GUI Reverse Engineering with Machine Learning

Inês Coimbra Morgado*, Ana C. R. Paiva*, João Pascoal Faria*†, and Rui Camacho*
*Department of Informatics Engineering, Faculty of Engineering, University of Porto,
rua Dr. Roberto Frias, 4200-465 Porto, Portugal
†INESC Porto

***Abstract*—This paper proposes a new approach to reduce the effort of building formal models representative of the structure and behaviour of Graphical User Interfaces (GUI). The main goal is to automatically extract the GUI model with a dynamic reverse engineering process, consisting in an exploration phase, that extracts information by interacting with the GUI, and in a model generation phase that, making use of machine learning techniques, uses the extracted information of the first step to generate a state-machine model of the GUI, including guard conditions to remove ambiguity in transitions.**

***Keywords*-Reverse Engineering; Model-Based Testing; Machine Learning; Inductive Logic Programming**

I. Introduction

Developers seldom provide formal models of the applications and building one is a very time consuming, error prone process. However, these enable several activities, such as code or test generation, migration to new platforms, documenting and easing the understanding of the application and are, thus, extremely important. This paper proposes a new reverse engineering approach to reduce the effort of building the formal model of the Graphical User Interface (GUI) of an existent System Under Analysis (SUA).

Reverse engineering was initially defined by Rekoff [16], in 1985, as *"the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system"*. This notion was later applied to software systems by Chikofsky and Cross [3], who defined reverse engineering as *"the process of analysing a subject system to identify the system's components and interrelationships"* (exploration of the system) *"and to create representations of the system in another form or at a higher level of abstraction"* (representation of the information).

There are three main approaches for reverse engineering: static, in which information is extracted from the source or byte codes; dynamic, in which information is extracted in run time, without accessing the source code; and hybrid, which mixes both static and dynamic approaches, trying to maximise the amount of extracted information.

There are already some works on reverse engineering, such as [15], [12], [11], [19] and [6] with testing purposes and [7] and [21] for migration between platforms. In general, every work on reverse engineering intends to understand the SUA.

In [6], the authors proposed an approach to reduce the effort of constructing formal models. The main idea was to automatically extract structural and behavioural information from the interaction with the GUI and to apply this information in the generation of a formal model, which would be, afterwards, manually validated in order to guarantee its consistency and completeness. However, the approach had some limitations as it was not able to extract and represent all types of information.

This paper describes a reverse engineering process with two phases: i) an exploration process to extract structural and behavioural information; ii) and an Inductive Logic Programming (ILP) [13], [14] based process, to help solving problems that may surface while generating the formal model.

Machine Learning stands as a sub-field of artificial intelligence [17] with focus on the development of new methods and techniques to improve the automatic construction of models for data.

ILP is a major field in Machine Learning with important applications in (Relational) Data Mining. The main ingredients for ILP are: background knowledge, B, and observations (usually called examples in the ILP literature), E.

It is usual to have two kinds of examples: positive (E^+) (instances of the target concept) and negative (E^-) (used to avoid overgeneralisations). A characteristic of ILP is that both data and models are expressed in a subset of First Order Logic providing a formal model for induction and an expressive language to encode both data and models. Background knowledge consists in a set of predicates encoding all the information that domain experts find relevant for constructing the models.

The major advantages of ILP that make it adequate for this work include its facility to encode data with structure, the facility of using data from diverse sources encoded in different formats and the induction of comprehensible models. It is also usual for an ILP system to combine numerical reasoning with symbolic relations within the same model. Moreover, there are off-the-shelf open-source systems, like Aleph [20], that are ready to use.

There have been some ILP previous works related to software engineering. Shapiro [18] presented an approach for interactive model construction with an oracle, which verified

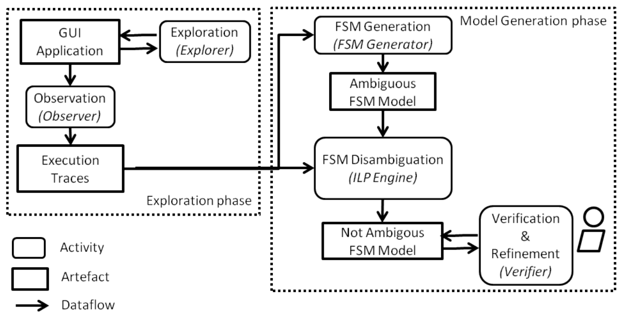

Figure 1. The dataflow view of the reverse engineering process

the correctness of the model and provided counter-examples when necessary; Bratko and Grobelnik [2] presented an approach to use ILP techniques for learning specifications; Cohen and Devanbu [5] studied the usefulness of ILP in software fault prediction systems. In a more broader note, Dzerosk and Lavrac present, in [8], other approaches that follow ILP and the corresponding applications and, in [9], Dzeroski presented the applications of ILP in relational data mining.

