in order to identify the cheapest one.

- **Master Detail UI Pattern**
  The master detail UI pattern is present in a Web page when selecting an element from a set results in filtering/updating another related set accordingly. As an example, consider a Web Application that displays the most sold audio CD in a certain week. There is an option, for example, a dropdown box that lets the user choose the week he wants to analyze (master). By changing the master selection, the list of the most sold CDs (detail) changes accordingly. Usually, besides the detail update, the majority of the Web page remains the same.

- **Menu UI Pattern** The menu UI pattern is very common in Web pages. It defines a tree structure with several options, for instance, to access some options/functionalities of a Web site.

- **Input UI Pattern** An input UI pattern is simply an input field (usually a text box) where a user can insert data.

4 Reverse Engineering Approach

The Web does not have specific design standards and the UI patterns can have slightly different implementations along the Web, so the definition of a global approach that can infer them is not a trivial activity. As depicted in Figure 2, the reverse engineering approach presented in this paper comprises the following components that are executed sequentially:

- **Extractor** — extracts the following information (using Selenium [24], a tool for automating interaction with Web Applications) from a user interaction with a Web Application:
  - the source HTML code of the Web pages visited;
  - the actions performed by the user (e.g., a click, a text input);
  - the URL of each page visited.
- **Analyser** — analyses the information gathered before and computes some metrics (e.g., differences between the source code of two subsequent Web pages);
- **Inferrer** — infers the UI Patterns based on heuristic rules defined over metrics and data calculated during the analysis phase (e.g., percentage of HTML changes between two subsequent visited pages).

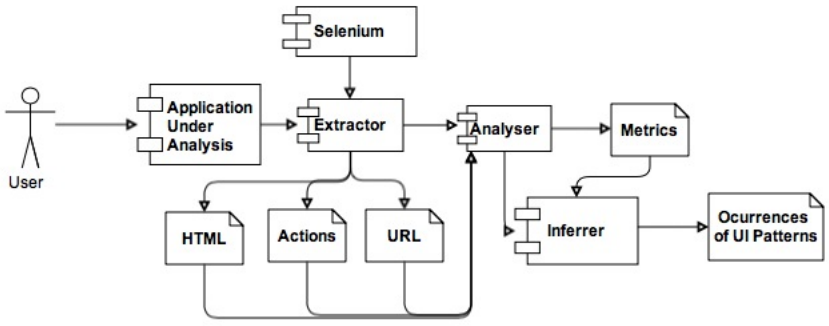

Fig. 2. Reverse Engineering Process

These components are described in more detail in the sequel.

A - Extractor

During the extraction phase, the PARADIGM-RE tool gathers the information needed to infer the UI patterns. This section describes such information and how it is extracted.

The information extracted comprises the following:

- **Source code of HTML pages loaded** — the tool records the HTML source code of the Web pages loaded during user interaction; we denote by $\mathbf{HTML_i}$ the source code of the i-th page loaded, with $1 \leq i \leq N$, where N is the total number of pages loaded;
- **URLs** — the tool records the URLs of the Web pages loaded; we denote by $\mathbf{URL_i}$ the URL of the i-th page loaded, with $1 \leq i \leq N$;
- **User actions (UA)** — the tool records in a HTML table, using Selenium [12], all the actions performed by the user during interaction; we denote by $\mathbf{UA_k}$ the *k-th* user action recorded, with $1 \leq k \leq M$, where M is the total number of user actions recorded.
  Each recorded user action $\mathbf{UA_k}$ has the following information:
  - $\mathbf{Command_k}$ — is the action that the user performed. Can be either"*type*", "*click*", "*clickAndWait*" (e.g., when the user clicks in a Web page element that loads a new page afterwards), "*typeAndWait*" (typing that loads different pages dinamically) and "*select*" (e.g., when the user clicks an option from a HTML dropdown list).
  - $\mathbf{TargetId_k}$ — is the id of the UI element object of the interaction.
  - $\mathbf{Value_k}$ — is related to "*type*" and "*typeAndWait*" actions and saves the string typed by the user in the text box. On "*select*", value is the option selected by the user.

