FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Reverse Engineering of Interaction Patterns

Clara Sacramento

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Paiva (PhD)

July 8, 2014

Reverse Engineering of Interaction Patterns

Clara Sacramento

Mestrado Integrado em Engenharia Informática e Computação

July 8, 2014

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.

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 dissertation 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, analyzes the gathered information, tokenizes it, and infers the existing UI patterns via syntactical analyzing. After 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.

First the theme developed during the course of the dissertation is introduced, starting by defining the context and issue at hand and describing the goals of this dissertation. Afterwards we present a literary review on reverse engineering approaches, approaches that infer patterns from Web applications, and approaches that extract models from applications. We explain the approach in detail, present a case study, and lastly, we present the conclusions taken from the work.

Resumo

Grande parte do esforço despendido em testes baseados num modelo está relacionado com a criação do modelo. Além disso, o modelo, mesmo sendo uma ferramenta poderosa de abstração, pode ter erros conceptuais introduzidos pelo testador. Esses problemas podem ser reduzidos ao gerar esses modelos automaticamente. Esta dissertação apresenta uma abordagem de engenharia reversa dinâmica que pretende extrair parte do modelo de uma aplicação Web a testar, diretamente da sua interface gráfica, através da identificação de padrões de interface de utilizador (*UI Patterns*). Esta abordagem de engenharia reversa explora automaticamente qualquer aplicação Web, guarda informação relacionada com a interação, analisa a informação guardada, analisa-a lexicalmente, e infere padrões através de análise sintáctica. Após ser complementado com informação adicional e validado, o modelo resultante serve de entrada para a abordagem de Teste de GUIs baseado em Padrões (PBGT), para testar a aplicação Web sobre análise.

Primeiro o tema desenvolvido no decorrer da dissertação é introduzido, começando por definir o seu contexto, motivação e objectivos. Depois é apresentada uma revisão bibliográfica de abordagens que usam engenharia reversa, abordagens que inferem padrões de aplicações Web, e abordagens que extraem modelos de aplicações. A abordagem desenvolvida é apresentada em detalhe, é apresentado um caso de estudo, e finalmente são apresentadas as conclusões do trabalho realizado.

Acknowledgements

First of all, I would like to thank my supervisor, Ana Paiva, from the Faculty of Engineering of University of Porto, for her guidance, insight, encouragement and support during the whole masters. Our weekly moments of discussion were indispensable for the success of this work, and specially for keeping me in the right track and focused on what was important (and not what it would be cool to do).

I would also like to thank all my teachers, from the first grade to my last year in college, as without them I wouldn't have had the necessary skills to accomplish this.

My friends' friendship has been very important to me and so I thank them for being there for me every step of the way. I would like to specially thank the members of NIAEFEUP, with a very special mention for Pedro, for all the talking, all the comfort and teasing, all the help, and the kind words in the darkest moments. Thank you for all the great moments we have spent together, and for all that we have accomplished together.

A special thank to my parents, Maria and Adelino, without whom I would not be the person I am today and would have not achieved all the great things I have in my life. Thank you for all of your love, your concern and specially your loving support. Thank you for being there for me around the clock.

And finally, a special thank to my boyfriend, Diogo. Thank you for your insight in some of my assignments. Thank you for all of your love, your friendship, the comfort you provided me, your understanding, your patience and for every time you have teased me along the years.

“"When someone says: ’I want a programming language in which I need only say what I wish done’, give him a lollipop."

Alan J. Perlis

Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 Motivation and Objectives . . . 2
1.3 Structure of the Report . . . 3

2 State-of-the-Art **5**

2.1 Introduction . . . 5
2.2 Concepts . . . 5
2.2.1 Reverse Engineering . . . 5
2.2.2 Model-based Testing . . . 6
2.2.3 UI (User Interface) Patterns . . . 7
2.3 Related Work . . . 7
2.3.1 Extraction of Information from Execution Traces . . . 7
2.3.2 Extraction of Information from Applications . . . 8
2.3.3 Model Extraction from Applications via Reverse Engineering . . . 9
2.3.4 Inferring Patterns from Web applications . . . 9
2.4 UI Patterns . . . 10
2.4.1 Capture-Replay Tools . . . 10
2.5 Chapter Conclusions . . . 11

3 RE-TOOL **13**

3.1 PBGT Overview . . . 13
3.1.1 Architecture . . . 13
3.1.2 PARADIGM DSL . . . 14
3.1.3 Produced Models . . . 17
3.2 Reverse Engineering Approach . . . 18
3.2.1 Previous Tool . . . 18
3.2.2 Current Tool . . . 20
3.3 Chapter Conclusions . . . 29

4 Case Study **31**

4.1 Evaluation . . . 31
4.1.1 Research Questions . . . 31
4.1.2 Evaluation Results . . . 31
4.2 Chapter Conclusions . . . 33

5 Conclusions **35**
5.1 Goal Satisfaction 35
5.2 Future Work 36

References **37**

A Appendix **43**

List of Figures

2.1 An explanation of the model-based testing process. 6

3.1 An overview of the PBGT project. 14
3.2 PARADIGM DSL Meta-model. 15
3.3 PARADIGM syntax. 16
3.4 Structure of the PARADIGM-RE tool 18
3.5 The architecture of the approach. 21
3.6 The inferrer's reasoning algorithm, expressed in a finite state machine. 26
3.7 A simplified example of a generated model. 29

LIST OF FIGURES

List of Tables

3.1 An example of execution traces produced on the Amazon.com website, extracted using Selenium IDE. 19
3.2 Abbreviated example of an execution trace file. 20
3.3 Example of an execution trace file, and of processed lines. 25
3.4 Default grammar used by LogProcessor to tokenize the execution trace file (ε indicates an empty string). 27
3.5 Execution trace example from which a Login pattern can be inferred. 28

4.1 Evaluation set results from the previous and current tool, given the same input. . 32
4.2 Evaluation set results from the previous tool, taken from [NPF14]. 33

A.1 A full history file example. 44

LIST OF TABLES

List of Algorithms

1 Pseudo-code algorithm to explore a page. 22

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
AUA	Application Under Analysis
CIO	Concrete Interaction Objects
DSL	Domain Specific Language
EFG	Event Flow Graph
FEUP	Faculdade de Engenharia da Universidade do Porto
FSM	Finite State Machine
HTML	HyperText Markup Language
GUI	Graphical User Interface
MBT	Model-Based Testing
PBGT	Pattern-based GUI Testing
RIA	Rich Internet Applications
SUT	System Under Testing
TDD	Test-Driven Development
UI	User Interface
UML	Unified Modeling Language
XML	eXtensible Markup Language

Chapter 1

Introduction

1.1 Context

Web applications are getting more and more important. Due to their stability and security against losing data, there is a growing trend to move applications towards the Web, with the most notorious examples being Google's mail and office software applications. Web applications can now handle tasks that before could only be performed by desktop applications [G+05], like editing images or creating spreadsheet documents.

Despite the relevance that Web applications have in the community, they still suffer from a lack of standards and conventions [CL02], unlike desktop and mobile applications. This means that the same task can be implemented in many different ways, which makes automated testing difficult to accomplish and inhibits reuse of testing code. For instance, authentication (*login*) failure usually triggers the appearance of an error message, but some implementations simply erase the inserted data, with no error message visible.

GUIs *(Graphical User Interfaces)* of all kinds are populated with recurring behaviors that vary slightly; examples being authentication via *login/password* pair and content search. These behaviors (patterns) are called User Interface (UI) patterns [VWVDVE01] and are recurring solutions to 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. 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 [DJK+99].

Testing is an increasingly important part of any development process as it is essential to improve the trust in the quality of software. Thus, there is an ever increasing need to develop new techniques in this field. This necessity is increased by the constant improvement of development techniques.

When it comes to GUI testing, there are some applicable testing techniques [Mem02]: **capture replay**, which requires a functional GUI; **unit testing**, which requires the manual implementation of the tests and may involve too much work in order to test all of the functionalities; **random input testing**, which is good at finding situations where the system crashes, but is not as good at finding other kinds of errors; and **model-based testing**, which enables automatic test case generation and execution, even though it requires a formal model in order to generate the test cases.

1.2 Motivation and Objectives

The focus of this dissertation is a component of a research project named PBGT (*Pattern-based GUI Testing*) [MPM13]. The main goal of of this project is to improve current model-based GUI testing methods and tools, contributing to conceive an effectively applicable testing approach in industry and to contribute to the construction of higher quality GUIs and software systems. One of the problems to overcome when implementing a model-based GUI testing approach is the time required to construct the model and the test case explosion problem. Choosing the right abstraction level of the model, extracting part of that model by a reverse engineering process and focusing the test cases to cover common recurrent behavior seems to be a way to solve these problems.

The proposal aims to continue the work done on PARADIGM-RE, a component of the PBGT process responsible for extracting part of the Web application model from the Web application itself via reverse engineering, by developing a completely new tool to extract a partial model of a Web application via reverse engineering.

The previous tool [NPCF13, NPF14] is a dynamic reverse engineering approach that extracts User Interface (UI) Patterns from existent Web applications. Its functioning can be summed up like this:

- First, the user interacts with the Web application, navigating it to the best of his ability while the tool records information related to the interaction (user actions and parameters, and for each page visited, its HTML source and URL);
- Second, the collected information is analyzed in order to calculate several metrics (e.g., the differences between subsequent HTML pages);
- Lastly, the existent UI Patterns are inferred from the overall information calculated based on a set of heuristic rules.

This approach was evaluated on several widely used Web applications and the results were deemed satisfactory, since the tool identified most of the occurring patterns and their location on the page. However, there are some patterns the tool doesn't identify, such as the Menu pattern, and the heuristics are considered to be still in an incipient state. The tool's major weakness is that it requires a user to interact with a Web application, a process that can quickly become morose and time-consuming.

As stated before, this dissertation aims to develop a new tool for extracting a partial model from a Web aplication meant to be tested. The major goals for this dissertation were to improve the existing reverse engineering and pattern inferring process, remove the need for user interaction, and automate the model construction; or in other words, 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. This is meant to speed up model construction, and mitigate the number of errors introduced into the model.

1.3 Structure of the Report

This document is structured into four main chapters. In this first section, Chapter 1, we start by introducing the theme to be developed during the course of the dissertation, define the context, motivations, goals of this dissertation and the issue at hand. Chapter 2 introduces essential concepts to understand the problems with which this document deals, and presents the state of the art on reverse engineering. Chapter 3 presents the developed tool, RE-TOOL, focusing on its architecture and functionalities and on some of the challenges which needed to be tackled during the development. Chapter 4 presents a case study. Finally, Chapter 5 presents some conclusions about this research work, along with the limitations of the implementation.