The remaining of this paper is organised as follows. Section II provides an overview of the proposed approach. Section III describes the machine learning process. Section IV provides some conclusions and future work.

II. A General Overview

In this section, an overview of the proposed approach is provided, explaining the intended model and how to solve a non-determinacy problem.

A. An Overview of the Process

In this paper, an approach to bring closer the advantages of reverse engineering and machine learning, in order to decrease the effort of building the formal model of a GUI application, is proposed. Figure 1 depicts an overview of the proposed approach.

The approach is divided in two phases: exploration and model generation. The process starts with the exploration: while an *Explorer* (automatic or manual) explores the *GUI* of the SUA, an *Observer* extracts structural and behavioural information, keeping a record of the *execution traces*. This process is dynamic (analysis of the system in run-time) and iterative, stopping when the GUI is considered to be explored. An approach to automate the exploration phase is presented in [6], even though the models extracted in the previous work differ from the ones of this work. Afterwards, a *FSM Generator* generates a Finite State Machine (FSM) [10] from the execution traces, which is used as input, together with the execution traces, for the *ILP Engine*, which solves eventual ambiguous situations. An ambiguity occurs when, from one source state, the same event may lead to different target states. This process is described in more detail in section III. In order to ensure the model is consistent and complete, the generated model goes through a process of manual validation and completion (*Verifier*).

B. Running Example

Along the paper, a sample application is used to better explain the approach. This application's main window (*AppWnd*) has three controls: an *input text box*, a *Find* button, which is initially disabled, and an *Exit* button. When text is inserted in the *input text box*, the *Find* button becomes enabled. Clicking on the *Find* button opens a new dialog (*FindDlg*), which contains an *input text box* (*FindWhat*) and an initially disabled button *Find*. Inserting in *FindWhat* sets the *Find* as enabled. Clicking on *Find* may have one of two results: either the text in *FindWhat* is found in *AppWnd*, closing the *FindDlg*, or the text cannot be found and a new dialog (*NotFoundDlg*) is opened, containing an *OK* button. Clicking on this button closes both the *NotFound* and the *Find* dialogs, returning to the *AppWnd* window.

C. The Execution Traces

Along the exploration, the *execution traces* are recorded. An execution trace is defined by a sequence of user actions along with the state of the GUI after each of those actions: the enabled/disabled actions and the content of the GUI controls. The first user action in any execution trace is *start*.

Table I presents three recorded execution traces on the application described in Section II-B: one in which the application is started and immediately exited (execution trace 0), one in which some text to be found (*"a"*) is actually found (execution trace 1) and another in which it is not found (execution trace 2). The difference in the result of the last two traces lays on the value of the *input text box*. The first column (*TraceId.StepId*) identifies the current execution trace and step, the second one (*Event*) indicates the user action and corresponding parameters that took place in each step and all the remaining columns, apart from the last one, indicate both the status (*E* for enabled and *D* for disabled) of the different actions and the content of the GUI controls. The last column is explained in Section II-D. *W1*, *W2* and *W3* correspond to the different windows/dialogs that are opened during the different executions (*AppWnd*, *FindDlg* and *NotFoundDlg*, respectively).

D. The FSM Model Generation

In this approach, a FSM is generated, representing the dynamic behaviour of the GUI. In the FSM, states are defined by the set of enabled user actions, *i.e.*, there is a transition between states when one of the GUI controls of the enabled windows change from enabled to disabled (or *vice versa*) or when a new window is opened. A transition is labelled by the event that triggered the state evolution. The FSM is represented using UML 2.0 [1].

Table I
EXECUTION TRACES WITH THE CORRESPONDING EXTRACTED INFORMATION: THE ENABLED/DISABLED STATUS OF THE DIFFERENT ACTIONS AND THE CONTENT OF THE GUI CONTROLS (IN ITALIC)