We also denote by $\mathbf{Page_k}$ the index (between 1 and N) of the page where $\mathbf{UA_j}$ (between 1 and M) occurred. It is worth noting that the only actions that cause load of a new Web page are the "*clickAndWait*" and "*typeAndWait*" actions. An example of an action saved using Selenium is the following:

```
<tr>
<td>type</td> <!- -Command- ->
<td>name=artist</td> <!- -Target - ->
<td>u2</td> <!- -Value - ->
</tr>
```

In this action, the user typed "u2" (the value) in the text box with the *id* "name=artist" (target). An example of the information saved during a user interaction with a Web Application can be seen in Table 1.

The RE tool is implemented in Java. The extraction process is performed according to the following workflow:

1. When the RE tool starts, it launches the Firefox Web browser and navigates to the specified URL of the AUT, given as a parameter.
2. The RE tool saves the initial HTML source page as $HTML_1.txt$.
3. The user starts Selenium IDE within Firefox.
4. The RE tool saves the user actions performed. After a user action of the kind "*clickAndWait*" or "*typeAndWait*", the RE tool saves the HTML of the new loaded page.

Table 1. Example Execution Trace

UA index	Page index	Command	TargetId	Value
-	-	open	/account/login.php	-
1	1	Type	name=formloginname	pbgt
2	1	Type	name=formpw	pbgtpass
3	1	Click	id=rememberusernamepwd	-
4	1	clickAndWait	name=login	-
5	2	clickAndWait	css=button[type="button"]	-
6	3	Click	link=Me	-
7=M	3=N	clickAndWait	link=Log Out	-

5. Step 4 is repeated until the user presses the "Escape" key on the keyboard, signaling the end of the extraction process, i.e., the end of an execution trace.

B - Analyser

In order to infer the UI patterns, the analyser calculates the following metrics and derived data, based on the information previously extracted:

- **HTMLDiff** — For every pair of consecutive HTML source files, $HTML_i$ and $HTML_{i+1}$, it calculates the difference between them, denoted $HTMLDiff_{i+1}$, containing the lines of HTML code updated, as determined by the *java-diff-utils* library [13].
- **RatioTotal** — For each $HTMLDiff_i$ file, it computes the ratio between its size and the average size over all the *HTMLDiff* files, denoted and calculated as follows:

$$RatioTotal_i = \frac{size(HTMLDiff_i)}{\frac{\sum_{j=2}^{N} size(HTMLDiff_j)}{N-1}}, 2 \leq i \leq N \tag{1}$$

- **RatioPrevious** — For each pair of consecutive HTML source files, it computes the difference between their sizes, calculated according to the formula:

$$RatioPrevious_{i+1} = S_{i+1} - S_i \tag{2}$$

$$S_i = size(HTML_i) \tag{3}$$

$$S_{i+1} = size(HTML_{i+1}) \tag{4}$$

- **URL Keywords** — URLs can have keywords that may be useful to help inferring some UI patterns. We denote by $\mathbf{keywords(URL_i)}$ the set of keywords contained in URL_i.
- **Textboxes ids and types** — Some textboxes may have valuable information in their names or ids (found in the HTML page source), which may lead to the detection of a UI Pattern. We denote by $\mathbf{TargetId_k}$ and $\mathbf{Type_k}$ the *id* and *type* of user action UA_k.

– **Number of occurrences of certain strings in HTML pages** — The number of times certain strings appear in a HTML page can help in pattern inference. We denote by $\mathbf{count(s, HTML_i)}$ the number of occurrences of string s in $HTML_i$.

After all the calculations, the UI inferring process starts.

C - Inferrer

Occurrences of UI patterns in the pages visited are inferred from the information gathered in the previous phases, based on a set of heuristics defined specifically for each UI pattern. The heuristics defined for a UI pattern P are combined with different weights to produce a final score *S(P)* that indicates the confidence of the presence of that UI pattern.

The heuristics weights (in Tables 2, 3, 4, 5) were defined based on a training set of Web Applications. For each application inide that set, we run the RE-Tool and tune the value of the weights for the existent UI patterns. The final values are the ones that inferr more patterns within those Web Applications. This training process is not completely automatic because the user needs to provide information about when a specific UI pattern is present.

