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. In addition, the model itself, while a powerful tool of abstraction, can have conceptual errors, introduced by the tester. These problems can be reduced by generating those models automatically. 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 reverse engineering approach explores automatically any web application, records information related to the interaction, analyses the gathered information, tokenizes it, and infers the existing UI patterns via syntactical analysing. After being complemented with additional information and validated, the model extracted is the input for the Pattern-Based Graphical User Interface Testing (PBGT) approach for testing existing web application under analysis.**

***Keywords*-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. Although web applications are changing, many of them still lack standards and conventions [2], [3], 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 (GUI) of all kinds are populated with recurring behaviours that vary slightly. An example is the authentication (login/password) process: while most implementations trigger the appearance of an error message when the wrong credentials are inserted, some simply erase the inserted data, with no visible error message. Another one is the content search that may vary the criteria to sort ascending or descending. These behaviours (patterns) are called User Interface (UI) patterns [4] and are recurring solutions that solve common UI design problems. Due to their widespread use, UI patterns allow users a sense of familiarity and comfort when using applications.

While UI patterns are familiar to users, their implementation may vary; 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 [5].

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 application. The model can be built manually by the user, but this is a time consuming process and not recommended, since it may introduce conceptual errors. Half the bugs found through model-based testing are conceptual errors in the model itself and not bugs in the system to test [6].

The goal of the work described in this paper is to continue previous work done in [7] on the reverse engineering process (PARADIGM-RE). More specifically, this research work aims to automate the model construction: independently/automatically explore a web application, infer the existing UI patterns in its pages, and finally produce part of the model with the UI Test Patterns that define the strategies to test the UI Patterns present in the web application. The model extracted is afterwards validated and complemented with additional information. This reverse engineering process speeds up model construction, and hopefully introduces less errors into the model.

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

The focus of this paper is the reverse engineering component of a research project named PBGT (*Pattern-based GUI Testing*) [8]. The goal of this project is to develop a model-based GUI testing approach and supporting tool, that can be used in industrial contexts. The models are built around the concept of UI patterns (common solutions to recurrent design problems) and are able to test recurrent behaviour on web applications regardless of implementation, after a testing configuration step in which the tester defines checks to perform, any preconditions necessary, and other components.

A. Architecture

The PBGT supporting tools have five main components:

1) **PARADIGM**, a DSL (*Domain Specific Language*) to define GUI testing models based on UI Test Patterns [9];
2) **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 to use that information to generate a test model written in PARADIGM;
3) **PARADIGM-ME**, a modelling and testing environment, built to support the creation of test models [10];
4) **PARADIGM-TG**, an automatic test case generation tool that generates test cases from test models defined in PARADIGM according to coverage criteria selected by the tester;
5) and finally, a test case execution tool, named **PARADIGM-TE**, which executes test cases, analyses their coverage with a coverage analysis tool named **PARADIGM-COV**[11], and returns detailed execution reports.

The architecture and workflow of the PBGT is shown in Fig. 1.

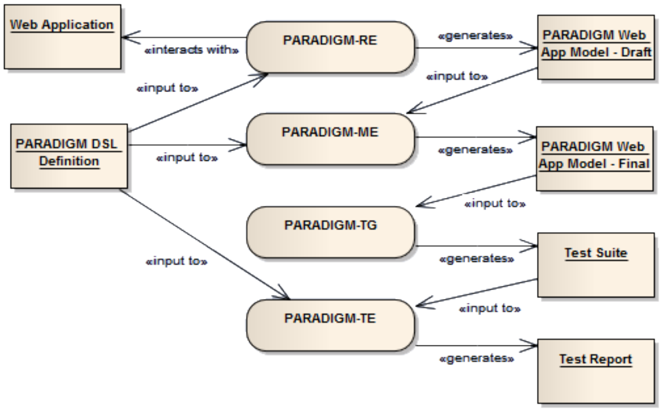

Figure 1: An overview of the PBGT project.