Chapter 2

State-of-the-Art

In this chapter we begin by making an in-depth presentation of essential concepts. This will be followed by a literature and state-of-the-art review in the fields of reverse engineering approaches that deal with Web applications, and approaches that infer patterns, in an effort to present state of the art and related work.

2.1 Introduction

In order to find the best approach to this problem, some research on already existing methodologies and concepts was needed. First an overview for the general categories researched is given, followed by the actual state of the art found, divided by relevant subcategories.

2.2 Concepts

2.2.1 Reverse Engineering

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" [CC+90].

There are two types of reverse engineering: static and dynamic, depending on whether the model is extracted from the source code or from the program in execution, respectively [Sys99]. Both approaches follow the same three main steps: *collect* the data, *analyse* it and *represent* it in a legible way, and both allow obtaining information about control and data flow [PRW03].

2.2.1.1 Static Analysis

Static reverse engineering tries to extract information about an application through its source code or through its byte code. Static reverse engineering techniques may be useful during the development of a software system as a way of ensuring the correctness of the implementation or as a way of being aware of the current stage of the development [ST00].

A common technique useful for static analysis is code parsers based on grammars, as they are useful for code recognition. Apart from this, static analysis usually uses techniques of fact extraction, fact aggregation and querying [TB+09, TV08].

2.2.1.2 Dynamic Analysis

Dynamic approaches extract information from the Application Under Analysis (AUA) in runtime, without access to the source code. Unlike the static techniques, dynamic approaches are able to extract information about concurrent behaviour, code coverage and memory management [Sys99].

2.2.1.3 Other Methods

Besides the previously mentioned methods, there is also the hybrid method, which combines static and dynamic analysis, and the historical method, which includes historic information to see the evolution of the software system [CDPC11].

2.2.2 Model-based Testing

Model-based testing is the application of model-based design for designing, and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a *System Under Test* (SUT), or to represent testing strategies and a test environment. The process is illustrated in Figure 2.1.

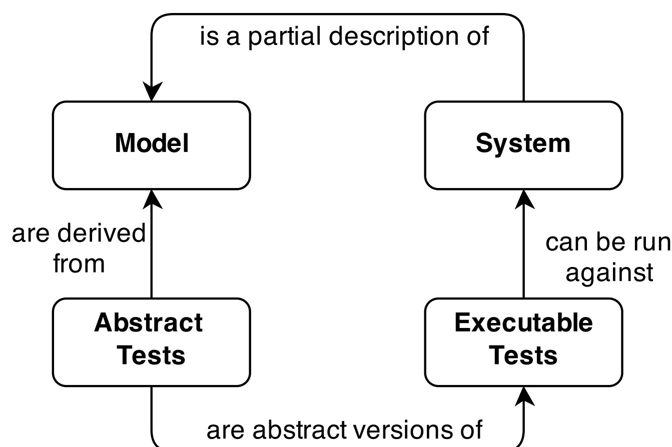

Figure 2.1: An explanation of the model-based testing process.

A model describing a SUT is usually an abstract, partial presentation of the SUT's desired behavior. Test cases derived from such a model are functional tests on the same level of abstraction as the model. These test cases are collectively known as an *abstract test suite*. An abstract test suite cannot be directly executed against an SUT because the suite is on the wrong level of abstraction, so an *executable test suite* needs to be derived from a corresponding abstract test suite.

The executable test suite can communicate directly with the system under test. This is achieved by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing environments, models contain enough information to generate executable test suites directly; in others, elements in the abstract test suite must be mapped to specific statements or method calls in the software to create a concrete test suite. This is called solving the "mapping problem" [AO08]. In the case of online testing, abstract test suites exist only conceptually but not as explicit artifacts.

Tests can be derived from models in different ways. Because testing is usually experimental and based on heuristics, there is no known single best approach for test derivation. It is common to consolidate all test derivation related parameters into a package that is often known as "test requirements", "test purpose" or even "use case(s)" [AO08]. This package can contain information about those parts of a model that should be focused on, or the conditions for finishing testing (test stopping criteria).

Because test suites are derived from models and not from source code, model-based testing is usually seen as a form of *black-box testing* (testing method in which the internal structure/design/implementation of the item being tested is not known to the tester).

2.2.3 UI (User Interface) Patterns

UI patterns (or *interface design* patterns) are ways of capturing common structure without being too concrete on the details, which gives developers and interface designers the flexibility to be creative [Tid10]. They are meant to be standard reference points, recurring solutions that solve common design problems.

2.3 Related Work

2.3.1 Extraction of Information from Execution Traces

Plenty of approaches that extract information from execution traces have been found.
Elbaum [EKR03] presents a testing approach that utilizes data captured in user sessions to create test cases. Duarte, Kramer and Uchitel defined an approach for behavior model extraction which combines static and dynamic information [DKU06]. Sampath *et al.* developed an approach for achieving scalable user-session-based testing of web applications, that applies concept analysis for clustering user sessions and a set of heuristics for test case selection [SSG+07]. TraceServer [AA11] is an extension of the Java PathFinder model checking tool [1] which collects and analyzes

[1] http://babelfish.arc.nasa.gov/trac/jpf

execution traces. jRapture [SCFP00] is a technique and a tool for capture and replay of Java program executions. ReGUI [CMPPF11, CMPPF12] is a dynamic reverse engineering tool made to reduce the effort of modeling the structure and behavior of a software application GUI. Fischer *et al.* developed a methodology that analyzes and compares execution traces of different versions of a software system to provide insights into its evolution, named EvoTrace [FOGG05]. Amalfitano's approach [AFT10] generates test cases from execution traces to help testing from Rich Internet Applications (RIAs), with the execution traces being obtained from user sessions and crawling the application.

2.3.2 Extraction of Information from Applications

The following approaches extract information from Web, desktop and mobile applications for analysis, processing and testing.
Ricca and Tonella's ReWeb [RT01] 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 [BFG02] that discovers and explores automatically Web-site execution paths that can be followed by a user in a Web application. Di Lucca *et al.*'s approach [DLDP05] integrates WARE [DLFT04], a static analysis tool that generates UML diagrams from a Web application's source code, and WANDA [ADPZ04], a Web application dynamic analysis tool, to identify groups of equivalent built client pages and to enable a better understanding of the application under study. Bernardi *et al.* [BDLD08] presents an approach for the semi-automatic recovery of user-centered conceptual models from existing web aplications, 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 [MTR08] that abstracts the Document Object Model (DOM) into a state model, and from the model derives test cases. Crawljax [Roe10] is a tool that obtains graphical site maps by automatically crawling through a Web application. WebDiff [CVO10] is a tool that searches for cross-browser inconsistencies by analyzing a website's DOM and comparing screenshots obtained in different browsers. Memon presented an end-to-end model-based Web application automated testing approach [Mem07] 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 [MvDR12]. Another approach by Mesbah *et al.*, named FeedEx [FM13] 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. Artzi *et al.* developed a tool called Artemis [ADJ+11] 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 [AFT11] that uses dynamic analysis of a Web application to generate end user documentation, compliant with known

standards and guidelines for software user documentation. Dincturk *et al.* [DCvB+12] proposed a RIA crawling strategy using a statistical model based on the model-based crawling approach introduced in [BVBD+11] to crawl RIAs efficiently. Dallmeier *et al.*'s Webmate [DBOZ12, DBOZ13] 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 user's request. Amalfitano *et al.*'s approach, AndroidRipper [AFT+12], uses ripping to automatically and systematically traverse the app's user interface, generating and executing test cases as new events are encountered.

2.3.3 Model Extraction from Applications via Reverse Engineering

There are plenty of approaches that extract models from desktop and Web applications, even if the extracted models may be different from the models extracted by this application. Memon's GUI Ripping [MBN03] is an approach that reverse engineers a model represented as structures called a GUI forest, event-flowgraphs and an integration tree directly from the executable GUI. Paiva *et al.*'s approach [PFV07, PFM08] reverse engineers structural and behavioural formal models of a GUI application by a dynamic technique, mixing manual and automatic exploration with the goal of diminishing the effort required to construct the model required in a model-based GUI testing process. Sanchez *et al.*'s approach [SRSCGM10] is a model-driven engineering process that performs reverse engineering of the layout of RAD-built GUIs, and produces a GUI model in order to perform actions like adapt the GUI for mobile device screens. Gimblett and Thimbleby's approach [GT10] discovers automatically a model of an interactive system by exploring the system's state space, and uses the produced models for structural usability analysis. Scheetz *et al.* have used a class diagram representation of the system's architecture to generate test cases using an AI planning system [SvMF99]. Moore [Moo96] describes experiences with manual reverse engineering of legacy applications to build a model of the user interface functionality. Silva *et al.*'s GUISurfer [SSG+10] is a framework that extracts behavioural models from the source code of GUI-based applications, in order to test their implementation. Lutteroth's approach [Lut08] is able to recover a higher-level layout representation of a hardcoded GUI using the Auckland Layout Model [ZLSW13]. Belluci *et al.*'s approach [BGPP12] performs reverse engineering of interactive dynamic Web applications into a model-based framework able to describe them at various abstraction levels, with the goal of facilitating the adaptation of the analysed Web applications to other types of interactive devices. Stojanovic *et al.*'s approach [SSV02] is a semi-automated reverse engineering approach to generate shared-understandable metadata of Web applications, in order to migrate the applications to the Semantic Web.

2.3.4 Inferring Patterns from Web 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 this dissertation means to extend [NPCF13, MPFC12]. 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 [Bri99] presented an approach to extract relations and patterns for the same data spread through many different formats. Chang [CHL03] proposes a similar method to discover patterns, by extracting structured data from semi-structured Web documents. Freitag [Fre98] proposed a general-purpose relational learner for information extraction from Web applications.

2.4 UI Patterns

User Interaction (UI) patterns are well-documented in a various number of sources [Tid10, VWVDVE01, SGR+05] and [2]. The patterns already supported (like the Search and Master/Detail patterns) enter the list of most popular patterns, according to the sources found, and if the selection of supported patterns were to be broadened, the pick of the next one(s) would be heavily influenced by the literature. Lin and Landay's approach [LL08] 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 [PWL08] presents UI patterns common in eGovernment applications. In [3] are presented the four best user interface pattern libraries, chosen by the UX Movement community.

2.4.1 Capture-Replay Tools

The execution traces of a Web application, on the client side, are usually captured via a capture-replay tool. Here we present the most popular capture-replay tools used nowadays.

