Inferring UI Patterns with Inductive Logic Programming

Miguel Nabuco, Ana C. R. Paiva
Departamento de Engenharia Informática,
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
miguelnabuco@fe.up.pt; apaiva@fe.up.pt;

Rui Camacho, João Pascoal Faria
INESC TEC e Departamento de Engenharia Informática,
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
rcamacho@fe.up.pt; jpf@fe.up.pt

Abstract—**This paper presents an approach to infer UI patterns existent in a web application. This reverse engineering process is performed in two steps. First, execution traces are collected from user interactions using the Selenium software. Second, the existing UI patterns within those traces are identified using Machine Learning inference with the Aleph ILP system. The paper describes and illustrates the proposed methodology on a case study over the Amazon web site.**

Keywords-Reverse Engineering, Inductive Logic Programming, Web Application, UI Patterns

I. Introduction

In the past years, the usage of web applications has been steadily increasing. This is due to the fact that they are now capable of handling tasks that were before performed only by desktop applications. Web applications design and testing are becoming a major concern, but their lack of standards and conventions is making development and deployment more complicated [9].

Some user interface (UI) patterns exist to provide end users a sense of easiness when using web applications. The time users are willing to take to learn how a web application works is very short. Users like conventions and patterns. The less they have to think how to get the information they need, the better [14]. UI patterns are recurring solutions that solve common design problems. They can be commonly found in web applications. The login form is a good example of it: it provides a simple way for the user to access his account and personal data. In the majority of web applications, a login form is composed by a username, a cyphered text box for the password, and a button to trigger the validation of the data. When this UI pattern is within a web application, the user can automatically infer how it is supposed to be used and what is its functionality. In addition, it is also possible to define specific techniques to discover UI patterns in existing software systems to extract behaviour models that may be useful in several contexts, like software maintenance [17] and model-based testing [13] [26]. This paper presents an approach to identify UI patterns from web applications in two steps. First, execution traces are collected from a dynamic reverse engineering approach. Second, the traces are input to an ILP (Inductive Logic Programming) system to infer the UI patterns.

The goal of an ILP system is to construct first-order definite clause theories [21] or rules, from examples and background knowledge. The background knowledge contains all the necessary predicates encoding the relevant information to infer the rules from the examples. The examples can be positive (instances of the target concept) or negative (non-instances of the target concept, but "close" to the positive examples).

The rest of the paper is structured as follows. Section II addresses the related work, as well as the tools available to perform the needed tasks. Section III describes how the system was implemented. Section IV provides a practical example of the system proposed. Section V provides the conclusions, reports some of the problems found and points out the future work.

II. State of the Art

Reverse engineering is "*the process of analysing the subject system to identify the system's components and interrelationships and to create representations of the system in another form or at a higher level of abstraction*" [5]. In reverse engineering, the subject system is not changed. There are different methods of applying reverse engineering to a system: the dynamic method, in which the data are retrieved from the system at run time without access to the source code, the static method, which obtains the data from the system's source code, and the hybrid method, which combines the two previous methods.

There are many possible approaches to obtain information from application's execution traces [4] [31]. Taniguchi [32] has developed a tool to construct sequence diagrams using dynamic analysis on JAVA programs. Duarte [10] combined static and dynamic information to construct behaviour models that can be used for model checking. Fischer [11] used execution traces for tracing the evolution of a software system. Morgado [20] created a tool to reduce the effort of obtaining models of the structure and behaviour of Graphical User Interfaces (GUI) [19]. Amalfitano [2] presented a technique for testing Rich

Internet Applications (RIAs) that generates test cases from the applications' execution traces.

Several tools have been created to obtain information from web applications. ReWeb [28] obtains dynamic information from web server logs that helps to find structural and navigational problems in web applications. WARE [27] is a static analyser that creates UML diagrams from the web application source code, Crawljax [29] is a tool that obtains graphical sitemaps, by automatically crawling through a web application and Selenium [15] is an open-source capture-replay tool that saves the users interaction in HTML files.

There also some examples in the literature that use ILP systems in the context of reverse engineering [1] [6] [18]. Cohen [7] uses ILP to recover software specifications from a real-world software system. Flach [12] applies ILP to learn integrity constraints from databases.

There are a significant number of ILP systems available on the Web, such as Progol [22], Golem [23] and Aleph [8]. When an ILP system learns from both types of examples it searches for consistent hypotheses (hypotheses that cover all the positives and none of the negative ones). This is known as predictive induction, and it is commonly used at solving classification and prediction tasks. Descriptive induction uses only positive examples to find the hypotheses. In this type of induction, used to discover regularities or uncovering patterns, there are no negative examples as all the data provided to the ILP can be valid [16].