B. PARADIGM DSL

PARADIGM is a DSL to be used in the domain of PBGT. The goal of the language is to gather applicable domain abstractions (e.g., UI test patterns), allow specifying relations between them and also provide a way to structure the models in different levels of abstraction to cope with complexity. PARADIGM's meta-model is illustrated in Fig. 2.

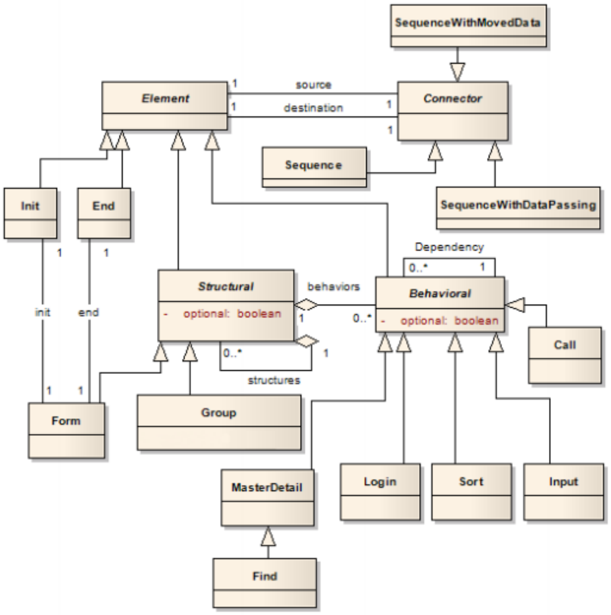

Figure 2: PARADIGM DSL Meta-model.

The PARADIGM language is comprised by elements and connectors [8]. There are four types of elements: *Init* (to mark the beginning of a model), *End* (to mark the termination of a model), *Structural* (to structure the models in different levels of abstraction), and *Behavioural* (UI Test Patterns describing the testing goals) [12].

As models become larger, coping with their growing complexity forces the use of structuring techniques such as different hierarchical levels that allow the use of an entire model **A** inside another model **B**, while abstracting the details of **A** when within **B**. Structuring techniques are common in programming languages like C++ and Java, with constructs such as modules and classes. *Form* is a structural element that may be used for that purpose. A *Form* is a model (or sub-model) with an *Init* and an *End* elements. *Group* is also a structural element but it does not have *Init* and *End* and, moreover, all elements inside the *Group* are executed in an arbitrary order.

In the concrete syntax of the PARADIGM language, elements and connectors are described by: (i) an icon/figure to represent the element graphically and (ii) a short label to

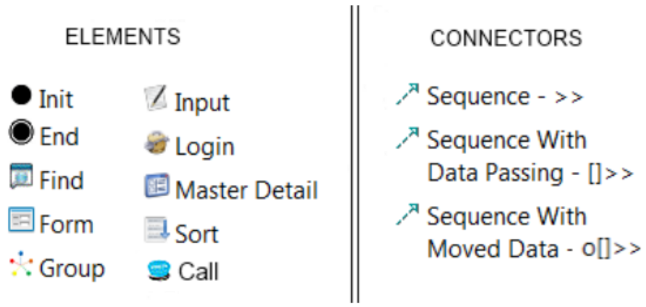

Figure 3: PARADIGM syntax.

name the element. The overall concrete syntax of the DSL is illustrated in Fig. 3. Additionally, elements within a model have a number to identify them, and optional elements have a "op" label next to its icon/figure.

This language has three connectors: *Sequence*, *SequenceWithDataPassing*, and *SequenceWithMovedData*. *Sequence* indicates that the testing strategy related to the target element cannot start until the testing strategy of the source element completes. *SequenceWithDataPassing* has the same meaning as *Sequence*, and additionally indicates that the target element receives data from the source element. *SequenceWithMovedData* has a similar meaning to *SequenceWithDataPassing*, but the source element moves data to the target instead of transferring a copy. In addition, there is another kind of relation among elements – *Dependency* – indicating that the target element depends on the properties of a set of source elements, for instance, when it is the result of a calculation.

