

Reverse Engineering of GUI Models¹

André M. P. Grilo^{*}, Ana C. R. Paiva^{*}, João Pascoal Faria^{*,§}

^{*} Departamento de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
[§]INESC Porto, Campus da FEUP, Rua Dr. Roberto Frias, n° 378, 4200-465 Porto, Portugal
{andre.grilo,apaiva,jpf}@fe.up.pt

Abstract. Graphical user interfaces (GUIs) are an important part of today's software and their correct execution is a very important requirement in order to ensure the effective use of the overall application. One way to find defects in GUIs is to test them by executing test cases and verifying the execution outputs. These test cases can either be created manually or produced automatically from a model of the GUI. It is unpractical to do extensive manual testing because it's a very time-consuming process; however creating a model of the GUI in order to generate automatically test cases is a very difficult task. This paper presents an approach for diminishing the effort required for constructing the model of an existing GUI. The technique is dynamic and exercises the GUI extracting information about the structure of the GUI and some of its behavior. This model is used in the context of model-based GUI testing.

Resumo. As interfaces gráficas com o utilizador (GUI) são uma parte importante do software dos nossos dias e a sua correcta execução é preponderante para assegurar a legitimidade de toda a aplicação. Para encontrar defeitos nas GUIs é comum testá-las executando um conjunto de casos de teste e verificando os resultados produzidos. Os casos de teste podem ser criados manual ou automaticamente a partir do modelo. Não é praticável testar manual e exaustivamente as GUIs. Contudo, construir um modelo para gerar testes automaticamente é uma tarefa muito difícil. Este artigo apresenta uma abordagem para diminuir o esforço necessário para construir modelos de GUIs, a técnica usada é dinâmica e exercita a GUI extraíndo a informação sobre a sua estrutura e o seu comportamento. Este modelo é usado no contexto de teste baseado em modelos.

Keywords: Reverse engineering; GUI modelling; GUI testing.

¹ Work supported by FCT (Portugal) under contract PTDC/EIA/66767/2006.

1 INTRODUCTION

Nowadays' software systems usually feature Graphical User Interfaces (GUIs). GUIs are the mediators between systems and users and their quality is a crucial point in the users' decision of using them. GUI testing is a critical activity aimed at finding defects in the GUI or in the overall application, and increasing the confidence in its correctness. However, GUI testing is a very time consuming Verification and Validation (V&V) activity. The application of model-based testing techniques and tools can be very helpful to systematize and automate GUI testing.

Still, the effort required to construct a detailed and precise enough model for testing purposes (in order to be able to generate not only test inputs but also expected outputs), together with mapping information between the model and the implementation (in order to be able to execute abstract test cases derived from the model on a concrete GUI), are obstacles to the wide adoption of these techniques.

One way to relief the effort mentioned is to produce a partial "as-is" model, together with mapping information, by an automated reverse engineering process. This model will have to be validated and detailed manually, in order to obtain a complete "should-be" model at the level of abstraction desired. Some defects in the application can be discovered in this stage. Overall, the goal is to automate the interactive exploratory process that is commonly followed by testers to obtain a model for an existing application.

This paper is divided in seven sections: the first one is an introduction to the subject of this paper; section 2 is a review of the different reverse engineering approaches and some existing projects about this subject; section 3, presents our approach and section 4 describes the algorithm of the reverse engineering process used; section 5 presents some behaviors that are possible to identify in a GUI, and the necessary rules to infer them; section 6 presents the results generated by our approach applied to the Microsoft Notepad software application; the last section presents some conclusions and future work.

2 STATE OF THE ART

Information systems are critical to the operations of most businesses, and many of these systems have been maintained over an extended period of time, sometimes twenty years or more. Nowadays, many organizations are choosing to reengineer their critical applications to better fit their needs and to take advantage of the new technologies.

Reverse engineering is defined by Chikofsky and Cross [1] as the process of analyzing a subject system in order to identify the systems components, their relationships and to create representations of the system in another form or at a higher level of abstraction. Reverse engineering normally involves extracting the design artifacts and building or synthesizing abstractions that are less implementation-dependent.

For fully understanding existing software, it is necessary to extract both static and dynamic information. Static information embraces usually software artifacts and their relations [2]. Examples of artifacts are classes, methods, and variables. The relations could embrace extending relationship between classes or interfaces, method calls between methods, containment relationships between classes and methods or variables etc., information that can be retrieved from the analysis of the source code. On the other hand, dynamic information not only includes software artifacts, but also contains sequential information, information about concurrency, code coverage, etc.