Since ILP can accept almost any kind of information, as background knowledge, to construct its models and both models and data are described in a very expressive language [24], the number of domains where the use of ILP systems can be advantageous to is quite large. The background knowledge of an ILP system can encode data with structure and can handle numerical computations together with relations. However, for complex applications, ILP systems can become very slow, especially if used for predictive induction.

In this work the Aleph ILP system was used. As typical in predictive ILP systems, Aleph transforms the induction process into a search through an order hypothesis space. In the experimental part of this work, Aleph performs a top-down search of the hypothesis space starting with the most general hypothesis and moving towards the more specific ones. The result of this search is the best hypothesis. If not all positive examples are covered in this search then Aleph removes the ones covered and performs another search using the remaining examples.

This work uses ILP to identify existing UI patterns within a set of execution traces extracted by a dynamic reverse engineering process.

III. THE METHODOLOGY

The approach described in this paper uses a dynamic method to extract execution traces from which additional information is inferred. 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). The execution traces collected are then input to an ILP system for identifying information about the existing UI patterns (from a pre-defined catalogue).

As explained in Section I, the ILP system needs examples and background knowledge to infer rules. In the presented application, the background knowledge has two parts. A first part that is independent of the particular UI being analysed. This part is composed by a set of definitions of UI patterns and some auxiliary/general purpose predicates, all of which are encoded in Prolog. This will allow the ILP system Aleph to be applicable in a wide range of UIs. The second part of the background knowledge is composed by the specifics of the traces. This part is therefore case specific. Finally all traces are encoded as the (positive) examples.

The execution traces are captured using Selenium IDE [22]. While the user navigates through a web application, this capture-replay tool is used to record his steps, such as the text inserted in a specified text box or the ID of a clicked button. Each trace generates a HTML table, such as the one seen in Table 1.

TABLE I. TRACE EXAMPLE

TRACE1		
open	/account/login.php	
type	name=form_loginname	pbgt
type	name=form_pw	pbgtpass
click	id=remember_usernamepwd	
clickAndWait	name=login	
clickAndWait	css=button[type="button"]	
click	link=Me	
clickAndWait	link=Log Out	

The data used by Aleph is encoded in Prolog. As such, a parser was developed to convert the extracted interaction traces (Table 1) to a Prolog structure. An execution trace, identified by TraceID, is a sequence of Actions (with a variable ID and the corresponding input values) and a TraceStep that identifies the position of the Action in the sequence/trace. A transition is encoded as the following:

transition(Action(ID,Data), TraceID, TraceStep)

To code the first interaction within trace 1 regarding the introduction of the text "pbgt" in the textbox with the ID "form_loginname", the transition would look like:

transition(enterTextBox(form_loginname,pbgt), trace1, 1)

So far, four different UI patterns are available for the ILP background knowledge. The goal is to (given the execution traces) identify the patterns in the web application under analysis. The patterns are:

A. Login

The login pattern is commonly found in web applications, especially in the ones that provide specific data that only a certain user (or group) may be able to access. This pattern is encoded as an input text box for the username, a text box with cyphered characters for the password, and a button that validates the data.

Login pattern was encoded in Aleph considering two cases: the case where both username and password are valid, and the submit action (click of the button) changes the URL page and the case where one of the parameters (or both) is invalid, and the submit action causes an error message to appear.

B. Search

The search UI pattern consists of an input text box, where the user types the content he wants to search for, and a button, that will send the query to the server. The result of the search is shown in a new page.

C. Sort

The sort pattern consists of a set of buttons that organizes a list of data, both in ascending and descending order. This pattern is commonly present in e-commerce applications, where the user can sort the products by different variables such as name and price. At this stage of the research work, the Sort pattern is encoded in a way that only the execution traces containing both of the ordering options (ascending and descending) are considered to be valid sort patterns.

D. Master Detail

The master detail pattern is present when selecting an element from a set results in filtering/updating another related set accordingly. For example, consider a web application where you can select a course from a set and according to that, the web application shows the students enrolled in that course. To infer this pattern, Aleph compares the content of a UI control before and after a user interaction.

Since interaction data (for instance, the text input) is encoded within the transitions, Aleph needs to backtrack in order to access information, for instance, to compare previous with current variable values. For instance, to identify a Login pattern, Aleph needs to look for the last assigned login and password values in order to access if these correspond to valid or invalid data. To encode the valid login pattern in Aleph, the following Prolog code is:

```
validUserPass(TraceId, TraceStep, Action, User, Pass):-
    transition(Action, TraceId, TraceStep),
    functor(Action, ActionWithoutParameters, _),
    member(ActionWithoutParameters, [clickButton]),
    backTrack(TraceId,Action, User, enterTextBox, TraceStep),
    backTrack(TraceId,Action, Pass, enterTextBox, TraceStep),
    passDB(User,Pass).
```

When Aleph finds a "clickButton" action in a trace, the predicate "backtrack" will try to find the most recent previous user text inputs in the same trace and store the text values in the "User" and "Pass" variables. The predicate "passDB" will then test if the pair username/password is valid.