C. UI Test Patterns

A UI Test Pattern defines a test strategy, which is formally defined by a set of test goals (for later configuration)[8], with the form:

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

- **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 language also defines language constraints to guarantee the building of well-formed models, such as "*A Connector cannot connect an element to itself*" and "*A Connector cannot have Init as destination, or End as source*", to cite a few examples.

The UI Patterns defined in the PARADIGM language 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 text box for email or username, and a cyphered text box for the password) and a submit button, with optionally a "remember me" checkbox. The authentication process has two possible outcomes: valid and invalid. Upon authentication failure a message may be shown.
- **Find**: 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 (e.g., price, name, relevance, alphabetically, etc.) according to a defined criteria (ascending or descending).
- **Master Detail**: This pattern is present in a web page 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.
- **Call**: This pattern is any kind of element where a click triggers some procedure that may result in a change of page.

D. Produced Models

The models produced by the reverse engineering tool PARADIGM-RE consist of an XML file in the format required by the PARADIGM-ME which contains information about the UI Test Patterns needed to test the UI Patterns found: their name, the input values for their variables, i.e., the values used during the exploration process, and blank preconditions and checks for the tester to fill in. The UI Test Patterns within the model are connected in a linear path according to the exploration order. The testing configuration information needs to be complemented and validated by the tester afterwards in order to generate test cases and execute them over the web application under analysis.

III. Reverse Engineering Approach

A. Previous Tool

The approach described in this paper aims to improve on the previous work [7] done on the PARADIGM-RE tool. In particular it aims to be fully automatic. The previous tool required the intervention of the user to interact with the web application under analysis in order to save the interaction traces and proceed from there. It 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 [7].

The information saved from a user interaction was 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 Table I.

Action	Target	Value
type	id=input_username	"user1"
type	id=input_password	"123pass"
clickAndWait	css=input[type="submit"]	EMPTY
type	id=searchInput	"coffee"
clickAndWait	id=mw-searchButton	EMPTY
select	id=sort	label=Price:Low to High
click	id=collapseButton1	EMPTY
click	link=Next	EMPTY
typeAndWait	id=freeSearch	ministry
type	id=authcode	T75Y5
type	name=firstName	james
type	name=lastName	bond
click	//ul[@id='ref1']/li[5]/a/span	EMPTY

Table I: Example of an execution trace file.

The previous reverse engineering approach identified the *Find*, *Login*, *Sort*, *Input* and *MasterDetail* patterns, and produced a high number of false positives [7]. Additionally, as aforementioned, the exploration process was done by saving the user interaction with the web application under test, requiring human interaction to identify patterns. The results were dependent on the execution traces followed by the tester.

B. Current Tool

The work described in this paper has the same primary aim as the previous tool (to identify UI patterns in the Application Under Test (AUT)), but has other goals. The first is to remove the need for user interaction with the AUT to identify patterns, by providing a reverse engineering process to explore automatically the web application under analysis. The second is to refine and improve the pattern inferring process, and lower or eliminate completely the percentage of false positives. The third is to identify more patterns than the previous tool.

The workflow of the tool, as seen in Fig. 4, has three components: **WebsiteExplorer**, **LogProcessor**, and **PatternInferrer**.

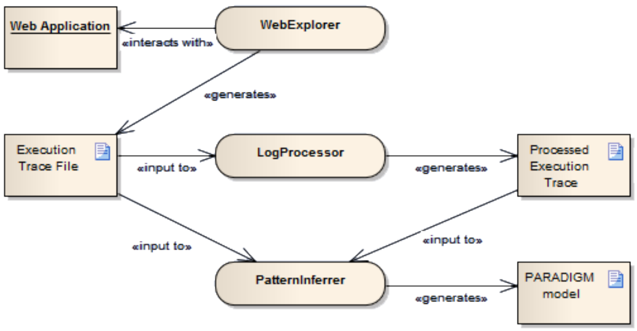

Figure 4: The workflow of the approach.