Selenium [4] is an open-source capture/replay tool that captures an user's interaction with a Web application in HTML files. It has multi browser, OS and language support, can be installed server-side and as a Mozilla Firefox add-on, has its own IDE *(Integrated Development Environment)*, and allows recording and playback of tests.

Watir Webdriver [5] is is an open-source (BSD) family of Ruby libraries for automating Web browsers and Web application testing. It has multi browser and OS support, a rich API, and has a functionality for non-tech users: the 'Simple' class. There also exist ports for other programming languages, such as *Watij* (for Java) and *Watin* (.NET).

IBM Rational Functional Tester (RFT) [6] is an automated functional testing and regression testing tool. This software provides automated testing capabilities for functional, regression, GUI, and data-driven testing. Rational Function Tester supports a range of applications, such as .Net, Java, Siebel, SAP, terminal emulator-based applications, PowerBuilder, Ajax, Adobe Flex, and others. It permits storyboard testing, automated testing, data-driven testing, and test scripting.

Sahi [7] is an open-source automation and testing tool for web applications. It allows recording and replaying across browsers, provides different language drivers for writing test scripts, and supports Ajax and highly dynamic web applications.

[2] http://designingWebinterfaces. com/designingWeb-interfaces-12-screen-patterns
[3] http://uxmovement.com/resources/4-best-design-pattern-libraries/
[4] Selenium: http://docs.seleniumhq.org/
[5] Watir: http://watirwebdriver.com/
[6] IBM RFT: http://www-03.ibm.com/software/products/en/functional
[7] Sahi: http://sahi.co.in/

2.5 Chapter Conclusions

In this chapter we gave a brief introduction of the concepts that serve as base to the thesis. We also presented related works on the areas of reverse engineering approaches that extract information from applications in general and Web applications in specific, approaches that extract models automatically from applications, and approaches that infer patterns from applications. Lastly, we presented documentation on UI patterns, and presented various capture/replay applications.

Most of the presented approaches that extract information from execution traces consume execution traces with different content than the ones the developed tool uses; the execution traces range from server logs, to sequences of actions enriched with context information like system state, to sequences of executed code instructions in a run path. Similarly, the presented approaches that extract general information from applications do so for many purposes: for testing (and in this category we test generation, model checking, and use case derivation), analysis, to search for inconsistencies, to better understand a Web application, to generate documentation, and others.

The applications that more closely ressemble the developed approach are the ones that extract models from Web applications, and in that category, the extracted models vary in content. Some extract the layout of the Web application; others extract a directed finite graph, which represents the states of the applications and the actions necessary to switch between states; and some use logical database models. Few model extraction approaches extract models from the GUI of the Web application, and those that do, do so for user interface reengineering (typically of legacy systems).

Finally, there have not been found approaches that extract UI patterns from applications. There are approaches that infer patterns, but the inferrable patterns are other types of patterns: design patterns, textual patterns from databases (related to the field of knowledge and text mining), or Web mining. Additionally, there are few, if none, approaches that mine patterns from GUIs. The closest we've found was an approach that mined usage patterns (common action sequences executed by users) from Web applications, which relates more to Web mining than model extraction.

Chapter 3

RE-TOOL

In this chapter, we begin by giving a brief overview of PBGT, the research project the developed work is inserted in. Afterwards, the previous approach and current tool are broken down in components and explained in detail.

3.1 PBGT Overview

As mentioned before, the focus of this dissertation is the reverse engineering component of a research project named PBGT (*Pattern-based GUI Testing*) [MPM13]. The goal of this project is to develop a model-based GUI testing approach and supporting tool, that can be used in industrial contexts.

3.1.1 Architecture

The PBGT supporting tool has main five components:

1. **PARADIGM**, a DSL (*Domain Specific Language*) to define GUI testing models based on UI Test Patterns [MP14];

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 use that information to generate a test model written in PARADIGM;

3. **PARADIGM-ME**, a modeling and testing environment, built to support the creation of test models [MP13];

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, analyzes their coverage with a coverage analysis tool named **PARADIGM-COV**[LV14], and returns detailed execution reports.

The architecture and workflow of the project is shown in Fig. 3.1.

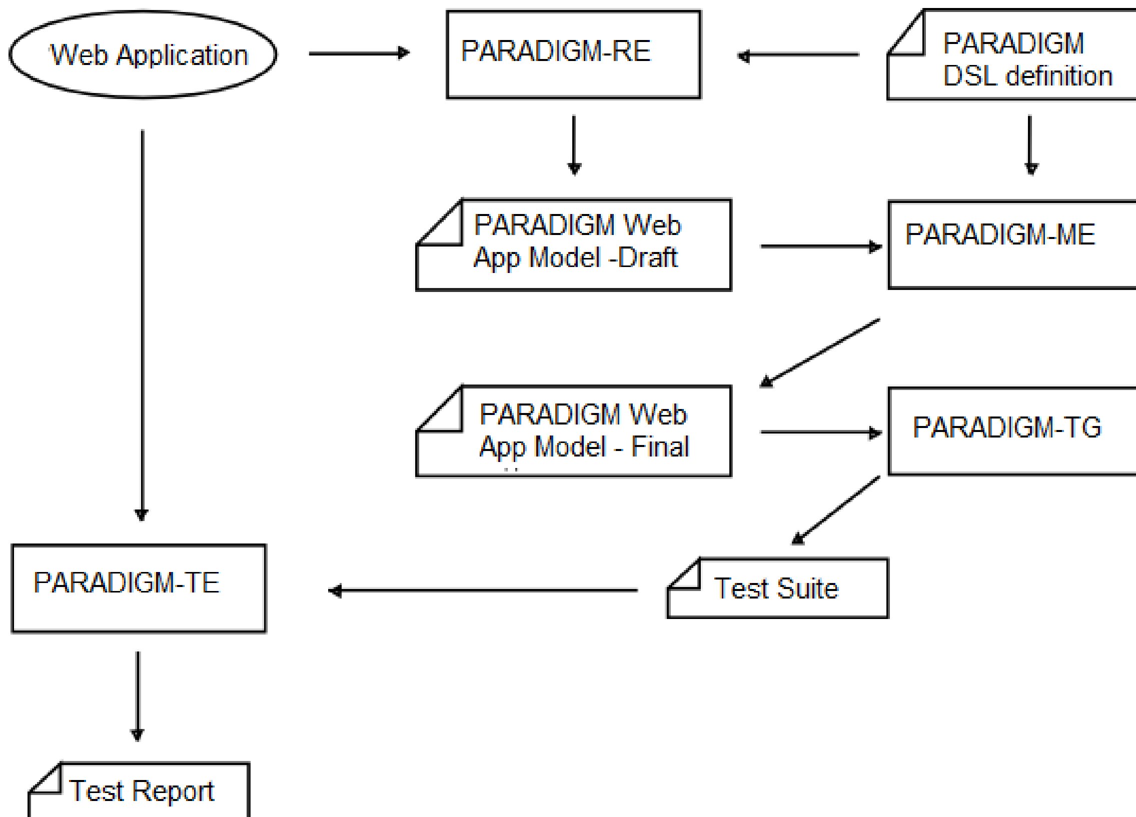

Figure 3.1: An overview of the PBGT project.

3.1.2 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 Figure 3.2.

The PARADIGM language is comprised by elements and connectors [MPM13]. There are four types of elements: *Init* (to mark the beginning of a model), *End* (to mark the termination of a

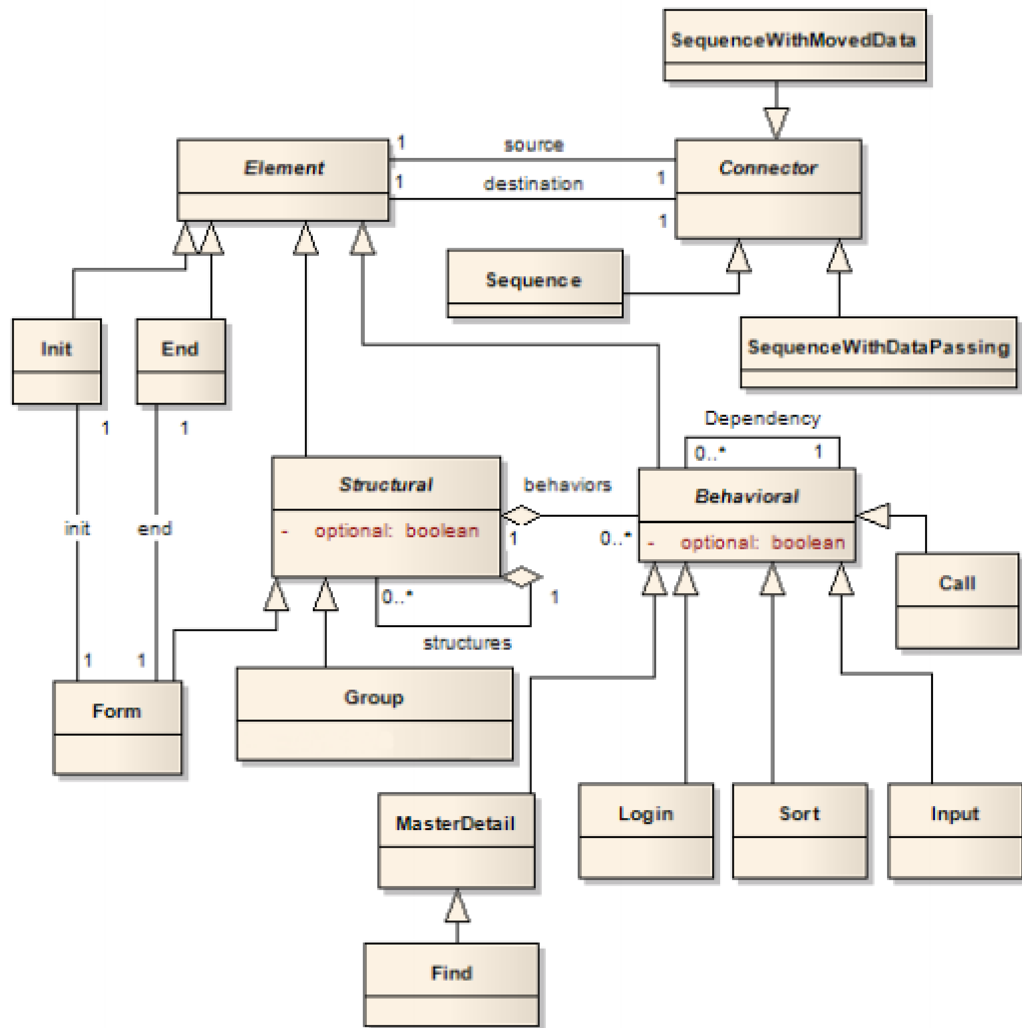

Figure 3.2: PARADIGM DSL Meta-model.

model), *Structural* (to structure the models in different levels of abstraction), and *Behavioral* (UI Test Patterns describing the testing goals).