There are two approaches for reverse engineering: a static approach, in which the static representations of the system (source code) are analyzed without executing the system[3]; and a dynamic approach, in which the system is executed and its external behavior is analyzed [4][5]. The static approach requires access to the source code of the system, which is not always available. Static approaches are particularly well suited for extracting information about the internal structure of the system and dependencies among structural elements. Besides decompilation techniques, dynamic approaches are the only option when the source code is not available. They are well suited to extract the physical structure of the system GUI and some of its behavior, but are more difficult to automate. We focus on dynamic approaches because our goal is to extract information for black-box testing purposes.

There are several examples of projects that use both approaches, one of them is CeLEST [6]. CeLEST is composed of two middleware tools (recorder and pilot). The main purpose of this project was to facilitate the migration and optimization of the uses of a legacy system on a new platform. In order to collect the necessary information to complete the possible task on the old system, CeLEST constructs a map of the system interface and a task model which describes the screen transitions that the user has to traverse in order to accomplish a task.

Another project that uses both approaches is ReWeb [7], the main purpose is to avoid the inevitable degradation of web sites by restructuring and rewriting the site. First it extracts the structure of the website then it analyses the html pages in order to detected syntax errors and other problems and to tune up the model.

Existing automated dynamic approaches try to explore automatically the system through its GUI, but may get blocked because they are not able to find the proper values for accessing all the system user interface elements and exercising all its functionality. E.g., this may happen when a login/password is required to proceed. To overcome this problem, we propose a hybrid approach that combines automatic and manual steps.

3 OVERVIEW OF THE REVERSE ENGINEERING AND TESTING PROCESS

Our reverse engineering process main purpose is to diminish the effort required for constructing the model of an existing GUI for model based testing purposes.

The tool described in this paper is capable of building a preliminary model in Spec# - a pre/post specification language [8] - by interacting with the existing GUI.

The model obtained by the reverse engineering process captures structural information about the GUI (the hierarchical structure of windows and interactive controls within windows and their properties) and also some behavioral information. The model describes the state of each window and window control (enable/disable status, content of text boxes, etc.) and the actions the user can perform on the window controls (e.g., press a button, fill in a text box, etc.). Typically, the preliminary model obtained by this process needs to be completed manually with additional behavior, for instance, some executable method bodies cannot be extracted automatically by the tool.

The final model in Spec# goes through a validation process (to ensure that it describes the correct behavior of the GUI) and then it is used to generate a test suite automatically, using the Spec Explorer tool [9]. A test suite is a set of test segments with sequences of operations that model user actions (with input parameters) interleaved with operations to check the outcomes of those actions.

Test execution is also supported by the Spec Explorer tool. In order to do that, mapping information relating model actions with real actions over GUI controls is needed. The mapping information is gathered during the reverse engineering process and is saved into a XML file. This file keeps information about physical properties of the GUI controls (in order to identify the GUI controls during test case execution), and keeps information about the mapping between abstract and concrete actions. Real actions are written in C# and are capable of simulating user actions over real GUI controls. During test case execution, related actions run in both the specification and implementation levels, in a "lock-step" mode, and, after each step, the results obtained are compared. Whenever an inconsistency is detected, it is reported.

The process of reverse engineering is performed by a dynamic approach which executes the application under test; the structure and behavior of the GUI are extracted and identified. The algorithm of the reverse engineering process is explained in more detail in the sequel.

4 DESCRIPTION OF THE ALGORITHM

The reverse engineering process constructs a GUI model in two phases.

The first phase of the reverse engineering process aims to extract physical properties of the GUI controls and also the navigation map among the different windows of the GUI application under test. The algorithm is as follows:

Phase 1 - Gathering structural information, Fig.1:

1. The user starts the application and the reverse engineering tool, and points out the application starting window.
2. The reverse engineering tool extracts information (physical properties) about all the GUI controls inside that window, and records it in a XML file.
3. The user interacts with GUI controls inside that window in order to open another window of the same application. The reverse engineering tool saves all the steps

performed by the user to navigate from the source window to the destination window in the XML file (in order to be replayed in the second phase of algorithm for opening windows).