WebsiteExplorer loads user configurations (more details on Section III-D), interacts automatically with the AUT and produces an execution trace file with the actions taken (see Table I for an example). **LogProcessor** is a text parser. It analyses the previous file, parses each line, searches for important keywords and uses them to identify the action contained in the line, producing an updated execution trace file (see Table II for an example). Finally, **PatternInferrer** analyses the updated execution file, identifies the existent patterns, their location in the website and any existing parameters, and produces an XML file with the results. The patterns identifiable by this tool are the UI Patterns that PBGT can test through the current state of the PARADIGM language, plus the *Menu* UI pattern.

C. AUT Exploration (WebsiteExplorer)

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

The exploration process is random, i.e., the next element to visit is selected randomly from the set of elements not yet visited. All elements have the same probability of being chosen from a list of elements.

Currently the explorable elements are <*select*> elements, <*input*> elements, and <*a*> (link) elements. More elements will be explored in future extensions of the tool. 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 drop-down menus, a random option is selected and 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.Before interacting with an element, the algorithm makes the following checks: if the element has not been visited in the current page, and if the element contains any unwelcome keywords that if explored may drive the explorer

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

```
Data: number_of_actions, number_of_iterations,
      website_base_url, configuration_file
Result: execution_trace_file
current_action := 0; redirections := 0;
configuration := parseConfigurationFile();
while current_action <
configuration.number_of_actions do
    if configurations.isSearchingForMenu() then
        menuElements.add(getMenuElementsInPage());
    end
    if configuration.isSearchingForMasterDetail() then
        masterElements.add(getMasterElementsInPage());

        detailElements.add(getDetailElementsInPage());
    end
    list := extractValidNonVisitedElements;
    next_element := chooseNextElement(list);
    if next_element == null then
        redirections++;
        if redirection <
        configuration.number_of_iterations then
            go_to_base_URL;
            write_to_execution_trace_file;
        else
            end_program;
        end
    else
        visit(element);
        visitedElements.add(element);
        write_to_execution_trace_file;
        current_action++;
        wait(configuration.politenessDelay);
    end
end
```

into unwanted paths. 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). These keywords may be predefined by the user in a configuration file that the RE tool uses as input.

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 drop-down 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).

The exceptions are the *Menu* and *MasterDetail* patterns. Since the explorer cannot be relied on to explore every element belonging to these patterns in each page, elements belonging to these patterns are found through analysing the current page source and extracting all elements that obey a set of rules. For the Menu pattern, the anchor links that are in <*nav*>, <*header*> or <*footer*> tags are automatically included as part of the *Menu* pattern. Besides this, Menu elements and Master Detail are identified by a set of identifiers passed via the configuration file, and they are identified as its respective pattern if they respect the condition `(//*[contains(@class,identifier)] OR //*[contains(@id,identifier)] OR //*[contains(@name,identifier)])`.

The results of the HTML analysis are passed directly to the **PatternInferrer** to be written in the final output file (see Section III-F).

D. Configuration

To improve results, almost all components used in the website interaction and pattern identifying processes can be loaded to the application via an XML configuration file, the only exception being the *PatternInferrer* grammar. This is done to allow the maximum flexibility to the tester to adjust the tool to the web application to test.

The components and values that can be loaded via configuration file are:

- **actions**: number of actions the crawler will execute before stopping;
- **redirections**: number of redirections to the home page the explorer will do before stopping;
- **politenessDelay**: time to wait (in milliseconds) after each action;
- **typedKeywords**: list of words to insert in text input elements;
- **searchKeywords**: regex with keywords that identify search elements (part of the *Find* pattern);
- **sortKeywords**: regex with keywords that identify *Sort* elements;
- **loginKeywords**: regex with keywords that identify *Login* elements;
- **generalWordsToExclude**: regex with keywords that mark elements that should not be accessed;
- **menuIdentifiers**: list of ids/classes/names that identify menu elements;
- **masterIdentifiers**: list of ids/classes/names that identify master elements;
- **detailIdentifiers**: list of ids/classes/names that identify detail elements;
- **historyFilepath**: specifies absolute filepath for history file;
- **processedHistoryFilepath**: specify absolute filepath for processed history file;
- **patternsFilepath**: specify absolute filepath for PARADIGM model file;

- **patternsToFind**: list of patterns that the explorer can search for - if ”all” occurs, there will be no restriction;
- **loginConfiguration**: credentials for a correct login in the web application, specified by the tester;
- **tokenizerPatterns**: patterns for the *LogProcessor* process;
- **includeChildrenNodesOnInteraction**: boolean variable; indicates that, on interaction with an input or select element, sibling elements should be marked in the history file, but not interacted with.