TraceId.StepId	Event (User Action)	Next GUI State									Next FSM State
		W1				W2				W3	
		Text	EnterText(X)	Find	Exit	*FindWhat*	EnterFindWhat(Y)	Find	Cancel	Ok	
0.0	Start	[]	E	D	E	-	-	-	-	-	S0
0.1	W1.Exit	-	-	-	-	-	-	-	-	-	End
1.0	Start	[]	E	D	E	-	-	-	-	-	S0
1.1	W1.EnterText(“a”)	[a]	E	E	E	-	-	-	-	-	S1
1.2	W1.Find	[a]	D	D	D	[]	E	D	E	-	S2
1.3	W2.EnterFindWhat(“a”)	[a]	D	D	D	[a]	E	E	E	-	S3
1.4	W2.Find	[a]	E	E	E	-	-	-	-	-	S1
1.5	W1.Exit	-	-	-	-	-	-	-	-	-	End
2.0	Start	[]	E	D	E	-	-	-	-	-	S0
2.1	W1.EnterText(“b”)	[b]	E	E	E	-	-	-	-	-	S1
2.2	W1.Find	[b]	D	D	D	[]	E	D	E	-	S2
2.3	W2.EnterFindWhat(“a”)	[b]	D	D	D	[a]	E	E	E	-	S3
2.4	W2.Find	[b]	D	D	D	[a]	D	D	D	E	S4
2.5	W3.Ok	[b]	E	E	E	-	-	-	-	-	S1
2.6	W1.Exit	-	-	-	-	-	-	-	-	-	End

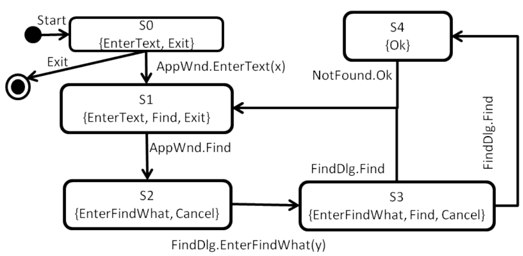

Figure 2. A state machine depicting an ambiguity: from state *S3*, action *Find* may lead to state *S4* (text not found) or to state *S1* (text found)

In table I, the last column (*Next GUI State*) maps each step of each transition to the corresponding FSM state, being each state identifier automatically generated for each set of enabled/disabled actions: state *S0* corresponds to the *AppWnd* window with the *EnterText* and *Exit* actions enabled. When the *EnterText* action takes place, a transition is triggered to state *S1* as the *Find* actions became available, *etc*. Figure 2 depicts the generated FSM.

E. The Ambiguity Issue

One of the problems of this approach is the possibility of ambiguity, *i.e.*, situations where, from the same source state (*Sx*), an event (user action, *E*) reaches different target states (*Sy* and *Sz*), as depicted in (a) of Figure 3. In Figure 2, this ambiguity is present as, from state *S3*, the event *FindDlg.Find* may lead back to state *S1* or forward to state *S4*.

Ambiguities can happen because states have only the set of enabled actions and the transitions may depend on other factors besides that set. In the context of this work, factors that may determine the target state (for the same source state and event) are:

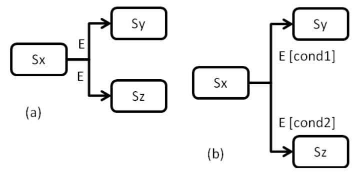

Figure 3. In (a), the event *E* may lead, from the state Sx to two different states (Sy and Sz). As such, a condition *cond1* is added to *E* from Sx to Sy and a condition *cond2* is added to *E* from Sx to Sz, as can be depicted in (b). These conditions will be defined with ILP.

- The parameters of the event (*e.g.*, the result of *Find(“a”)* and *Find(“b”)* may be different, even though the event itself is the same (*Find*));
- The values of the properties of GUI controls (*e.g.*, the effect of an event occurring when a text box is empty may differ from when the same event occurs when the text box contains some text);
- External data (*e.g.*, the content of the clipboard may enable the *Paste* event).

In these situations, it is necessary to identify the actual cause and update the FSM accordingly, adding a guard condition over the relevant variables (parameters of the event, values of the GUI controls' properties and external data) to each transition of the event. These conditions will define the situations when the different target states are reached, as illustrated in Figure 3.

III. THE LEARNING PROCESS

ILP was chosen as an adequate inductive method to identify and solve the conditions in which ambiguous situations occur and classify them accordingly. In order to better understand this problem, the situation depicted in Figure 2 was explored and the Aleph ILP system was applied to that case. As stated in Section I, an ILP system must be provided with background knowledge and examples, which are written in Prolog [4], so that guard conditions that discriminate the two situations and determine which state to go next (see Figure 3 b) may be inferred. As such, the facts *transition/5* are provided as positive examples and both the facts *stateVariable/4* and the predicates *actionInfo/4* are provided to the background knowledge.