As models become larger, coping with their growing complexity forces the use of structuring techniques such as different hierarchical levels that allow use one entire model **A** inside another model **B** 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.

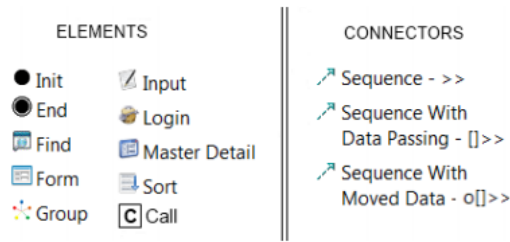

Figure 3.3: PARADIGM syntax.

PARADIGM elements and connectors are described by: (i) an icon/figure to represent the element graphically and (ii) a short label to name the element. The overall syntax of the DSL is illustrated in Figure 3.3. 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 has completed. *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.

3.1.2.1 UITP (User Interface Test Patterns)

A UI Test Pattern defines a test strategy to test a specific UI pattern, which is formally defined by a set of test goals (for later configuration)[MPM13] with the form:

$$< Goal;V;A;C;P > \tag{3.1}$$

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 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. 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, etc.) according to a defined criteria (ascending or descending, alphabetically, etc.).

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.

Input: This pattern is any kind of element in which text can be inserted.

3.1.3 Produced Models

The models produced by the reverse engineering tool PARADIGM-RE consist of a 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 pre-conditions 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.

3.2 Reverse Engineering Approach

3.2.1 Previous Tool

The approach described in this dissertation aims to improve on the previous work [NPCF13] 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, analyzed 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.
This tool identifies interface patterns using Machine Learning inference with the Aleph ILP system [1] running on user interaction execution traces produced using Selenium [2]. It was deemed necessary then to transform the whole process into an iterative one, with the model being updated at every iteration.

This was accomplished in [NPF14], where the tool was extended with a pattern identifying module using heuristics. The structure of the previous tool can be seen in Figure 3.4.

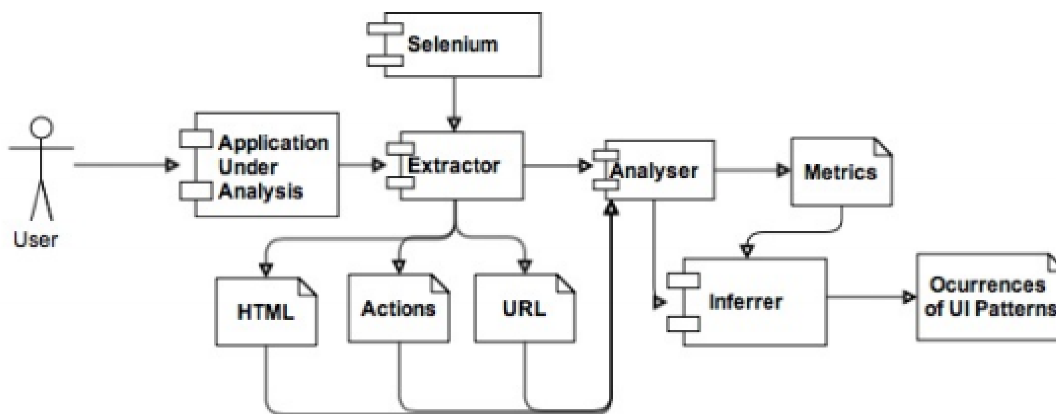

Figure 3.4: Structure of the PARADIGM-RE tool [NPF14]

The user interacts with the Web application, using Selenium to save the actions taken. An example of execution traces saved by Selenium can be seen on table 3.1.

The **extractor** saves the HTML of all pages visited, their URLs, and the actions taken in each page. All that information is passed along to the **analyzer**, whose purpose is to produce metrics like page ratios, differences between consecutive pages, and others, from the data given. Those metrics are passed to the **inferrer**, who runs the heuristics suite, identifies existing patterns, and

[1] Aleph: http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph_toc.html
[2] Selenium: http://docs.seleniumhq.org/

amazon		
actionType	**element**	**text**
open	/	
type	id=twotabsearchtextbox	tablet
clickAndWait	css=input.nav-submit-input	
select	id=sort	label=Most Popular
click	id=pagnNextString	
click	id=pagnPrevString	
clickAndWait	link=Image	
clickAndWait	link=Detail	
click	link=android tablet	
click	css=li.refinementImage >a.. >span.refinementLink	
clickAndWait	css=#result_3 >h3.newaps >a >span.lrg.bold	
clickAndWait	link=Explore similar items	

Table 3.1: An example of execution traces produced on the Amazon.com website, extracted using Selenium IDE[3].

produces a XMI file with the occurrences found. An example of the contents of such a file, with one of each inferrable pattern, can be found on Listing 3.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Paradigm:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3   xmlns:Paradigm="http://www.example.org/Paradigm" title="patterns"/>
4    <nodes xsi:type="Paradigm:Init" name="XInit" number="1"/>
5    <nodes xsi:type="Paradigm:Sort" name="Sort2" number="2"/>
6      <nodes xsi:type="Paradigm:Login" name="Login3" number="3"/>
7      <nodes xsi:type="Paradigm:MasterDetail" name="MasterDetail4" number="4"/>
8      <nodes xsi:type="Paradigm:Input" name="Input5" number="5"/>
9      <nodes xsi:type="Paradigm:Find" name="Find6" number="7"/>
10   <nodes xsi:type="Paradigm:End" name="End" number="8"/>
11 </Paradigm:Model>
```

Listing 3.1: An example of a .paradigm file with identified patterns

This approach was evaluated on several worldwide used Web applications and the results were deemed satisfactory, since the tool identified most of the occurring patterns and their location on the page. However, there are some patterns the tool doesn't identify, such as the Menu pattern, and the high number of false positive results indicate the heuristics are considered to be still in an incipient state.

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 3.2.

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 3.2: Abbreviated example of an execution trace file.

The previous reverse engineering approach identified the *Find*, *Login*, *Sort*, *Input* and *Master-Detail* patterns, and produced a high number of false positives [NPCF13]. Additionally, and as already mentioned, 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.

3.2.2 Current Tool

The work described in this dissertation is a completely new tool, unrelated to the previous tool, but which has the same primary aim as the previous tool (to identify UI patterns in the Application Under Test (AUT)), only with 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 (including patterns that are going to be included in latter versions of the PARADIGM DSL, like the Menu pattern). The architecture of the tool is seen in Figure 3.5. The approach was developed using Java[4], and the means used to interact with

[4]https://www.java.com/

Web pages was Selenium Server[5]. We also used the Apache Commons libraries, namely Apache Commons IO[6] and Apache Commons Lang[7].

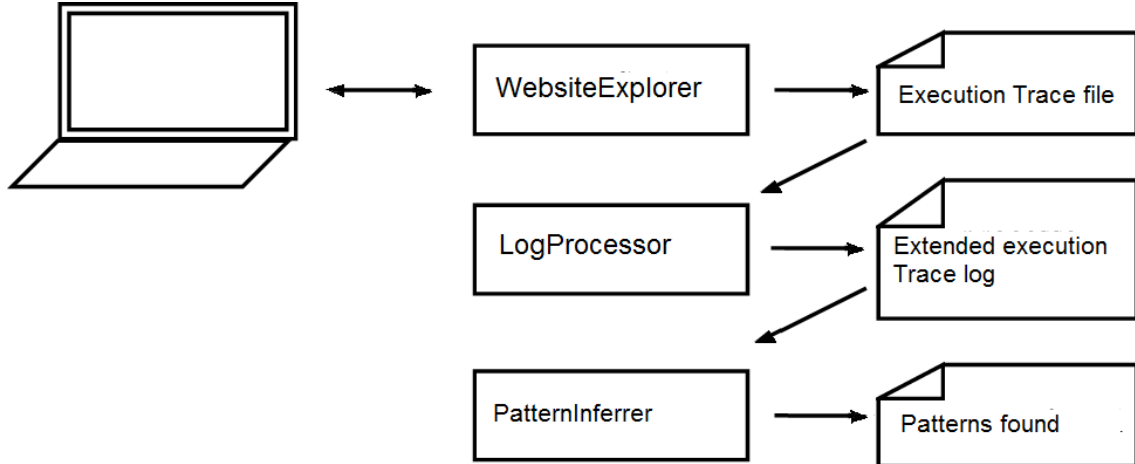

Figure 3.5: The architecture of the approach.

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

WebsiteExplorer loads user configurations (more details on Section 3.2.2.2), interacts automatically with the AUT and produces an execution trace file with the actions taken (see Table 3.2 for an example). **LogProcessor** is a text parser. It analyzes the previous file, parses each line, searches for important keywords and uses them to identify the action contained in the line, and produces an updated execution trace file (see Table 3.3 for an example). Finally, **PatternInferrer** analyzes 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 all of those defined in the PARADIGM language, plus the *Menu* UI pattern, which is going to be included in the PARADIGM DSL in the future.

3.2.2.1 AUT Exploration (WebsiteExplorer)

The interaction process is described in a simplified manner in Algorithm 1. The first thing the algorithm does is load the user-defined configurations file (if it exists). The algorithm itself is limited by two variables: *number_of_actions* and *number_of_redirections*. When any of these variables reaches its respective limit, the exploration process ends. The algorithm searches for the patterns allowed by the configuration (which can be all of them, or a user-defined subset), and for

[5]http://docs.seleniumhq.org/docs/05_selenium_rc.jsp
[6]http://commons.apache.org/proper/commons-io/
[7]http://commons.apache.org/proper/commons-lang/

each interaction it sequentially executes the following actions: search for valid elements that have not been visited yet and store them in a list; choose a random element to visit (if the previous list is not empty); visit the element (the method used to visit the element depends on the type of element and the pattern it belongs to); mark the element as visited; write the action taken to the execution trace file; increment the relevant counters; and begin a new iteration in the resulting page. If at any time the list of valid explorable elements is empty, it means the current path has been fully explored, and the algorithm returns to the base URL to try another path.