E. Action File Processing (LogProcessor)

This component is a lexical analyser, 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, and a regex (regular expression)named **identifying_regex** that identifies an action of that type.

For every line, not all rules are tested; 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, added to the processed trace file, and then the program moves on to the next line. If no rules match, only the action is written on the 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
click	id=collapseButton1	EMPTY	clickCollapse
click	link=Next	EMPTY	clickNextLink
typeAndWait	id=freeSearch	ministry	typeSearch PageChange
type	id=authcode	T75Y5	typeAuth
type	name=firstName	james	typeFirstName
type	name=lastName	bond	typeLastName
click	//ul[@id='ref_67978']/li[5]/a/span	EMPTY	click

Table II: Example of an execution trace file, and of processed lines.

The identifiable tokens by default are: *login, username, email, password, verifyPassword, submit, captcha, auth, search, sort, link, option, checkbox, radio, homeLink, imageLink, nextLink, prevLink, firstLink, lastLink, languageLink, buttonLink, searchResultLink, link, collapse, firstNameInput, lastNameInput, numberInput, input, button*, and *clear*. However, the identifiable patterns can also be overridden and added to via a configuration file (see Section III-D).

The tokens produced by the *LogProcessor* affect the pattern inferring done by *PatternInferrer*. For example, the patterns involved in the inferring of the *Login* pattern are *username, email, login* (these identify username and email inputs, and login inputs or actions, depending on which action the token is appended to), *password* (which identifies a password input), and *submit*, which are used to indicate the closing of a classical form submit action (when *matchSubmit(line)* is true), and thus, the end of a standard pattern. Some patterns are identified to distinguish between proper patterns and other types of action non relevant to the inferring process, such as *verifyPassword*, which can be used to distinguish between a Login form and a Register form, and *searchResultLink*, which prevents search result links from being marked as part of a *Find* pattern. Some exist simply to give better context to the action, like *nextLink* or *lastName*.

F. UI Pattern Inferring (PatternInferrer)

This component is a syntactical analyser, that takes as input the extended execution trace file returned by the *LogProcessor*, runs it against a predefined grammar, and returns the patterns found. The processing rules are formalized in Table III.

If during the process the tool detects the same UI Pattern instance several times, the tool is capable of ignoring the repetitions. Its reasoning is explained in Fig. 5.

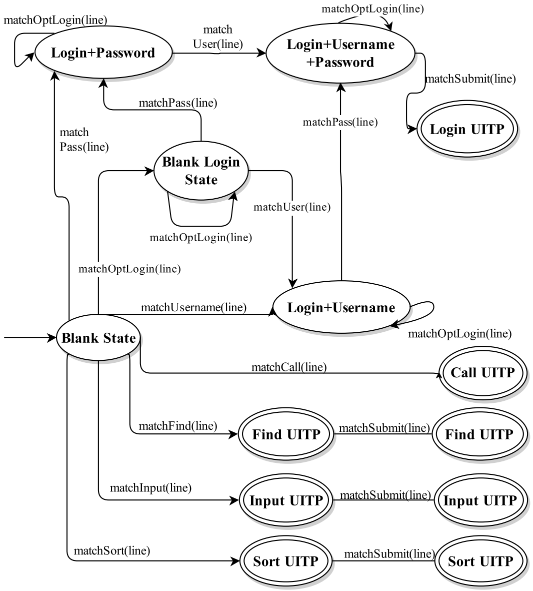

Figure 5: The inferrer's reasoning algorithm, expressed in a finite state machine.