A. Encoding of Execution Traces

The *transition/5* facts encode the different transitions of the FSM, having the following signature:

transition(Source, Action, Target, TraceId, StepId).

meaning that there is a transition from the *Source* state to the *Target* state, labelled with *Action* in the step *StepId* of the execution trace *TraceId*. For example, the transition from *S0* to *S1* in the execution trace 1, step 1 is encoded as:

transition(s0, enterText, s1, trace1, step1).

On the other hand, the *stateVariable/4* facts encode information on the GUI controls' properties on each trace and corresponding step, with the following signature:

stateVariable(Control, TraceId, StepId, Value).

meaning that, at step *StepId* of the execution trace *TraceId*, the GUI control *Control* has the value *Value*. For the execution traces described in this paper, the state variables for the steps corresponding to state *S3* (execution trace 1, step 3 and execution trace 2, step 3) are encoded as:

stateVariable(text, trace1, step3, [a]).
stateVariable(findWhat, trace1, step3, [a]).

for the *execution trace 1* and

stateVariable(text, trace2, step3, [b]).
stateVariable(findWhat, trace2, step3, [a]).

for *execution trace 2*.

Both the *transition/5* and *stateVariable/4* facts can be automatically generated from the execution traces of Table I.

B. Encoding of Patterns

The background knowledge also requires the patterns the Aleph system may use to disambiguate the FSM. The patterns currently under consideration are patterns of GUI behaviour, which are encoded through a set of predicates that capture the commonalities of the behaviour. In this example, the useful pattern is a *TextSearch* pattern. In this pattern, there is always a GUI object storing the string to be searched in another GUI object, an user action that actually triggers the search, and two possible results: *found* or *notFound*. In each instantiation of this pattern, the names of the GUI elements may vary, as well as what happens when the search is successfull or not. The signature of these predicates is:

actionInfo(Action, TraceId, StepId, Result).

which indicates the *Result* of the *Action* on the execution trace *TraceId*, step *StepId*. The encoding of the instantiation of the *TextSearch* pattern for this example, (*i.e.*, for the names of GUI objects and user action in this example) is as follows:

actionInfo(find2, TraceId, StepId, found) ←
    stateVariable(text, TraceId, StepId, Text) ∧
    stateVariable(findWhat, TraceId, StepId, FindWhat) ∧
    member(FindWhat, Text).
actionInfo(find2, TraceId, StepId, notFound) ←
    stateVariable(text, TraceId, StpeId, Text) ∧
    stateVariable(findWhat, TraceId, StepId, FindWhat) ∧
    not member(FindWhat, Text).

Other examples of GUI patterns are *Login*, *RangeValidation*, and *MandatoryField*. Even though the predicates that describe these patterns must be encoded manually, they can be reused for other applications, by automatically instantiating them (*i.e.*, the names of objects and actions involved) for each case.

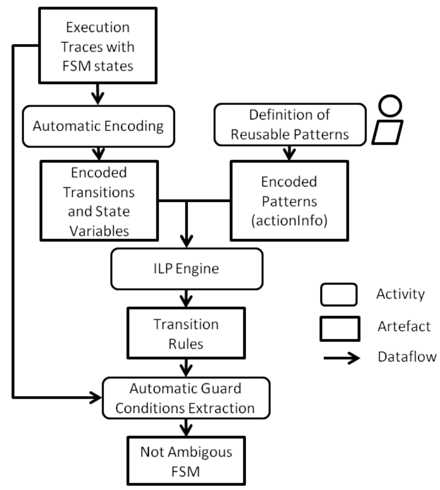

Figure 4. Dataflow view of the learning process

C. Infering Transition Rules and Guard Conditions

For this example, the Aleph system found five rules. Three of those rules encode the deterministic transitions between states before *S3*. The disambiguation of the transition departing from state *S3* is encoded in the following two rules:

transition(Source, Action, Target, TraceId, StepId) ←
    stateName(Target, s1) ∧
    actionInfo(Action, TraceId, StepId, found).

when *FindWhat* was found in *Text* (transition from *S3* to *S1* and

transition(Source, Action, Target, TraceId, StepId) ←
    stateName(Source, s3) ∧
    stateName(Target, s4) ∧
    actionInfo(Action, TraceId, StepId, notFound).

when *FindWhat* was not found in *Text*, transition from *S3* to *S4*.

With all this information, it is possible to infer that it is necessary to add a condition to transitions *S3-S1* (*cond1*) and *S3-S4* (*cond2*):

cond1 = member(FindWhat, Text)
cond2 = not member(FindWhat, Text).

Figure 4 summarises all the process described in this Section, depicting the dataflow of the learning process.