After all the patterns are encoded and the execution traces provided, the Aleph system must be configured in order to search for the UI patterns (the hypothesis). The execution traces do not qualify as good or bad examples, they are all valid. Therefore, Aleph must be configured to use only positive examples with the parameter setting:

:- set(evalfn, posonly).

The "set" predicate is used to configure all of Aleph parameters. Among the most used parameters is "nodes" that limits the number of clauses Aleph constructs during the search for the best hypothesis.

Aleph [30] must also be told which predicates, from the background knowledge, to use in the construction of the hypotheses. This is achieved by the use of "determinations":

:- determination(traceOk/1, existPatterns/2).

In this way, Aleph knows that it shall provide rules that cover the higher number of examples with the predicate "existPatterns". This particular predicate provides a list containing all types of patterns the system recognises. Aleph will then, trace by trace, verify which patterns are present and return a list with the used ones. At the end of the execution, the rules shall reveal the UI patterns found and the steps it needed to infer each one.

IV. AN ILLUSTRATIVE EXAMPLE

The approach proposed was tested on the Amazon [3] web application. This web application was chosen due to its widespread use in the world. The first step was to collect a significant number of execution traces so Aleph could have a reasonable degree of certainty when inferring the rules.

The execution traces were captured using Selenium IDE. While most of the information was captured automatically when the user navigated through the web application, some of the information had to be manually input. Since it is unfeasible to save all the web page information (due to memory limitations) at each state, the user had to select the most relevant one to save on the execution trace. It is intended to automate the extraction of execution steps by interacting automatically with the web application and without the intervention of the user. However, for the purpose of current experiments, and to assess the feasibility of the approach, it is, for now, a manual step.After the execution traces captured, the JAVA tool converted the execution traces (15 total) to Prolog code that Aleph can interpret. An example of a trace can be seen below:

```
transition(clickButton(id=twotabsearchtextbox),trace1,1).
transition(enterTextBox(id=tabsearchbox,cds),trace1,2).
transition(clickButton(css=input.submit-input),trace1,3).
transition(changePage(css=input.submit-input),trace1,4).
transition(clickButton(link=Software),trace1,5).
transition(changePage(link=Software),trace1,6).
transition(clickButton(id=tabsearchbox),trace1,7).
transition(enterTextBox(id=tabsearchbox,office),trace1,8).
transition(clickButton(css=input.submit-input),trace1,9).
transition(changePage(css=input.submit-input),trace1,10).
```

This trace describes two searches performed in the Amazon web site, as it can be seen in the enterTextBox transitions. Aleph then inferred the rules from the set of traces.

```
[Rule 1] [Pos cover = 7 Rand cover = 37]
patterns(A) :-
```

```
existPatterns(A,[loginUserPass,searchPattern]).

[Rule 2] [Pos cover = 4 Rand cover = 45]
patterns(A) :-
    existPatterns(A,[sortPattern]).

[Rule 3] [Pos cover = 12 Rand cover = 22]
patterns(A) :-
    existPatterns(A,[searchPattern]).
```

The rules show that the search pattern was found in 12 of the 15 traces. This result is obvious since most of the people use Amazon to search for the products they want. The login pattern and the search pattern appear together in 7 traces. The sort pattern was found in 4 traces.

V. CONCLUSIONS

This paper proposed a new method to identify recurrent behaviour present in web applications, by identifying UI patterns.

Aleph ILP system successfully discovered the UI patterns in the execution traces obtained by navigation through the Amazon website. Aleph was configured to use only positive examples, as described in [25]. Inferring the rules proved to be time costly, since the process took 573 seconds, roughly under 10 minutes. As future work, there is the need to improve the estimation method, making it less lengthy; to continue the automation of the whole process, including the web application navigation and execution trace extraction; and to enlarge the set of UI patterns to identify.