For simplicity's sake, only the valid paths are shown in Fig. 5. All patterns except *Login* can be valid even in the absence of a form submit action; this is done to account for dynamic submission via Javascript. In the case of the *Login*

⟨pattern⟩	⊨	⟨login⟩ | ⟨find⟩ | ⟨sort⟩ | ⟨input⟩ | ⟨call⟩
⟨find⟩	⊨	⟨match-find⟩ ⟨opt-submit⟩
⟨sort⟩	⊨	⟨match-sort⟩ ⟨opt-submit⟩
⟨input⟩	⊨	⟨match-input⟩ ⟨opt-submit⟩
⟨call⟩	⊨	⟨match-link⟩ `PageChange`
⟨login⟩	⊨	⟨opt-login-rec⟩ ⟨match-user-pass⟩ ⟨match-submit⟩
⟨match-user-pass⟩	⊨	⟨match-user⟩ ⟨match-pass⟩ | ⟨match-pass⟩ ⟨match-user⟩
⟨opt-submit⟩	⊨	⟨match-submit⟩ | ϵ
⟨match-submit⟩	⊨	`clickFormSubmit` | `clickButton`
⟨match-find⟩	⊨	⟨opt-find-rec⟩ ⟨search-item⟩ ⟨opt-find-rec⟩
⟨match-sort⟩	⊨	`clickSort` | `selectSort`
⟨match-input⟩	⊨	`clickInput` | `typeInput` | `typeNumberInput` | `typeFirstNameInput` | `typeLastNameInput`
⟨match-user⟩	⊨	⟨opt-login-rec⟩ ⟨user⟩ ⟨opt-login-rec⟩
⟨match-pass⟩	⊨	⟨opt-login-rec⟩ ⟨pass⟩ ⟨opt-login-rec⟩
⟨user⟩	⊨	`typeUsername` | `typeEmail` | `typeLogin`
⟨pass⟩	⊨	`typePassword`
⟨match-opt-login⟩	⊨	`clickLogin PageChange` | `typeAuth` | `typeCaptcha` | `clickOption` | `clickRadio`
⟨match-link⟩	⊨	`clickLink` | `clickHomeLink` | `clickImageLink` | `clickNextLink` | `clickPrevLink` | `clickFirstLink` | `clickLastLink` | `clickLanguageLink` | `clickButtonLink` | `clickSearchResultLink`
⟨opt-find-rec⟩	⊨	⟨opt-find-rec⟩ | ⟨opt-search⟩ | ⟨search-item⟩ | ϵ
⟨opt-search⟩	⊨	`clickSearch`
⟨search-item⟩	⊨	`typeSearch` | `selectSearch`
⟨opt-login-rec⟩	⊨	⟨opt-login-rec⟩ | ⟨match-opt-login⟩ | ϵ

Table III: Default grammar used by *LogProcessor* to tokenize the execution trace file (ϵ indicates an empty string).

pattern, there can only be one password and one username or email record; this is done to distinguish login forms from register forms and password/email change forms.

The previous figure (Fig. 5) does not account for the *Menu* and *Master Detail* patterns; as mentioned before, these patterns are identified through HTML analysis and passed directly from the *WebsiteExplorer* to the *PatternInferrer* to write. This grammar only deals with the patterns identifiable through the execution trace file.

After the file is processed, an XMI file is produced containing all the pattern occurrences found, and the values used during the exploration assigned to each variable, as per Formula 1. One can argue that generating test cases from a model obtained by a reverse engineering process will not find errors. But in this case, we produce part of a PARADIGM model with the identified UI Test Patterns and, afterwards, this model is validated and completed manually by the tester. In particular, he defines the checks to perform within each test strategy (e.g., Stay on the same page), the preconditions (e.g., login=="guest") and possibly other test input data besides the ones used during the exploration.

An example of an execution trace file extract, from which a Login pattern can be inferred, can be seen in Table IV.