The weights are assigned so that a score $S(P) \geq 1$ indicates a sufficiently high confidence to assume that the UI pattern occurs. So the score is calculated using the formula:

$$S(P) = \sum_{i}^{N} W(i)V(i), \tag{5}$$

In this formula, i is the heuristic being evaluated, N is the total number of heuristics for that specific UI pattern, *W(i)* is the weight of heuristic i and *V(i)* indicates if the heuristic i holds (value 1) or not (value 0).

There are also pre-conditions associated with some UI Patterns that must hold in order to procede with the score calculation.

D - UI Patterns Heuristics

In the sequel we present the heuristics and corresponding weights defined for each UI Pattern.

Login

The score calculation process starts just when there are two "*type*" commands with different ids in the same page. This is the pre-condition. After that, the RE tool looks for common keywords, such as "login" (heuristic **A**), "user" and "pass" (heuristic **B**), in their *ids*. Most often, the input fields (usually textboxes) to receive passwords transform the characters entered in cyphered tokens, which is

coded in HTML assigning "password" to the "*type*" of the input field (heuristic **C**) as shown next.

<input id="appassword" name="password" type="password">

The URL is also parsed to find "login" (or "logon") keywords that, although not found frequently in the URL, increase the degree of confidence in the presence of a login UI pattern (heuristic **D**).

Table 2 summarizes the heuristics defined for finding occurrences of the Login UI Pattern, including the mandatory pre-condition. Some of the conditions presented below are derived from the UAs.

Pre-Condition: Two "type" commands UA_j and UA_k with $TargetId_i \neq TargetId_k$ occurring in page i. (Without loss of generality, we assume that UA_j refers to the username and UA_k refers to the password.)

Table 2. Login pattern heuristics

ID	Heuristic	Weight
A	"login" is-substring-of $TargetId_j$	0.8
B	"user" is-substring-of $TargetId_j \wedge$ "pass" is-substring-of $TargetId_k$	0.8
C	"password" is a substring of $Type_k$	0.6
D	{"login","logon"} $\cap$ keywords(URL_i) $\neq \emptyset$	0.8

Search

The process to look for Search UI Pattern starts only if the execution trace (e.g., Table 1) has one "*type*" command preceding a "*clickAndWait*" command. So, the inferring process applied to identify a Search UI pattern starts by analyzing the input field where the user inserted the content that he wants to search for. In several Websites, such field is a text box named '*search*", or contains the word '*search*" in its *id* or *title* (heuristic **E**). For instance, in the Amazon Website, the search box id contains the keyword search, as it can be seen in the example below with the information retrieved by Selenium from the user interaction.

<input type='text' id='twotabsearchtextbox' title='Search For' value=" name='field-keywords' autocomplete='off'>

When performing a Search operation, most of the times the URL may have additional interesting information, such as, the word "search" (heuristic **F**) and the content to search for (heuristic **G**). For example, in the Australian Charts

Website, if a search for "daft punk" is performed, the URL will be:

```
www.australian-charts.com/search.asp?search=daft+punk&cat=s
```

To complete the Search pattern inference, the RE tool counts the number of times that the content to look for appears in the resulting Web page. If the total is higher than a specific number, there is more confidence that a Search UI pattern was found (heuristic **H**). The overall heuristics used to find occurrences of the Search UI pattern are summarized in Table 3.

Pre-Condition: A "*type*" command UA_k precedes a '*clickAndWait*" command in page i.

Table 3. Search pattern heuristics

ID	Heuristic	Weight
E	"search" is-substring-of $TargetId_k$	0.8
F	"search" $\in$ keywords(URL_{i+1})	0.7
G	$Value_k \in$ keywords(URL_{i+1})	0.6
H	count($Value_k, HTML_{i+1}$) > 10	0.8

Sort

To infer the presence of a Sort UI pattern, some heuristics were defined based on the *RatioPrevious*, *RatioTotal* and *URL keywords* calculated before. Considering that most of the times, performing a sort operation, does not change much the HTML source code of the current page, it is a good indicator of the presence of a Sort UI pattern when:

- *RatioPrevious* is low ($<$ 5 KB) meaning that there are few differences between the HTML source code of the current page and the previous one (heuristic **K**);
- *RatioTotal* is low ($<$ 0.15) indicating that differences between the HTML source code of current page and the previous one is considerably less that the average of differences between all pairs of consecutive pages of the execution trace (heuristic **L**).