4. If a new window was opened go to step 2 until there are no more new windows or the application is closed.

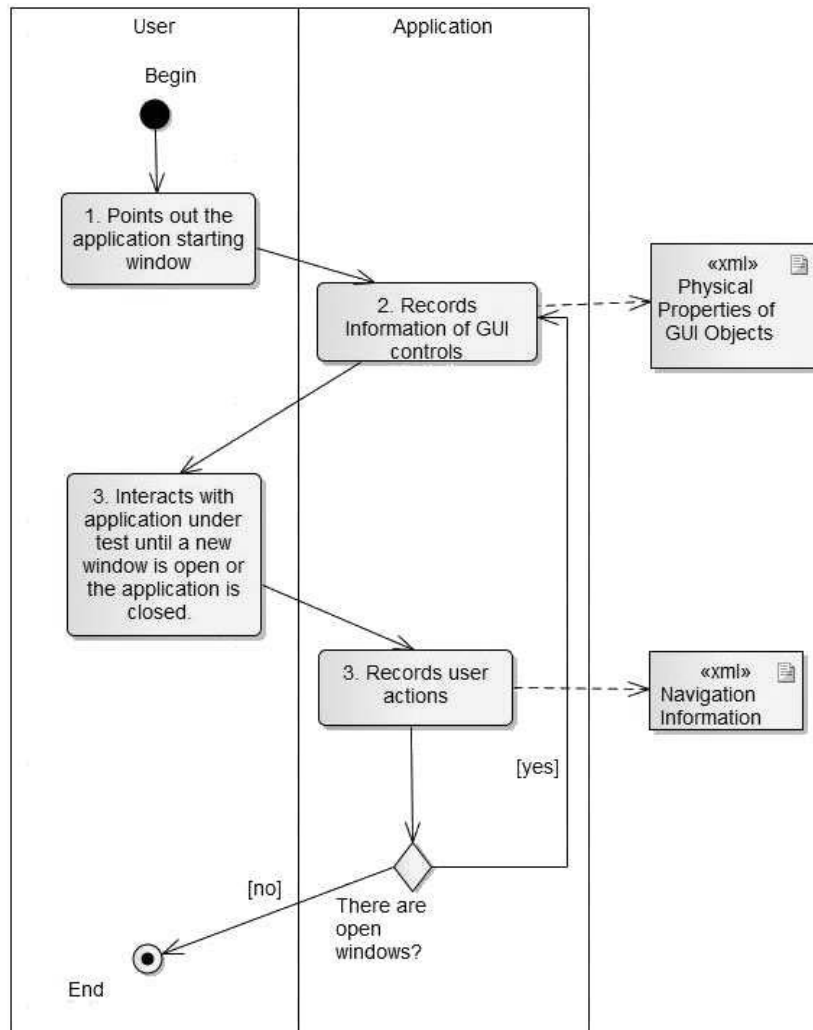


Fig.1. Gathering Structural Information

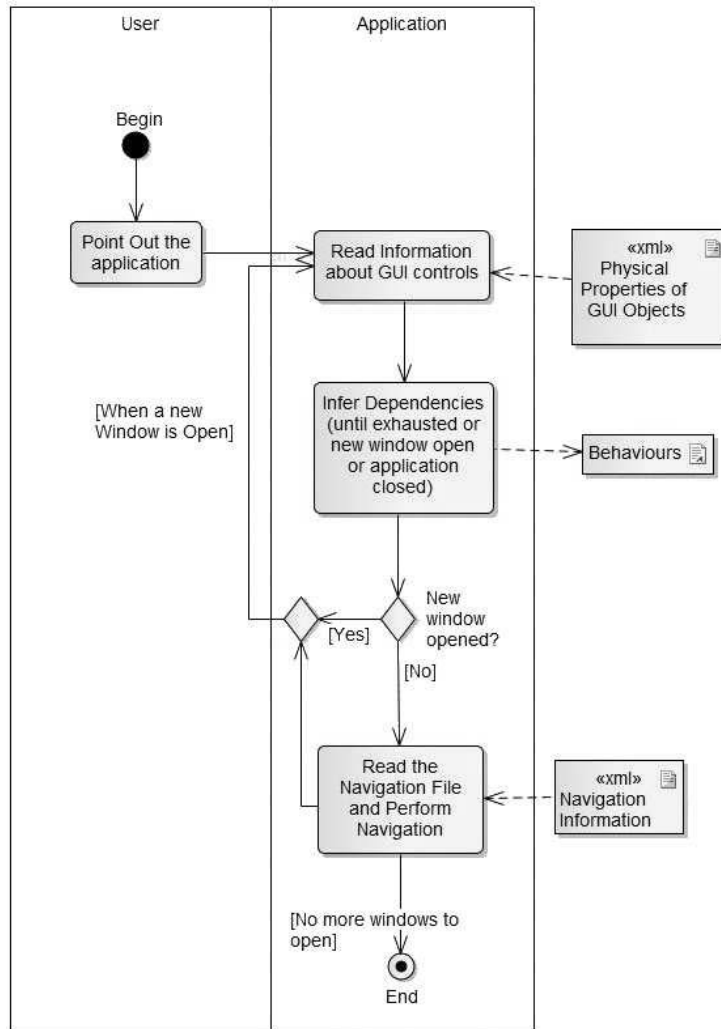


Fig.2. Gathering Behavioral Information

The second phase of the reverse engineering process aims to extract behavioral information, namely dependencies among GUI controls inside each window. The information gathered in the first phase is used here to navigate among windows. The algorithm is as follows:

Phase 2 - Gathering behavioral information, Fig.2:

1. The tester points out the starting window.
2. The tool reads information about GUI controls in this window from the XML file produced in phase 1.

3. To infer dependencies among GUI controls in the current window, the tool interacts with them and checks the changes produced on the properties of other GUI controls, until all controls and actions have been exercised. The dependencies discovered are saved in an XML file.
4. Based on the information captured in phase 1, the tool checks if there is any window that can be reached from the current one and has not yet been explored. If it is the case, the tool replays the steps recorded in the previous phase in order to navigate to that window and the algorithm proceeds to step 2. Otherwise, the exploration stops.

5 RULES TO INFER BEHAVIOUR

In order to identify the different kinds of behaviors that are common in GUIs, it is necessary to define some rules to infer the behavior. To discover the different behaviors three steps are performed: firstly, the actual state of the application is saved; then some action is applied to a GUI control; finally, the final state is compared with the initial state in order to infer dependencies among GUI controls.

Without restrictions, the state space S of an application comprising a set of GUI controls $O = \{o_1, \dots, o_N\}$ will be the Cartesian product of the domain values of the GUI controls properties (p_i), i.e., $S = \text{dom}(o_1.p_1) \times \text{dom}(o_1.p_2) \times \dots \times \text{dom}(o_1.p_m) \times \dots \times \text{dom}(o_N.p_1) \times \text{dom}(o_N.p_2) \times \dots \times \text{dom}(o_N.p_k)$ [10]. There is a distinguished initial state s_0 that represents the initial state when the application is started. The set of properties of a GUI control o depends on its type and is denoted by $\text{Properties}(o.type)$. The value of a property p of a GUI control o in state s is denoted by $s.o.p$.