Action	Target	Value	Processing Return
clickAndWait	//a[@class=active and @href=... and contains(text(),'Login')]	EMPTY	clickLogin pageChange
type	//input[@id=email and @name=email and @type=text]	login@en.pt	typeEmail
type	//input[@id=password and @type=password and @name=password]	pass	typePassword
click	//input[@id=save_login and @name=save_login and @type=checkbox]	EMPTY	clickLogin
clickAndWait	//input[@class=btn_small grey and @name=commit and @type=submit]	EMPTY	clickSubmit pageChange

Table IV: Execution trace example from which a Login pattern can be inferred.

IV. Evaluation

In order to evaluate the developed reverse engineering approach, some experiments were performed. They aimed to answer the following research questions.

A. Research Questions

R1) Is it possible to automatically infer UI Patterns from a web application?

R2) Is it possible to improve the results provided by the previous RE tool?

B. Evaluation Results

The RE tool was initially experimented iteratively over a set of web applications, with the goal of refining and fine-tuning the inferring grammars used to find UI Patterns.

Afterwards, the RE tool was used to detect UI Patterns in several publicly known and widely used web applications. This time, the purpose was to evaluate the RE tool, i.e., determine which UI pattern occurrences the tool was able to detect in each application execution trace (ET) and compare them to the patterns that really exist in each trace.

Five applications were chosen from the most popular Websites[1]: Amazon, Wikipedia, Ebay, Youtube, Facebook and Yahoo.

The results of the experiments are presented in Table V. It shows the number of instances of each UI pattern that exist in the execution traces, the ones that the tools correctly found (true positives) and the ones that the tools mistakenly found (false positives).

As we can see in Table V, the reverse engineering tool found few false positives, most related to the *Menu* UI Pattern. In addition it is worth a mention that the tool found

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

Current Tool				
Pattern	**Present in ET**	**True Positive**	**False Negative**	**False Positive**
Login	4	4	0	0
Find	15	13	2	0
Sort	1	1	0	0
Input	29	28	1	0
Call	249	235	14	0
Menu	212	216	0	4
MasterDetail	58	46	12	0
Total	**568 (100%)**	**543 (95.59%)**	**29 (5.1%)**	**4 (0.7%)**

Table V: Evaluation set results from the current tool.

Listing 1: An example of a generated model with identified patterns

```
<?xml version=”1.0” encoding=”UTF–8”?>
<Paradigm:Model xmi:version=”2.0” xmlns:xmi=”http://
      www.omg.org/XMI” xmlns:xsi=”http://www.w3.org
      /2001/XMLSchema–instance” xmlns:Paradigm=”http://
      www.example.org/Paradigm” title=”default”>
<nodes xsi:type=”Paradigm:Init” outgoingLinks=”//
       @relations.0”/>
  <nodes xsi:type=”Paradigm:Login” name=”Login” number
        =”1” incomingLinks=”//@relations.0”
        outgoingLinks=”//@relations.1”>
    <fields name=”button” id=”button_”/>
    <fields name=”username” id=”username_”/>
    <fields name=”password” id=”password_”/>
  </nodes>
   ...
  <nodes xsi:type=”Paradigm:End” incomingLinks=”//
        @relations.4”/>
  <relations xsi:type=”Paradigm:Sequence” label=”>>”
        source=”//@nodes.1” destination=”//@nodes.2”/>
  <relations xsi:type=”Paradigm:Sequence” label=”>>”
        source=”//@nodes.2” destination=”//@nodes.0”/>
   ...
</Paradigm:Model>
```

95.59% of the patterns present in the automatically explored execution traces.

However, the case study does not mention how many patterns were present in the web applications that were not visited, which we have not verified (false negatives). Only one path produced by the application was considered for each web application, and all the paths were ran using the standard configuration, excluding any login credentials included.

A simplified version of one of the generated graphical PARADIGM models can be seen in Fig. 6, and the corresponding XMI model can be seen in Listing 1.

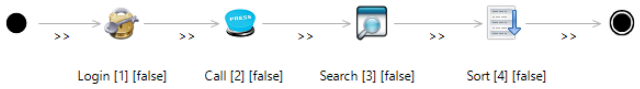

Figure 6: A simplified example of a generated model.