In addition, it is very common to find some keywords within the corresponding URL when a Web page has a Sort UI pattern, such as *asc* (ascending), *desc* (descending), *order* (heuristic **J**), *sort* and *sortType* (heuristic **I**). The heuristics defined to find occurrences of the sort UI pattern are summarized in Table 4.

Master Detail

As it is explained in the previous section, the two HTML captures (before – $HTML_i$ – and after – $HTML_{i+1}$ – changing a master value) are very similar,

Table 4. Sort pattern heuristics

ID	Heuristic	Weight
I	{"sort","sortType"} $\cap$ keywords(URL_{i+1}) $\neq \emptyset$	0.8
J	{"asc","desc", "order"} $\cap$ keywords(URL_{i+1}) $\neq \emptyset$	0.6
K	$RatioPrevious_{i+1} < 5$ KB	0.4
L	$RatioTotal_{i+1} < 0.15$	0.4

containing very few changes. There are two simple metrics defined on top of this difference. The RatioPrevious value (heuristic **M**), along with the RatioTotal value (heuristic **N**), will allow the algorithm to know the amount of information changed between the two pages and its correlation with the rest of the pages in that Web Application. The other metric is the number of lines in the HTMLDiff file (heuristic **O**). The heuristics used for finding occurrences of this pattern are summarized in Table 5.

Table 5. Master Detail pattern heuristics

ID	Heuristic	Weight
M	$RatioPrevious_{i+1} < 5$ KB	0.5
N	$RatioTotal_{i+1} < 0.1$	0.5
O	number of lines($HTMLDiff_{i+1}$) < 6	0.5

We tested these heuristics on the Web Applications training set and tuned the values that would determine or not the presence of a Master Detail pattern. For the case of RatioPrevious we set the value to 5 KB meaning that when the difference between two consecutive pages is less than 5 KB value there is more probability of having found a Master Detail. For the case of RatioTotal, we assumed value 0.1. For the case of the number of lines, when this value is lower than 6 it means that there is more probablity of inferring a Master Detail.

Input

In each page, an occurrence of the Input pattern is inferred for each object ID that is the target of at least one "type" command in that page, and is not involved in a Search or Login pattern in the same page (as search string, username or password).

Menu

The ambiguity of the definition of menu in a Web Application makes its recognition very difficult. A menu can be anywhere on the page, with or without subsections, and be spread horizontally or vertically. It is usually on the top, bottom, left or right part of the page. There are no specific HTML elements to

construct a menu, and so the Website developers are free to implement a menu in the way they desire. However, a good number of them use a specific menu type, called "Site navigation" [28]. Its structure is similar to the following example:

```
<div><ul>
<li><strong>Home</strong></li>
<li><a href="about.html">About Us</a></li>
<li><a href="clients.html">Our Clients</a></li>
<li><a href="products.html">Our Products</a></li>
<li><a href="contact.html">Contact Us</a></li>
</ul></div>
```

Usually the *href* attributes in a menu are set to local pages. From all the HTML page sources, the algorithm tries to find common elements among 80% of them (since there can be pages in a Web Application that contain no Menu pattern), and sees if, from the common elements, there is a general structure like the one described above. There can be either zero or one instance of a Menu UI pattern per execution trace.

5 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 heuristics and weights used to maximize the capacity to find UI patterns.

If during the RE 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.

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. The Web Applications chosen for this evaluation set were different than the ones used to refine the heuristics in the training set, to prevent biased results.

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 15 most popular Websites [9]: Amazon, YouTube, Facebook, Wikipedia and Yahoo. These Web Applications were chosen because they contain instances of the UI patterns the approach proposes to detect. Also, they are well designed and contain very few known bugs.