Depending on their type and state (enabled/disabled), each object accepts user actions (e.g., press a button, set text, etc.). Some of these actions may have parameters (e.g., set text). The set of all possible actions that can be performed on GUI controls is denoted by A . The set of actions available in a GUI control o is denoted by $\text{Actions}(o.type)$ and is a subset of A .

Performing an action a with parameters par on a GUI control o in a GUI state s may cause a transition to a new state s' . Each transition is described by the triggering user action (GUI control, action and parameters), source state and target state. The set of possible transitions is denoted by the transition function $T: A \times PAR \times O \times S \rightarrow S$.

In the case where actions need parameters, there is a configurable set of predefined values to explore, for example, the action `setText` may have $\text{ParamValues}(\text{setText}) = \{'a', 'A', '1'\}$.

An important dependency among GUI controls is the modification of a property p of a control o' when an action a is performed on another control o . The set of this kind of dependencies can be formalized by a set of tuples

$$M = \{ \langle a, o, p, o' \rangle \mid a: A, o: O, o': O, p \text{ is property of } o', o \neq o' \}.$$

The algorithm to extract these dependencies is as follows:

```

s0, s, s' : S, a : A, o : O
M = {}
s = s0
Forall o
  Forall a in Actions(o.type)
    if ParamValues(a) ≠ {}
      Forall par in ParamValues(a)
        s' := T(a, par, o, s)
        M := M U {<a, o, p, o'> | o' : O, p : Properties(o'.type)
                  and s'.o'.p ≠ s.o'.p and o ≠ o'}
        s := s'
      else
        s' := T(a, null, o, s)
        M := M U {<a, o, p, o'> | o' : O, p : Properties(o'.type)
                  and s'.o'.p ≠ s.o'.p and o ≠ o'}
        s := s'

```

The above algorithm can be specialized in order to find special kinds of dependencies:

- **Enabling/disabling dependency.** Actions over some GUI controls can enable or disable other GUI controls in the same window. E.g., in the “Find” dialog of the “Notepad” application, the button “Find Next” is only enabled when text is inserted in the textbox “Find What” (see case study).
- **Value propagation dependency.** The contents of GUI controls may depend on the values of other GUI controls (e.g., summation). To find these dependencies, the algorithm will explore only the action “update text” on GUI controls of type “textbox” ($O' = \{o \mid o.type = \text{textbox}\}$).
- **Master detail dependency.** An example of this behavior is when updating a Combo Box or List box causes changes to the content of a grid or list view.