```
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
```

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

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 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[8]. 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 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). 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). An example of a produced CSV file can be seen in Table A.1, in Section A.

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 analyzing 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 3.2.2.4).

3.2.2.2 Configuration

To improve results, almost all components used in the website interaction and pattern identifying processes can be loaded to the application via a 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;

[8] XPath: www.w3schools.com/XPath/

redirections: number of redirects to the home page the crawler 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;

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: specify absolute file path for history file;

processedHistoryFilepath: specify absolute file path for processed history file;

patternsFilepath: specify absolute file path 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 app;

tokenizerPatterns: patterns for LogProcessor;

includeChildrenNodesOnInteraction: boolean; on interaction with an input or select element, if children elements should be included in the history file, but not interacted with;

3.2.2.3 Action File Processing (LogProcessor)

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**, and a regex (Regular Expression[9]) 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

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

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 3.2.

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_679781011']/li[5]/a/span	EMPTY	click

Table 3.3: 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 3.2.2.2).

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*. An example of a processed execution trace file (resulting of running LogProcessor on the contents of Table A.1) can be seen in Listing A.2.

3.2.2.4 UI Pattern Inferring (PatternInferrer)

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. The processing rules are formalized in Table 3.4.

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 Figure 3.6.

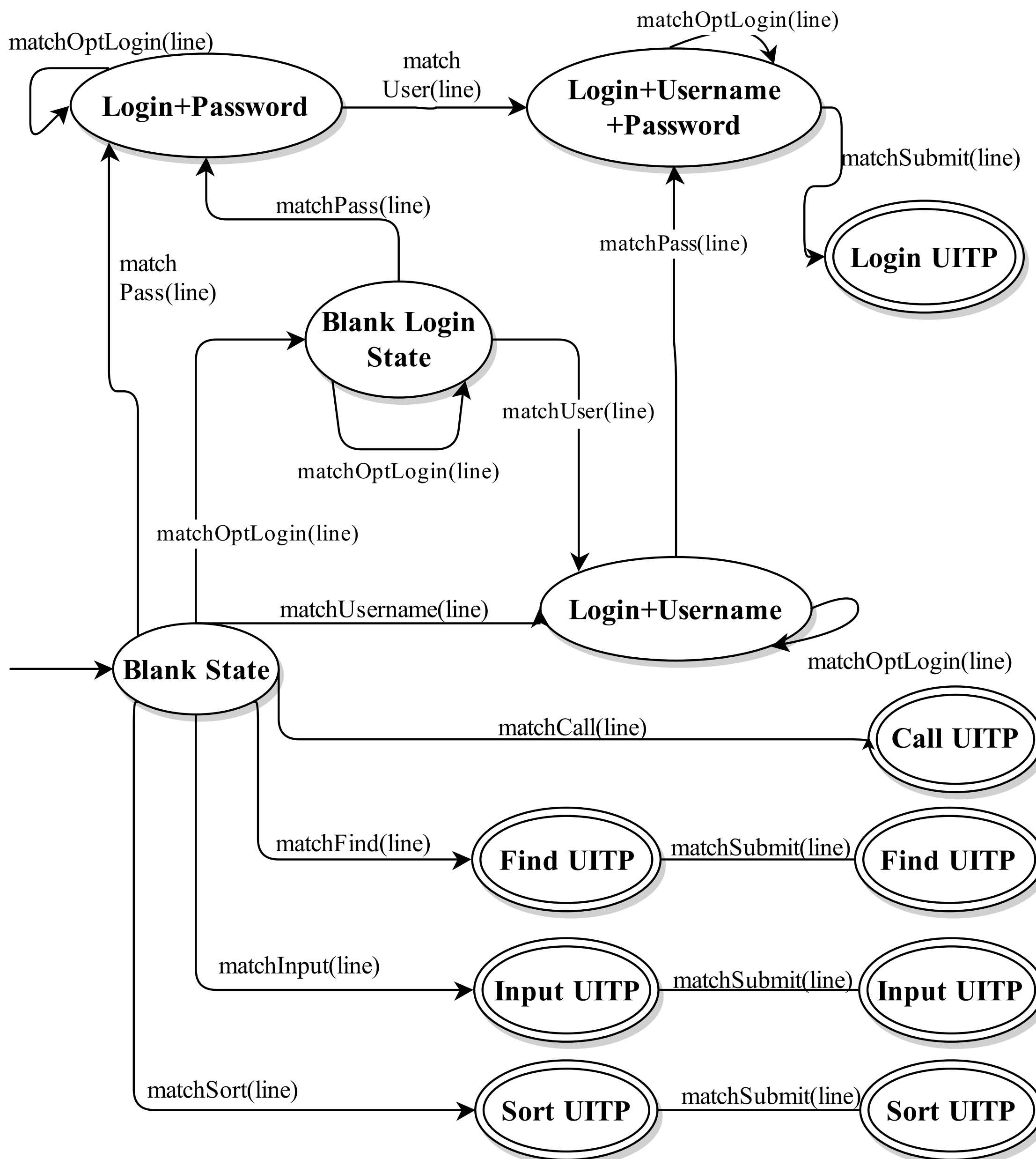

Figure 3.6: The inferrer's reasoning algorithm, expressed in a finite state machine.

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

Table 3.4: Default grammar used by LogProcessor to tokenize the execution trace file (ε indicates an empty string).

For simplicity's sake, only the valid paths are shown in the figure (Fig. 3.6). 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, 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. 3.6) 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 assigned to each variable, as per Formula 3.1.2.1. As mentioned before, the checks to perform within each test strategy and any preconditions must be specified by the tester later on in the PARADIGM-ME tool.

An example of a execution trace file extract from which a Login pattern can be inferred can be seen in Table 3.5, and full example of a PARADIGM file (with inputs being the contents of Table A.1 and Listing A.2) can be seen on Listing A.4.

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

3.2.2.5 PARADIGM files produced by PatternInferrer

The final PARADIGM files produced contain instances of UI Test Patterns, that correspond to all of the patterns discovered during the exploration made by the WebsiteExplorer and contain blank test configurations meant for the tester to fill in. The files are XMI files which follow the PARADIGM DSL (see Section 3.1.2).

Each pattern is stored in a *<nodes>* tag, which has the following attributes: *type* (the type of pattern), *name* (a unique string identifier), *number* (a unique integer identifier), *incomingLinks* (the incoming sequence connector) and *outgoingLinks* (the outgoing sequence connector).

Each *<nodes>* tag has two types of children: *<configurations>* tags, which contain testing configurations, and *<fields>* tags, which identify web elements involved in the testing configurations. Configurations are defined by attributes, and the specification of the test (such as data to insert or element to choose) are defined in *<fields>* child nodes. Some of the attributes that a configuration may have are: *validity* (whether the inserted data is supposed to be valid or not); *check* (the type of check to apply to the result of the operation); *result* (the number of results expected - for testing Find patterns); *master* (identifies the master in a MasterDetail pattern); and *mappingURL* (the URL in which the testing elements are).

A simplified example of a generated model can be seen in Listing A.1, in the Appendix chapter (Section A), and the corresponding graphic model, generated via PARADIGM-ME, can be seen in Figure 3.7.

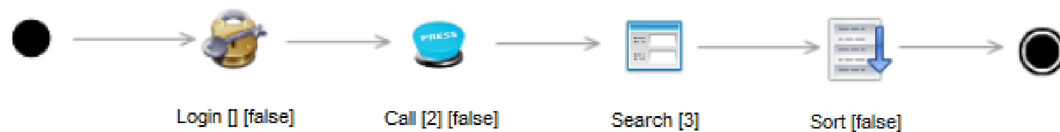

Figure 3.7: A simplified example of a generated model.

3.3 Chapter Conclusions

In this chapter we gave a brief overview of PGBT, explained in detail the previous and current tool, to better highlight their differences and explain their functioning. The previous tool requires user interaction to explore a Web application, analyzes the execution trace file, HTML source and URLs of all pages visited, and infers patterns through a set of heuristic rules; the current tool is fully automatic, requires only the execution trace file, and infers patterns through syntactical analysis of a previously tokenized execution trace file.

Chapter 4

Case Study

In this chapter we introduce a case study, done to validate the developed work and assess its efficacy and compliance of the dissertation goals.

4.1 Evaluation

4.1.1 Research Questions

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

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

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

4.1.2 Evaluation Results

The RE tool was initially tested iteratively over a set of Web applications, with the goal of fine-tuning the inferring grammars used to discover UI Patterns.

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

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 4.1. This table shows the number of instances of each UI pattern that exist in the execution traces, the ones that the tools correctly

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

found and the ones that the tools mistakenly found (false positives). In the case of the previous tool, the Menu and Call patterns weren't included because the tool doesn't identify those patterns.

Previous Tool				
Pattern	**Present in ET**	**True Positive**	**False Negative**	**False Positive**
Find	15	9	6	0
Login	4	3	1	0
Sort	1	1	1	306
Input	29	29	0	6
MasterDetail	58	58	0	22
Total	**107 (100%)**	**100 (93.46%)**	**7 (6.54%)**	**334 (312.15%)**
Current Tool				
Pattern	**Present in ET**	**True Positive**	**False Negative**	**False Positive**
Find	15	13	2	0
Login	4	4	0	0
Sort	1	1	0	0
Input	29	28	1	0
Call	249	235	14	0
Menu	212	212	0	4
MasterDetail	58	46	12	0
Total	**568 (100%)**	**539 (94.89%)**	**29 (5.11%)**	**4 (0.7%)**

Table 4.1: Evaluation set results from the previous and current tool, given the same input.

As we can see in Table 4.1, the current reverse engineering tool found few false positives, most related to the Menu UI Pattern. In addition it is worth to mention that the tool found 95.59% of the patterns present in the automatically explored execution traces. Given the same input, the current tool improved all of the statistics, and specially the rate of false positives.

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. Only one path produced by the application was considered for each Web application, and all the paths were traversed using the standard configuration, excluding any login credentials included (the contents of the standard configuration can be seen in Listing A.3, in Section A).

When comparing the evaluation set results with those proposed in [NPF14] (which can be seen in Table 4.2) we see that the current tool finds less patterns than the previous tool. This can be explained: the previous tool had its exploration guided by a human user, specifically a tester, which knew how to guide the exploration; the current tool is guided by an algorithm, that picks a random element for each successive action. When taking this into account, it is natural that an automatic tool finds less patterns than one which it is being guided by an expert. Even so, what the current tool lacks in terms of discovered patterns, it makes up in its percentage of correctly found patterns, which is better than its predecessor's.

Pattern	Present in ET	Correctly Found	False Positives
Login	9	6	0
Find	24	21	0
Sort	11	4	0
Input	28	26	3
MasterDetail	24	9	9
Total	**99**	**67 (67.7%)**	**12**

Table 4.2: Evaluation set results from the previous tool, taken from [NPF14].

4.2 Chapter Conclusions

In this chapter we presented a case study that showcased the results the developed approach produces. In comparison to the previous tool, we can see that the inferring precision has improved (the true positive and false positive percentages have increased and lowered, respectively), and the number of discovered patterns has lowered slightly for the majority of the identifiable patterns, with a possible reason being detailed.

Case Study

Chapter 5

Conclusions

This dissertation 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. Then the tool 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 modeling 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 Pattern Based GUI Testing project that aims to build a model based testing environment to be used in 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.

5.1 Goal Satisfaction

The evaluation of the overall approach was conducted in several popular 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 aplication, and thus answer the research questions.

5.2 Future Work

Despite the satisfactory results obtained, the approach can still be improved. The tool does not handle dynamic pages very well. As so, the features planned for future versions of the reverse engineering tool should include methods for better Javascript handling.

Additionally, the evaluation done to the tool has shown that, while the pattern inferring precision has improved, the number of discovered patterns has lessened. Future work should include experimenting with different exploration algorithms, to compare their efectiveness in finding the existing UI patterns in web applications. Another way to improve pattern discovery could mean changing the exploration method from a purely random one to a random-like breadth exploration, or any other method to explore the Web application fully instead of stopping at a set number of actions.

References

[AA11] Igor Andjelkovic and Cyrille Artho. Trace server: A tool for storing, querying and analyzing execution traces. In *JPF Workshop, Lawrence, USA*, 2011.

[ADJ+11] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A framework for automated testing of javascript web applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 571–580. IEEE, 2011.

[ADPZ04] Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding web applications through dynamic analysis. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 120–129. IEEE, 2004.

[AFT10] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 274–283. IEEE, 2010.

[AFT11] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio 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*, pages 11–20. IEEE, 2011.

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[BDLD08] Mario Luca Bernardi, Giuseppe A Di Lucca, and Damiano Distante. Reverse engineering of web applications to abstract user-centered conceptual models. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 101–110. IEEE, 2008.

[BFG02] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WW W'2002*. Citeseer, 2002.

[BGPP12] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Claudio Porta. Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms. In *Proceedings of the 2012 ACM International Conference on*

Intelligent User Interfaces, IUI '12, pages 217–226, New York, NY, USA, 2012. ACM.

[Bri99] Sergey Brin. Extracting patterns and relations from the world wide web. In *The World Wide Web and Databases*, pages 172–183. Springer, 1999.

[BVBD+11] Kamara Benjamin, Gregor Von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel Onut. *A strategy for efficient crawling of rich internet applications*. Springer, 2011.

[CC+90] Elliot J Chikofsky, James H Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, 2011.

[CHL03] Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. Automatic information extraction from semi-structured web pages by pattern discovery. *Decision Support Systems*, 35(1):129–147, 2003.

[CL02] Larry L Constantine and Lucy AD Lockwood. Usage-centered engineering for web applications. *Software, IEEE*, 19(2):42–50, 2002.

[CMPPF11] Inês Coimbra Morgado, Ana Paiva, and João Pascoal Faria. Reverse engineering of graphical user interfaces. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 293–298, 2011.

[CMPPF12] Inês Coimbra Morgado, Ana CR Paiva, and João Pascoal Faria. Dynamic reverse engineering of graphical user interfaces. *International Journal On Advances in Software*, 5(3 and 4):224–236, 2012.

[CVO10] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[DBOZ12] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, pages 11–15. ACM, 2012.

[DBOZ13] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. Webmate: Generating test cases for web 2.0. In *Software Quality. Increasing Value in Software and Systems Development*, pages 55–69. Springer, 2013.

[DCvB+12] Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. A statistical approach for efficient crawling of rich internet applications. In *Web Engineering*, pages 362–369. Springer, 2012.

[DJK+99] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294. ACM, 1999.

[DKU06] Lucio Mauro Duarte, Jeff Kramer, and Sebastian Uchitel. Model extraction using context information. In *Model Driven Engineering Languages and Systems*, pages 380–394. Springer, 2006.

[DLDP05] Giuseppe A Di Lucca and Massimiliano Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Web Site Evolution, 2005.(WSE 2005). Seventh IEEE International Symposium on*, pages 87–94. IEEE, 2005.

[DLFT04] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering web applications: the ware approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):71–101, 2004.

[EKR03] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59. IEEE Computer Society, 2003.

[FM13] A Milani Fard and Ali 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*, page 10, 2013.

[FOGG05] Michael Fischer, Johann Oberleitner, Harald Gall, and Thomas Gschwind. System evolution tracking through execution trace analysis. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 237–246. IEEE, 2005.

[Fre98] Dayne Freitag. Information extraction from html: Application of a general machine learning approach. In *AAAI/IAAI*, pages 517–523, 1998.

[G$^+$05] Jesse James Garrett et al. Ajax: A new approach to web applications, 2005.

[GT10] Andy Gimblett and Harold Thimbleby. User interface model discovery: towards a generic approach. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 145–154. ACM, 2010.

[LL08] James Lin and James 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*, pages 1313–1322. ACM, 2008.

[Lut08] Christof Lutteroth. Automated reverse engineering of hard-coded gui layouts. In *Proceedings of the Ninth Conference on Australasian User Interface - Volume 76*, AUIC '08, pages 65–73, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[LV14] Ana C. R. Paiva Liliana Vilela. Paradigm-cov - a multimensional test coverage analysis tool. 2014.

[MBN03] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260, 2003.

[Mem02] Atif M Memon. Gui testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.

[Mem07] Atif M Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[Moo96] Melody M Moore. Rule-based detection for reverse engineering user interfaces. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 42–48. IEEE, 1996.

[MP13] Tiago Monteiro and Ana C. R. Paiva. Pattern based gui testing modeling environment. In *ICST Workshops*, pages 140–143. IEEE, 2013.

[MP14] Rodrigo M. L. M. Moreira and Ana 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)*, pages 126 – 135, 2014.

[MPFC12] Inês Coimbra Morgado, Ana CR Paiva, Joao Pascoal Faria, and Rui Camacho. Gui reverse engineering with machine learning. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, pages 27–31. IEEE, 2012.

[MPM13] Rodrigo MLM Moreira, Ana CR Paiva, and Atif Memon. A pattern-based approach for gui modeling and testing. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 288–297. IEEE, 2013.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130. IEEE, 2008.

[MvDR12] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *Software Engineering, IEEE Transactions on*, 38(1):35–53, 2012.

[NPCF13] Miguel Nabuco, Ana CR Paiva, Rui Camacho, and Joao Pascoal Faria. Inferring ui patterns with inductive logic programming. In *Information Systems and Technologies (CISTI), 2013 8th Iberian Conference on*, pages 1–5. IEEE, 2013.

[NPF14] Miguel Nabuco, Ana CR Paiva, and Joao Pascoal Faria. Inferring user interface patterns from execution traces of web applications. Manuscript submitted for publication, 2014.

[PFM08] Ana CR Paiva, João CP Faria, and Pedro MC Mendes. Reverse engineered formal models for gui testing. In *Formal Methods for Industrial Critical Systems*, pages 218–233. Springer, 2008.

[PFV07] Ana CR Paiva, João CP Faria, and Raul FAM Vidal. Towards the integration of visual and formal models for gui testing. *Electronic Notes in Theoretical Computer Science*, 190(2):99–111, 2007.

[PRW03] Michael J Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *10th Working Conference on Reverse Engineering*, pages 80–89, 2003.

[PWL08] Florence Pontico, Marco Winckler, and Quentin Limbourg. Organizing user interface patterns for e-government applications. In *Engineering Interactive Systems*, pages 601–619. Springer, 2008.

[Roe10] Danny Roest. Automated regression testing of ajax web applications. Master's thesis, Delft University of Technology, February 2010.

[RT01] Filippo Ricca and Paolo Tonella. Understanding and restructuring web sites with reweb. *Multimedia, IEEE*, 8(2):40–51, 2001.

[SCFP00] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. *jRapture: A capture/replay tool for observation-based testing*, volume 25. ACM, 2000.

[SGR+05] Daniel Sinnig, Ashraf Gaffar, Daniel Reichart, Peter Forbrig, and Ahmed Seffah. Patterns in model-based engineering. In *Computer-Aided Design of User Interfaces IV*, pages 197–210. Springer, 2005.

[SRSCGM10] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 147–150. ACM, 2010.

[SSG+07] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web applications. *Software Engineering, IEEE Transactions on*, 33(10):643–658, 2007.

[SSG+10] João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva, and José Creissac 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*, EICS '10, pages 181–186, New York, NY, USA, 2010. ACM.

[SSV02] Nenad Stojanovic, Ljiljana Stojanovic, and Raphael Volz. A reverse engineering approach for migrating data-intensive web sites to the semantic web. In *Intelligent Information Processing*, pages 141–154. Springer, 2002.

[ST00] Tarja Systä and Universitatis Tamperensis. Static and dynamic reverse engineering techniques for java software systems. 2000.

[SvMF99] Michael Scheetz, Anneliese von Mayrhauser, and Robert France. Generating test cases from an oo model with an ai planning system. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 250–259. IEEE, 1999.

[Sys99] Tarja Systä. Dynamic reverse engineering of java software. In *ECOOP Workshops*, pages 174–175, 1999.

[TB+09] Alexandru Telea, Heorhiy Byelas, et al. Querying large c and c++ code bases: the open approach. 2009.

[Tid10] Jenifer Tidwell. *Designing interfaces*. O'Reilly, 2010.

[TV08] Alexandru Telea and Lucian Voinea. An interactive reverse engineering environment for large-scale c++ code. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 67–76. ACM, 2008.

[VWVDVE01] Martijn Van Welie, Gerrit C Van Der Veer, and Anton Eliëns. Patterns as tools for user interface design. In *Tools for Working with Guidelines*, pages 313–324. Springer, 2001.

[ZLSW13] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The auckland layout editor: An improved gui layout specification process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, NY, USA, 2013. ACM.

Appendix A

Appendix

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Paradigm:Model xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:Paradigm="http://www.example.org/Paradigm"
 title="http://www.mcgame.com/" >
  <nodes xsi:type="Paradigm:Init" name="Init" number="0"
  outgoingLinks="//@relations.0"/>

  <nodes xsi:type="Paradigm:Login" name="Login" number="1"
  incomingLinks="//@relations.0" outgoingLinks="//@relations.1">
    <configurations validity="Invalid" check="StayOnSamePage" >
     <inputs field="username" value="login"/>
      <inputs field="password" value="123abc"/>
    </configurations>
    <fields name="username" id="//input[@id=email and @name=email
    and @type=text]"/>
    <fields name="password" id="//input[@id=password and @name=password
    and @type=password]"/>
  </nodes>

  <nodes xsi:type="Paradigm:Call" name="Call" number="1"
  incomingLinks="//@relations.1" outgoingLinks="//@relations.2">
    <configurations />
    <field name="call_0" id="//a[@class=active
    and @href=http://www.mcgame.com/de/account/login
    and contains(text(),’Login’)]"/>
  </nodes>

  <nodes xsi:type="Paradigm:Find" name="Search" number="2"
  incomingLinks="//@relations.2" outgoingLinks="//@relations.3">
    <configurations check="NumberOfResults_more_than"
    result="10" >
      <inputs field="find_0" value="computer"/>
```