The numerical results were combined and can be found in Table 6.

Overall, the success rate of the tool was quite satisfactory. The tool found 67 of 99 pattern occurrences (67.7%).

Despite some miscounts, it identified most of the patterns. The Menu and the

Table 6. Evaluation set results

Pattern	Present in ET	Correctly found	False positives
Login	9	6	0
Search	24	21	0
Sort	11	4	0
Input	28	26	3
Menu	3	1	0
Master Detail	24	9	9
Total	**99**	**67(67.7%)**	**12**

Master Details patterns were the most problematic. This is due to the ambiguity of the Master Detail definition, meaning that a Master Detail can be presented in many ways, making it difficult to specify precise heuristics. The Search and Login patterns were almost always found correctly due to their heuristics being more strict and specific. The three Input false positives were in fact text boxes belonging to the three Search patterns that were not found.

6 Related Work

Reverse engineering is "*the process of analysing 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*" [5]. In reverse engineering, the subject system is not changed. There are different 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 [20], the static method, which obtains the data from the system source code [30], and the hybrid method, which combines the two previous methods.

There are several ways to obtain information from application execution traces [25] [2]. ReGUI [17] is a tool that reduces the effort of obtaining models of the structure and behaviour of Graphical User Interfaces (GUI) [18]. Another approach for behaviour model extraction combined static and dynamic information and was used in model checking [8]. EvoTrace [10] uses execution traces to trace the evolution of a software system. Amalfitano [1] presented a technique for testing Rich Internet Applications (RIAs) that generates test cases from the applications execution traces.

Several tools have been created to obtain information from Web Applications. Re-Web [23] obtains dynamically information from Web server logs that helps to find structural and navigational problems in Web Applications. WARE [7] is a static analyzer that creates UML diagrams from the Web Application source code, Crawljax [24] is a tool that obtains graphical sitemaps, by automatically crawling through a Web Application and Selenium is an open-source capture-replay tool that saves the users interaction in HTML files.

Concerning UI Patterns, there are different sources that enumerate the most popular/frequently used ones [21] [22] [26] [29]. Although certain pattern layouts

may be different from different sources or implementations, they all present a common behaviour and functionality. As an example, the Master Detail pattern may be presented graphically in several ways but its functionality stays the same: selecting a master option will filter and show different results for the detail area.

In the area of inferring patterns from Web Applications, it was discovered that, despite the fact that there are approaches to extract information from the Web, besides our previous work applying Inductive Logic Programming (ILP) to execution traces to infer UI Patterns [19] [16], none of the approaches found deal with UI pattern detection. They mostly deal with other sets of patterns, more related to the area of Web mining. The purpose of these patterns is to find relations between different data or to find the same data in different formats. Brin [3] presents an approach to extract relations and patterns for the same data spread through many different formats. Chang [4] proposes a similar method to discover patterns, by extracting structured data from semi-structured Web documents.

7 Conclusions and Future work

This paper proposed a dynamic reverse engineering method to identify recurrent behavior present in 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 performed on the Web Application.

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 work still needs some improvement. The features planned for future versions of the RE tool include the support for a larger set of UI patterns and the definition of more precise heuristics based on other web elements (such as manipulating the HTML DOM tree).

Acknowledgments

This work is financed by the ERDF — European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT — Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.