ACKNOWLEDGMENT

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

REFERENCES

[1] D. Alrajeh, O. Ray, A. Russo, S. Uchitel; “Extracting Requirements from Scenarios with ILP”; Lecture Notes in Artificial Intelligence, Vol. 4455, Springer-Verlag, pp. 64-78, 2007

[2] D. Amalfitano; “Rich Internet Application Testing Using Execution Trace Data”; Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 274-283, 2010

[3] Amazon UK website: http://www.amazon.co.uk

[4] I. Andjelkovic, C. Artho; “Trace Server: A tool for storing, querying and analyzing execution traces”; JPF Workshop, 2011

[5] E. J. Chikofsky, J. H. Cross; "Reverse engineering and design recovery: a taxonomy"; IEEE Software Journal, vol.7, no.1, pp.13-17, Jan. 1990

[6] W. W. Cohen, P. T. Devanbu; “A Comparative Study of Inductive Logic Programming Methods for Software Fault Prediction”; in Fourteenth International Conference on Machine Learning, pp. 66–74, 1997

[7] W. W. Cohen; “Recovering Software Specifications with Inductive Logic Programming”; Proc. Twelfth National Conference on Artificial Intelligence, pp. 142-148, 1994

[8] J. P. D. Conceição; “The Aleph system made easy”; FEUP MsC Thesis, 2008

[9] L. L. Constantine, L.A.D Lockwood; "Usage-centered engineering for Web applications"; IEEE Software Journal, vol.19, no.2, pp.42-50, Mar/Apr 2002

[10] L. M. Duarte, J. Kramer, S. Uchitel.; “Model extraction using context information”; In Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems (MoDELS'06), pp. 380-394, 2006

[11] M. Fischer, J. Oberleitner, H. Gall, T. Gschwind; “System evolution tracking through execution trace analysis”; CSMR 2005, pp. 112-121, 2005

[12] P. Flach; “From extensional to intensional knowledge: Inductive Logic Programming techniques and their application to deductive databases”; Chapter in Logic Databases, pp. 356-387, 1998

[13] A. Grilo, A. C. R. Paiva, J. P. Faria; “Reverse Engineering of GUI Models for Testing”; Proceedings of the 5th Conferencia Ibérica de Sistemas y Tecnologias de la Información (CISTI 2010), pp. 284-289, 2010

[14] S. Krug; “Don’t make me think: A common sense approach to the Web”; New Riders Publishing Thousand Oaks, 2005

[15] J. Larson; “Testing Ajax applications with Selenium”; InfoQ magazine, 2006

[16] N. Lavrac; “Introduction to Inductive Logic Programming”; J. Stefan Instititute, 2002

[17] P. Molina, S. Meliá, O. Pastor; “User Interface Conceptual Patterns”; Proceedings of the 4th International Workshop on Design Specification & Verification of Information Systems, pp. 3-540, 2002

[18] I.C. Morgado, A.C.R Paiva, J.P. Faria, R. Camacho; "GUI reverse engineering with machine learning"; Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on, pp.27-31, June 2012

[19] I. C. Morgado, A. Paiva, J. P. Faria; “Reverse Engineering of Graphical User Interfaces”; in The Sixth International Conference on Software Engineering Advances, Barcelona, pp. 293–298, 2011

[20] I. C. Morgado, A.Paiva, J.P. Faria; “Dynamic Reverse Engineering of Graphical User Interfaces”; International Journal on Advances in Software, vol. 5, pp. 223-235, 2012

[21] S. Muggleton; “Bayesian inductive logic programming”; In Warmuth, Proceedings of COLT-94, ACM Conference on Computacional Learning, ACM Press, 1994

[22] S. Muggleton; “Inverse entailment and progol”; New Generation Computing, vol. 13, pp. 245-286, 1995

[23] S. Muggleton, C. Feng; “Efficient induction of logic programs”; Proceedings of the First Conference on Algorithmic Learning Theory, 1990

[24] S. Muggleton, L. De Raedt; “Inductive Logic Programming: Theory and Methods”; Journal of Logic Programming, vol 19-20, pp. 629-679, 1994

[25] S. Muggleton; “Learning from positive data”; Proc. Sixth International Workshop on Inductive Logic Programming, pp. 358-376, 1996

[26] A. Paiva, J. Faria, P. Mendes; “Reverse engineered formal models for GUI testing”; Formal Methods for Industrial Critical Systems, pp. 218-233, 2008

[27] M. Di Penta; “Integrating static and dynamic analysis to improve the comprehension of existing web applications”; Proceedings of 7th IEEE International Symposium on Web Site Evolution, pp. 87-94, 2005

[28] F.Ricca, P.Tonella; “Understanding and restructuring web sites with ReWeb”; IEEE Multimedia, vol. 8, pp. 40-51, Apr-Jun 2001

[29] D. Roest; “Automated Regression Testing of Ajax Web Applications”; Faculty EEMCS, Delft University of Technology Msc thesis, 2010

[30] A. Srinivasan; “The Aleph Manual”; 2003

[31] J. Steven, P. Ch, B. Fleck, A. Podgurski; “jRapture: A capture/replay tool for observation-based testing”; Proceedings of the International Symposium on Software Testing and Analysis, ACM Press, pp. 158-167, 2000

[32] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, K. Inoue; "Extracting sequence diagram from execution trace of Java program"; Principles of Software Evolution, Eighth International Workshop on, pp. 148- 151, Sept. 2005