open	http://www.mcgame.com/	EMPTY
type	//input[@id=query and @name=query and @type=text]	computer
clickAndWait	//input[@type=submit]	EMPTY
clickAndWait	//a[@class=active and @href=http://www.mcgame.com/de/account/login and contains(text(),’Login’)]	EMPTY
type	//input[@id=email and @name=email and @type=text]	banana
type	//input[@id=password and @name=password and @type=password]	apple
click	//input[@id=save_login and @name=save_login and @type=checkbox]	EMPTY
clickAndWait	//input[@class=btn_small grey and @name=commit and @type=submit]	EMPTY
open	http://www.mcgame.com/	EMPTY
type	//input[@class=alarm_amount and @name=alarm_amount and @type=text]	dress
clickAndWait	//a[@href=http://www.mcgame.com/de/store/game/10210253-watch-dogs-season-pass and contains(text(),’Watch Dogs Season Pass’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/publishers and contains(text(),’Kataloge von Publishern’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/publishers/1104-rokapublish]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de and contains(text(),’Mac Spiele’)]	EMPTY
type	//input[@id=searchtextbox and @name=field-keywords and @title=Searchen and @type=text] banana	
clickAndWait	//input[@class=nav-submit-input and @title=Go and @type=submit]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/pages/about and contains(text(),’Uber McGame’)]	EMPTY
select	//select[@id=sort and @class=sort]	label=Price: Low to
clickAndWait	//a[@href=http://www.mcgame.com/de/store/publishers/1045-activision and contains(text(),’Activision’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/help and contains(text(),’Hilfe & Support’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/publishers/1041-bethesda-softworks and contains(text(),’Bethesda Softworks’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/publishers/1055-capcom and contains(text(),’Capcom’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/mac/core and contains(text(),’Core Games’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/offers and contains(text(),’Angebote’)]	EMPTY
clickAndWait	//a[@href=http://www.mcgame.com/de/store/game/10470030-hitman-absolution-professional-edition and contains(text(),’Hitman Absolution: Professional Edition’)]	EMPTY
select	//select[@id=searchDropbox and @class=twotab and @onchange=this.parentForm.submit();]	label=Strategie
clickAndWait	//a[@class=active and @href=https://www.mcgame.com/de/account/login and contains(text(),’Login’)]	EMPTY
clickAndWait	//a[@id=forgot_password and @href=https://www.mcgame.com/reset/password and contains(text(),’Passwort vergessen?’)]	EMPTY