References

1. D. Amalfitano. “Rich Internet Application Testing Using Execution Trace Data”. Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 274-283, 2010
2. I. Andjelkovic, C. Artho. “Trace Server: A tool for storing, querying and analyzing execution traces”. JPF Workshop, 2011
3. S. Brin. “Extracting Patterns and Relations from the World Wide Web”. WebDB Workshop at 6th International Conference on Extending Database Technology. 1998
4. C. H. Chang, C. N. Hsu, S. C. Lui. “Automatic information extraction from semi-structured Web-pages by pattern discovery”. Decision Support Systems Journal, vol. 35, pp. 129-147, 2003
5. E. J. Chikofsky, J. H. Cross. “Reverse engineering and design recovery: a taxonomy”. IEEE Software Journal, vol.7, no.1, pp.13-17, Jan. 1990
6. L. L. Constantine, L.A.D Lockwood. “Usage-centered engineering for Web applications”. IEEE Software Journal, vol.19, no.2, pp.42-50, Mar/Apr 2002
7. M. Di Penta; “Integrating static and dynamic analysis to improve the comprehension of existing Web applications”; Proceedings of 7th IEEE International Symposium on Web Site Evolution, pp. 87-94, 2005
8. L. M. Duarte, J. Kramer, S. Uchitel. “Model extraction using context information”. In Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems (MoDELS’06), pp. 380-394, 2006
9. eBizMba. “Top 15 most popular Websites”. http://www.ebizmba.com/articles/most-popular-Websites . Accessed June 2013
10. M. Fischer, J. Oberleitner, H. Gall, T. Gschwind. “System evolution tracking through execution trace analysis”. CSMR 2005, pp. 112-121, 2005
11. J. J. Garrett. “Ajax: a new approach to Web applications”. http://www.adaptivepath.com/publications/essays/archives/000385.php .2006
12. G. Gheorghiu. “A look at Selenium”. Better Software. October 2005
13. Java-diff-utils. The DiffUtils library for computing diffs, applying patches, generating side-by-side view in Java. http://code.google.com/p/java-diff-utils/ . accessed Jun 2013
14. T. Monteiro, A. Paiva. “Pattern based gui testing modeling environment”. Fourth International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software, TESTBEDS. 2013
15. R. Moreira, A. Paiva, A. Memon. “A Pattern-Based Approach for GUI Modeling and Testing”. Proceedings of the 24th annual International Symposium on Software Reliability Engineering (ISSRE 2013), 2013
16. I. C. Morgado, A.C.R Paiva, J.P. Faria, R. Camacho. “GUI reverse engineering with machine learning”. Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on, pp.27-31, June 2012
17. I. C. Morgado, A.Paiva, J.P. Faria. “Dynamic Reverse Engineering of Graphical User Interfaces”. International Journal on Advances in Software, vol. 5, pp. 223-235, 2012
18. I. C. Morgado, A. Paiva, J. P. Faria. “Reverse Engineering of Graphical User Interfaces”. in The Sixth International Conference on Software Engineering Advances, Barcelona, pp. 293-298, 2011
19. M. Nabuco, A. Paiva, R. Camacho, J. Faria. “Inferring UI patterns with inductive logic programming”. 8th Iberian Conference on Informations Systems and Technologies (CISTI), 2013

20. A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria, "Reverse Engineering of GUI Models for Testing", 5th Iberian Conference on Information Systems and Technologies (CISTI), Santiago de Compostela, Spain, 2010
21. T. Neil. “12 standard screen patterns”. http://designingWebinterfaces.com/designing-Web-interfaces-12-screen-patterns . accessed June 2013
22. Pattenry. “UI design patterns and library builder”. http://paternry.com . accessed June 2013
23. F.Ricca, P.Tonella. “Understanding and restructuring Web sites with ReWeb”. IEEE Multimedia, vol. 8, pp. 40-51, Apr-Jun 2001
24. D. Roest. “Automated Regression Testing of Ajax Web Applications”. Faculty EEMCS, Delft University of Technology Msc thesis, 2010
25. J. Steven, P. Ch, B. Fleck, A. Podgurski. “jRapture: A capture/replay tool for observation-based testing”. Proceedings of the International Symposium on Software Testing and Analysis, ACM Press, pp. 158-167, 2000
26. A. Toxboe. “UI patterns - user interface design pattern library”. http://ui-patterns.com . accessed June 2013
27. M. Welie, C. Gerrit, A. Eliens. “Patterns as tools for user interface design”. Workshop on Tools for Working With Guidelines. Biarritz, France, 2000
28. W3C Wiki. Creating multiple pages with navigation menus. http://www.w3.org/wiki/Creating_multiple_pages_with_navigation_menus . accessed June 2013
29. M. van Welie. “Interaction design pattern library”. http://www.welie.com/patterns/ . accessed June 2013
30. J. C. Silva, C. Silva, R. D. Gonçalo, J. Saraiva, and J. C. Campos. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems. 2010