V. 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” [13]. There are four methods of applying reverse engineering to a system: the dynamic method, in which the data is 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 [14], 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 SVNor Git[15]. 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 [16].

There are plenty of approaches that extract information from web applications [17], [18], [19], [20], [21]. ReGUI [22], [23] is a dynamic reverse engineering tool made to reduce the effort of modelling 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 [24].

There are also plenty of approaches that explore web applications for analysis and processing. Benedikt *et al.* introduced a framework called VeriWeb [25] that discovers and explores automatically Web-site execution paths that can be followed by a user in a web application. Bernardi *et al.* [26] present 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 [27] that abstracts the Document Object Model (DOM) into a state model, and from the model derives test cases. Crawljax [28] 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 [29] 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 [30]. Artzi *et al.* developed a tool called Artemis [31] 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 [32] 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 [33] 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 [34], [35] is a tool that analyses 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 various sources [36], [4], [37], [38]. Lin and Landay's approach [39] 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 [40] 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 [7], [5]. 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 [41] presented an approach to extract relations and patterns for the same data spread through many different formats. Chang [42] 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 identify UI Patterns within existing web applications, by extracting information from an execution trace and afterwards inferring the existing UI Patterns. The tool then identifies the corresponding UI Test Patterns to test the identified UI Patterns and gathers some information for their configurations. The result is then exported into a PARADIGM model to be completed in the modelling environment, PARADIGM-ME. Afterwards, the model can be used to generate test cases that are executed on the web application under test. 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 PARADIGM model.

The evaluation of the overall approach was conducted in several worldwide used web applications. The result was quite satisfactory, as the reverse engineering 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), and was able to translate those occurrences into valid PARADIGM files, useful to testers.

The tool was able to improve the previous reverse engineering tool by increasing the true positive percentage, lowering the false positive percentage, and broadening the range of identifiable patterns, while at the same time negating the need for user interaction in the process of identifying UI Test Patterns from a web application, and thus answers the research questions.

Despite the satisfactory results obtained, the approach still needs some improvement. The tool does not handle dynamic pages very well. As such, the features planned for future versions of the reverse engineering tool include 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. In addition, we plan to define different exploration algorithms and compare their effectiveness in finding the existing UI patterns in web applications.

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] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa, "Rich internet applications," *Internet Computing, IEEE*, vol. 14, no. 3, pp. 9–12, 2010.

[4] 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.

[5] 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.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*. ACM, 1999, pp. 285–294.

[7] 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.

[8] 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.

[9] R. M. L. M. Moreira and A. C. R. Paiva, "A gui modeling dsl for pattern-based gui testing - paradigm," in *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2014)*, 2014, pp. 126 – 135.

[10] T. Monteiro and A. C. R. Paiva, "Pattern based gui testing modeling environment." in *ICST Workshops*. IEEE, 2013, pp. 140–143. [Online]. Available: http://dblp.uni-trier.de/db/conf/icst/icstw2013.html#MonteiroP13

[11] A. C. R. P. Liliana Vilela, "Paradigm-cov - a multimensional test coverage analysis tool," in *CISTI 2014 - 9ł Conferencia Ibrica de Sistemas y Tecnologas de Informacin, Barcelona*, 2014.

[12] R. M. L. M. Moreira and A. C. R. Paiva, "Towards a Pattern Language for Model-Based GUI Testing," in *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP 2014)*, 2014.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] A. Grilo, A. Paiva, and J. Faria, "Reverse engineering of gui models for testing," in *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*, June 2010, pp. 1–6.

[21] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes, "Reverse engineered formal models for gui testing." in *FMICS*, ser. Lecture Notes in Computer Science, S. Leue and P. Merino, Eds., vol. 4916. Springer, 2007, pp. 218–233. [Online]. Available: http://dblp.uni-trier.de/db/conf/fmics/fmics2007.html#PaivaFM07

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

[38] 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.

[39] 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.

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

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

[42] 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.