Table A.1: A full history file example.

```
34      </configurations>
35      <fields name="find_0" id="//input[@id=query and @name=query
36      and @type=text]"/>
37    </nodes>
38
39    <nodes xsi:type="Paradigm:Sort" name="Sort" number="3"
40    incomingLinks="//@relations.3" outgoingLinks="//@relations.4">
41      <configurations check="alphabetically" position="">
42        <inputs field="Price" value="100"/>
43      </configurations>
44      <fields name="Price" id="//select[@id=price]"/>
45    </nodes>
46
47    <nodes xsi:type="Paradigm:End" name="End" number="4" 2
48    incomingLinks="//@relations.4"/>
49
50    <relations xsi:type="Paradigm:Sequence" label=">>"
51    source="//@nodes.0" destination="//@nodes.1"/> 6
52    <relations xsi:type="Paradigm:Sequence" label=">>"
53    source="//@nodes.1" destination="//@nodes.2"/> 8
54    <relations xsi:type="Paradigm:Sequence" label=">>"
55    source="//@nodes.2" destination="//@nodes.3"/> 10
56    <relations xsi:type="Paradigm:Sequence" label=">>"
57    source="//@nodes.3" destination="//@nodes.4"/> 12
58  </Paradigm:Model>
```

Listing A.1: An example of a .paradigm file with identified patterns

```
1   open
2   typeSearch
3   clickSubmit pageChange
4   clickLogin pageChange
5   typeEmail
6   typePassword
7   clickLogin
8   clickSubmit pageChange
9   open
10  type
11  clickLink pageChange
12  clickLink pageChange
13  clickLink pageChange
14  clickLink pageChange
15  typeSearch
16  clickSubmit pageChange
17  clickLink pageChange
18  selectSort
19  clickLink pageChange
20  clickLink pageChange
```

```
21 clickLink pageChange
22 clickLink pageChange
23 clickLink pageChange
24 clickLink pageChange
25 clickLink pageChange
26 selectSubmit
27 clickLogin pageChange
28 clickLink pageChange
```

Listing A.2: The result of LogProcessor executing with the content of Table A.1 as input.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <conf>
3    <!--
4        actions: number of actions the crawler will execute before stopping
5        redirections: number of redirects to the home page the crawler will do
6           before stopping
7        politenessDelay: time to wait (in milliseconds) after each action
8        typedKeywords: list of words to insert in text input elements
9      searchKeywords: regex with keywords that identify search elements
10     sortKeywords: regex with keywords that identify sort elements
11     loginKeywords: regex with keywords that identify login elements
12     generalWordsToExclude: regex with keywords that mark elements that should not
          be
13       accessed
14     menuIdentifiers: list of ids/classes/names that identify menu elements
15     masterIdentifiers: list of ids/classes/names that identify master elements
16     detailIdentifiers: list of ids/classes/names that identify detail elements
17     historyFilepath: specify absolute filepath for history file
18     processedHistoryFilepath: specify absolute filepath for processed history file
19     patternsFilepath: specify absolute filepath for PARADIGM model file
20     patternsToFind: list of patterns that the explorer can search for. If "all"
21       occurs, there will be no restriction.
22     loginConfiguration: credentials for a correct login in the web app.
23     tokenizerPatterns: patterns for LogProcessor
24     includeChildrenNodesOnInteraction: boolean; on interaction with an input or
25       select element, if children elements should be included  in the history file,
26       but not interacted with.
27   -->
28   <actions>50</actions>
29   <redirections>5</redirections>
30   <politenessDelay>500</politenessDelay>
31   <includeChildrenNodesOnInteraction>false</includeChildrenNodesOnInteraction>
32   <searchKeywords>(input|select|textarea)(.*(=q|q(ue)?ry|s(ea)?rch|pesq(uisa)?|
       procura(r)?|busca(dor)?).*)</searchKeywords>
33   <sortKeywords>((input|select|textarea).*sort)</sortKeywords>
34   <loginKeywords>(user(name)?|pass(word)?|e?mail|(sign(ed)?(\\s|_)?(in|out)|log(ged
       )?(\\s|_)?(in|out)))</loginKeywords>
```

```
35  <generalWordsToExclude>(buy|sell|mailto|add(\\s|_)?to(\\s|_)?cart|checkout)</
        generalWordsToExclude>
36  <typedKeywords>
37        <item>curtains</item>
38        <item>coffee</item>
39        <item>phone</item>
40        <item>shirt</item>
41        <item>computer</item>
42        <item>dress</item>
43        <item>banana</item>
44        <item>sandals</item>
45    </typedKeywords>
46  <menuIdentifiers>
47    <item>nav</item>
48    <item>head</item>
49    <item>menu</item>
50    <item>top</item>
51    <item>head</item>
52    <item>foot</item>
53  </menuIdentifiers>
54  <masterIdentifiers>
55    <item>refine</item>
56    <item>relatedSearches</item>
57    <item>spell</item>
58    <item>category</item>
59    <item>categories</item>
60  </masterIdentifiers>
61  <detailIdentifiers>
62    <item>results</item>
63    <item>searchResults</item>
64    <item>entry</item>
65  </detailIdentifiers>
66  <historyFilepath>history.csv</historyFilepath>
67  <processedHistoryFilepath>history.csv.processed</processedHistoryFilepath>
68  <patternsFilepath>patterns.paradigm</patternsFilepath>
69  <patternsToFind>
70    <!-- item>All</item-->
71    <item>Search</item>
72    <item>Sort</item>
73    <item>MasterDetail</item>
74    <item>Call</item>
75    <item>Input</item>
76    <item>Login</item>
77    <item>Menu</item>
78  </patternsToFind>
79  <!-- loginConfiguration>
80    <username>user-name</username>
81    <password>password</password>
82  </loginConfiguration-->
```

```
83  <tokenizerPatterns>
84      <patternEntry>
85            <name>login</name>
86            <regex>sign(ed)?(\\s|_)?(in|out)|log(ged)?(\\s|_)?(in|out)</regex>
87        </patternEntry>
88    <patternEntry>
89      <name>submit</name>
90      <regex>submit</regex>
91    </patternEntry>
92    <patternEntry>
93      <name>homeLink</name>
94      <regex>(home|main(\\s|_)?page|index|logo)</regex>
95    </patternEntry>
96    <patternEntry>
97      <name>imageLink</name>
98      <regex>link(.*)img|img(.*)link</regex>
99    </patternEntry>
100   <patternEntry>
101     <name>nextLink</name>
102     <regex>(link(.*)next|next(.*)link)</regex>
103   </patternEntry>
104   <patternEntry>
105     <name>prevLink</name>
106     <regex>(prev(ious)?(.*)link|link(.*)prev(ious)?)</regex>
107   </patternEntry>
108   <patternEntry>
109     <name>firstLink</name>
110     <regex>(first(.*)link)</regex>
111   </patternEntry>
112   <patternEntry>
113     <name>lastLink</name>
114     <regex>(link(.*)last)</regex>
115   </patternEntry>
116   <patternEntry>
117     <name>languageLink</name>
118     <regex>lang</regex>
119   </patternEntry>
120   <patternEntry>
121     <name>buttonLink</name>
122     <regex>href.*(button|btn)</regex>
123   </patternEntry>
124   <patternEntry>
125     <name>searchResultLink</name>
126     <regex>search.*result(.*row)?</regex>
127   </patternEntry>
128   <patternEntry>
129     <name>link</name>
130     <regex>link|href</regex>
131   </patternEntry>
```