IV. Conclusions and Future Work

This paper proposed a new approach to extract a model of a GUI by an exploration process complemented with machine learning. The exploration process extracts data that is used as examples and background knowledge for ILP, which will solve ambiguous situations that cannot be solved solely by the exploration process. It is yet necessary to explore the encoding of more complex ambiguities on the Aleph system.

As future work, the whole process will be transformed into an iterative one, with the model being complemented at each iteration. In each iteration, the exploration would be guided by the already extracted information so that it could provide more information to the ILP and, consequently, more knowledge may be produced. This would result in a more complete and intelligent exploration.

ACKNOWLEDGMENTS

This work is financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within the project FCOMP-01-0124-FEDER-020554 and the PhD scholarship SFRH/BD/81075/2011.

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*, 2nd ed. Addison-Wesley Professional, May 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1088874

[2] I. Bratko and M. Grobelnik, "Inductive Learning Applied to Program Construction and Verification," *Extended Papers from the IFIP TC12 Workshop on Artificial Intelligence from the Information Processing Perspective: Knowledge Oriented Software Design*, pp. 169–182, Sep. 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=646677.756917

[3] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990. [Online]. Available: http://dl.acm.org/citation.cfm?id=624579.624902

[4] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 4th ed. Berlin, Germany: Springer-Verlag New York Berlin Heidelberg, 2003.

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

[6] I. Coimbra Morgado, A. Paiva, and J. Pascoal Faria, "Reverse Engineering of Graphical User Interfaces," in *The Sixth International Conference on Software Engineering Advances*, no. c, Barcelona, 2011, pp. 293–298.

[7] G. Di Lucca, A. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "WARE: a tool for the reverse engineering of Web applications," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE Comput. Soc, 2002, pp. 241–250. [Online]. Available: http://goo.gl/ZaG7t

[8] S. Dzerosk and N. Lavrac, *Relational Data Mining*. Berlin: SV, 2001. [Online]. Available: http://www-ai.ijs.si/SasoDzeroski/RDMBook/

[9] S. Dzeroski, "Relational Data Mining," in *Computer Software Engineering Series*, 2010, pp. 887–911. [Online]. Available: http://www.arnetminer.org:8080/publication/relational-data-mining-2999763.html

[10] H. Edamatsu, "Finite State Machine," Aug. 1995. [Online]. Available: http://www.freepatentsonline.com/EP0356940.html

[11] A. Grilo, A. Paiva, and J. Faria, "Reverse engineering of GUI models for testing," *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference on*, no. July, pp. 1–6, 2010. [Online]. Available: http://goo.gl/bXcIy

[12] D. R. Hackner and A. M. Memon, "Test case generator for GUITAR," in *Companion of the 13th international conference on Software engineering - ICSE Companion '08*. New York, New York, USA: ACM Press, May 2008, p. 959. [Online]. Available: http://dl.acm.org/citation.cfm?id=1370175.1370207

[13] S. Muggleton, "Inductive logic programming," in *Proceedings of the 1st Conference on Algorithmic Learning Theory*, 1990, pp. 43–62.

[14] S. H. Muggleton and L. D. Raedt, "Inductive Logic Programming: Theory and Methods," *Journal of Logic Programming*, vol. 19,20, pp. 629–679, 1994.

[15] A. C. R. Paiva, J. a. C. P. Faria, and P. M. C. Mendes, "Reverse engineered formal models for GUI testing," in *The 12th international conference on Formal methods for industrial critical systems*. Berlin, Germany: Springer-Verlag, Jul. 2007, pp. 218–233. [Online]. Available: http://dl.acm.org/citation.cfm?id=1793603.1793621

[16] M. Rekoff, "On Reverse Engineering," *IEEE Trans. Systems, Man, and Cybernetics*, no. March-April, pp. 244 – 252, 1985. [Online]. Available: http://www.citeulike.org/group/1374/article/3944848

[17] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[18] E. Y. Shapiro, *Algorithmic Program Debugging*, 1983.

[19] J. a. C. Silva, "GUIsurfer : a tool for reverse engineering of graphical user interfaces," Ph.D. dissertation, 2010. [Online]. Available: http://hdl.handle.net/1822/12267

[20] A. Srinivasan, "Aleph Manual," 2007. [Online]. Available: http://goo.gl/5fRuY

[21] E. Stroulia and T. Systä, "Dynamic analysis for reverse engineering and program understanding," *ACM SIGAPP Applied Computing Review*, vol. 10, no. 1, pp. 8–17, Apr. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=568235.568237