```
132     <patternEntry>
133       <name>option</name>
134       <regex>option</regex>
135     </patternEntry>
136     <patternEntry>
137       <name>username</name>
138       <regex>user(\\s|_)?(name|id)?</regex>
139     </patternEntry>
140     <patternEntry>
141       <name>verifyPassword</name>
142       <regex>verify(\\s|_)?pass(word)?|pass(word)?(\\s|_)?confirm(ation)?</regex>
143     </patternEntry>
144     <patternEntry>
145       <name>password</name>
146       <regex>pass(word)?</regex>
147     </patternEntry>
148     <patternEntry>
149       <name>email</name>
150       <regex>e?mail</regex>
151     </patternEntry>
152     <patternEntry>
153       <name>checkbox</name>
154       <regex>checkbox</regex>
155     </patternEntry>
156     <patternEntry>
157       <name>collapse</name>
158       <regex>collapse</regex>
159     </patternEntry>
160     <patternEntry>
161       <name>firstNameInput</name>
162       <regex>(input|textarea).*first(\\s|_)?name</regex>
163     </patternEntry>
164     <patternEntry>
165       <name>lastNameInput</name>
166       <regex>(input|textarea).*last(\\s|_)?name</regex>
167     </patternEntry>
168     <patternEntry>
169       <name>sort</name>
170       <regex>((input|textarea|select).*sort)</regex>
171     </patternEntry>
172     <patternEntry>
173       <name>search</name>
174       <regex>(input|select|textarea)(.*(=q\\s|q(ue)?ry|s(ea)?rch|pesq(uisa)?|
            procura(r)?|busca(dor)?).*)</regex>
175     </patternEntry>
176     <patternEntry>
177       <name>captcha</name>
178       <regex>captcha</regex>
179     </patternEntry>
```

```
180     <patternEntry>
181       <name>auth</name>
182       <regex>auth</regex>
183     </patternEntry>
184     <patternEntry>
185       <name>numberInput</name>
186       <regex>number|price|quantity|qty\\s|zip(\\s|_)?code</regex>
187     </patternEntry>
188     <patternEntry>
189             <name>button</name>
190             <regex>button</regex>
191         </patternEntry>
192     <patternEntry>
193       <name>clear</name>
194       <regex>clear</regex>
195     </patternEntry>
196     <patternEntry>
197             <name>input</name>
198             <regex>\\/\\/(input|textarea).*text.*</regex>
199         </patternEntry>
200   </tokenizerPatterns>
201 </conf>
```

Listing A.3: The default configuration loaded of there is no user-specified configuration

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Paradigm:Model xmi:version="2.0"
3   xmlns:xmi="http://www.omg.org/XMI" xmlns:Paradigm="http://www.example.org/Paradigm
       "
4  title="http://www.mcgame.com/" >
5    <nodes xsi:type="Paradigm:Init" name="Init" number="0" outgoingLinks="//
         @relations.0"/>
6    <nodes xsi:type="Paradigm:Find" name="Find1" number="1" incomingLinks="//
         @relations.0" outgoingLinks="//@relations.1">
7      <configurations check="NumberOfResults_more_than" Position="1" result="10" >
8        <inputs  field="find_0" value="computer"/>
9      </configurations>
10     <fields name="find_0" id="//input[@id=query and @name=query and @type=text]"/>
11   </nodes>
12   <nodes xsi:type="Paradigm:Call" name="Call2" number="2" incomingLinks="//
         @relations.1" outgoingLinks="//@relations.2">
13     <configurations />
14     <field name="call_0" id="//a[@class=active and @href=http://www.mcgame.com/de/
           account/login and contains(text(),'Login')]"/>
15   </nodes>
16   <nodes xsi:type="Paradigm:Login" name="Login3" number="3" incomingLinks="//
         @relations.2" outgoingLinks="//@relations.3">
17     <configurations validity="Invalid" check="StayOnSamePage" >
```

```
18        <inputs  field="username" value="banana"/>
19        <inputs  field="password" value="apple"/>
20      </configurations>
21      <fields name="username" id="//input[@id=email and @name=email and @type=text]"/
          >
22      <fields name="password" id="//input[@id=password and @name=password and @type=
          password]"/>
23    </nodes>
24    <nodes xsi:type="Paradigm:Input" name="Input4" number="4" incomingLinks="//
        @relations.3" outgoingLinks="//@relations.4">
25      <configurations value="dress" />
26      <field name="input_0" id="//input[@class=alarm\_amount and @name=alarm\_amount
          and @type=text]"/>
27    </nodes>
28    <nodes xsi:type="Paradigm:Call" name="Call5" number="5" incomingLinks="//
        @relations.4" outgoingLinks="//@relations.5">
29      <configurations />
30      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/game
          /10210253-watch-dogs-season-pass and contains(text(),'Watch Dogs Season
          Pass')]"/>
31    </nodes>
32    <nodes xsi:type="Paradigm:Call" name="Call6" number="6" incomingLinks="//
        @relations.5" outgoingLinks="//@relations.6">
33      <configurations />
34      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/publishers
          and contains(text(),'Kataloge von Publishern')]"/>
35    </nodes>
36    <nodes xsi:type="Paradigm:Call" name="Call7" number="7" incomingLinks="//
        @relations.6" outgoingLinks="//@relations.7">
37      <configurations />
38      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/publishers
          /1104-rokapublish]"/>
39    </nodes>
40    <nodes xsi:type="Paradigm:Call" name="Call8" number="8" incomingLinks="//
        @relations.7" outgoingLinks="//@relations.8">
41      <configurations />
42      <field name="call_0" id="//a[@href=http://www.mcgame.com/de and contains(text()
          ,'Mac Spiele')]"/>
43    </nodes>
44    <nodes xsi:type="Paradigm:Find" name="Find9" number="9" incomingLinks="//
        @relations.8" outgoingLinks="//@relations.9">
45      <configurations check="NumberOfResults_more_than" Position="1" result="10" >
46        <inputs  field="find_0" value="banana "/>
47      </configurations>
48      <fields name="find_0" id="//input[@id=searchtextbox and @name=field-keywords
          and @title=Searchen and @type=text]"/>
49    </nodes>
50    <nodes xsi:type="Paradigm:Call" name="Call10" number="10" incomingLinks="//
        @relations.9" outgoingLinks="//@relations.10">
```

```
51      <configurations />
52      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/pages/about and
            contains(text(),'Uber McGame')]"/>
53    </nodes>
54    <nodes xsi:type="Paradigm:Sort" name="Sort11" number="11" incomingLinks="//
        @relations.10" outgoingLinks="//@relations.11">
55      <configurations />
56        <inputs  field="sort_0" value="label=Price: Low to High "/>
57      <fields name="sort_0" id="//select[@id=sort and @class=sort]"/>
58    </nodes>
59    <nodes xsi:type="Paradigm:Call" name="Call12" number="12" incomingLinks="//
        @relations.11" outgoingLinks="//@relations.12">
60      <configurations />
61      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/publishers
            /1045-activision and contains(text(),'Activision')]"/>
62    </nodes>
63    <nodes xsi:type="Paradigm:Call" name="Call13" number="13" incomingLinks="//
        @relations.12" outgoingLinks="//@relations.13">
64      <configurations />
65      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/help and contains(
            text(),'Hilfe \ Support')]"/>
66    </nodes>
67    <nodes xsi:type="Paradigm:Call" name="Call14" number="14" incomingLinks="//
        @relations.13" outgoingLinks="//@relations.14">
68      <configurations />
69      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/publishers
            /1041-bethesda-softworks and contains(text(),'Bethesda Softworks')]"/>
70    </nodes>
71    <nodes xsi:type="Paradigm:Call" name="Call15" number="15" incomingLinks="//
        @relations.14" outgoingLinks="//@relations.15">
72      <configurations />
73      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/publishers
            /1055-capcom and contains(text(),'Capcom')]"/>
74    </nodes>
75    <nodes xsi:type="Paradigm:Call" name="Call16" number="16" incomingLinks="//
        @relations.15" outgoingLinks="//@relations.16">
76      <configurations />
77      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/mac/core and
            contains(text(),'Core Games')]"/>
78    </nodes>
79    <nodes xsi:type="Paradigm:Call" name="Call17" number="17" incomingLinks="//
        @relations.16" outgoingLinks="//@relations.17">
80      <configurations />
81      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/offers and
            contains(text(),'Angebote')]"/>
82    </nodes>
83    <nodes xsi:type="Paradigm:Call" name="Call18" number="18" incomingLinks="//
        @relations.17" outgoingLinks="//@relations.18">
84      <configurations />
```

```
85      <field name="call_0" id="//a[@href=http://www.mcgame.com/de/store/game
            /10470030-hitman-absolution-professional-edition and contains(text(),'
            Hitman Absolution: Professional Edition')]"/>
86    </nodes>
87    <nodes xsi:type="Paradigm:Call" name="Call19" number="19" incomingLinks="//
        @relations.18" outgoingLinks="//@relations.19">
88      <configurations />
89      <field name="call_0" id="//a[@class=active and @href=https://www.mcgame.com/de/
            account/login and contains(text(),'Login')]"/>
90    </nodes>
91    <nodes xsi:type="Paradigm:Call" name="Call20" number="20" incomingLinks="//
        @relations.19" outgoingLinks="//@relations.20">
92      <configurations />
93      <field name="call_0" id="//a[@id=forgot\_password and @href=https://www.mcgame.
            com/reset/password and contains(text(),'Passwort vergessen?')]"/>
94    </nodes>
95    <nodes xsi:type="Paradigm:End" name="End" number="21" incomingLinks="//@relations
        .20"/>
96
97    <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.0"
        destination="//@nodes.1"/>
98    <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.1"
        destination="//@nodes.2"/>
99    <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.2"
        destination="//@nodes.3"/>
100   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.3"
        destination="//@nodes.4"/>
101   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.4"
        destination="//@nodes.5"/>
102   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.5"
        destination="//@nodes.6"/>
103   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.6"
        destination="//@nodes.7"/>
104   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.7"
        destination="//@nodes.8"/>
105   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.8"
        destination="//@nodes.9"/>
106   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.9"
        destination="//@nodes.10"/>
107   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.10"
        destination="//@nodes.11"/>
108   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.11"
        destination="//@nodes.12"/>
109   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.12"
        destination="//@nodes.13"/>
110   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.13"
        destination="//@nodes.14"/>
111   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.14"
        destination="//@nodes.15"/>
```

```
112   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.15"
          destination="//@nodes.16"/>
113   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.16"
          destination="//@nodes.17"/>
114   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.17"
          destination="//@nodes.18"/>
115   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.18"
          destination="//@nodes.19"/>
116   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.19"
          destination="//@nodes.20"/>
117   <relations xsi:type="Paradigm:Sequence" label=">>" source="//@nodes.20"
          destination="//@nodes.21"/>
118 </Paradigm:Model>
```

Listing A.4: The result of running the approach on Table A.1 and Listing A.2.