As soon as the exploration process is finished and the XML file is complete, we generate a Spec# model by applying transformations to the XML file.

The reverse engineering algorithm was implemented on top of UI Automation [11] which is the new accessibility framework for Microsoft Windows, available on all operating systems that support Windows Presentation Foundation.

6 CASE STUDY

The case study was based on the Notepad application. The reverse engineering tool extracted the structure of the Notepad and recorded that structure into an XML file. This file saves information about windows (<window>), the navigation steps (<steps>), controls (<control>) and dependencies among GUI controls (<dependency>). An excerpt of that file is shown below:

```

<?xml version="1.0" encoding="utf-8" ?>
<applicationxmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <window>
    <name>Find</name>
    <AutomationID />
    <ControlType>Dialog</ControlType>
    <L_Controls>
      ...
      <Control>
        <name>Find Next</name>
        <AutomationID>1</AutomationID>
        <ControlType>button</ControlType>
      </Control>
      <Control>
        <name>Find what:</name>
        <AutomationID>1152</AutomationID>
        <ControlType>edit</ControlType>
      </Control>
      ...
    </L_Controls>
  </window>
  <navigation>
    <Nav1>
      <source>Notepad</source>
      <destination>Find</destination>
      <steps>
        <Nsteps>6</Nsteps>
        <0>setText 15document = "a"</0>
        <1>FocusChange Edit</1>
        <2>Mouse Click</2>
        <3>FocusChange Find</3>
        <4>Mouse Click</4>
        <5>FocusChange FindWhat</5>
      </steps>
    </Nav1>
    ...
  </navigation>
  <dependency>
    <dep1>
      <window>Find What</window>
      <source>textbox Find what:</source>
      <destination>button Find Next</destination>
      <action>setText("A", Find What:)</action>
    </dep1>
  </dependency>
</application>

```

```

    <property>Find Next.IsEnabled = true</property>
  </depl>
  ...
</dependency>
...
</application>

```

After extracting all the structural information of the windows of Notepad and recorded the necessary steps to open new windows, the second phase of the algorithm tries to find different behaviors in the several windows.

One of the behaviors that the algorithm was capable to identify in the Find dialog was the enable/disable between the text box (Find what) and the button Find next, as it is possible to identify in Fig.3.

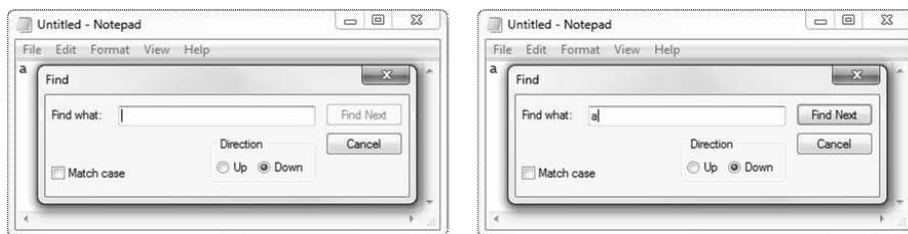


Fig.3. Enable/Disable Behavior in Find Dialog

The specification, in Spec#, generated to represent this behavior is shown below.

```

[Action] void setTextFindWhat(string str)
modifies ButtonFindNext.IsEnabled;
{...}

```

In order to make the specification generated more readable, the name of the method follows the rule:

<Action><caption>

and the name of the GUI controls follows the rule:

<classType><caption>+ '.' + property

In the method above, the name of the method is concatenation of setText (the action) and FindWhat (caption of the textbox control); the name of the GUI control is the concatenation of Button (the classType), FindNext (the caption) and IsEnabled (the property).

7 CONCLUSIONS AND FUTURE WORK

Automated GUI testing has become tremendously important as GUIs become progressively more complex and popular. One way to automate and systematize more the GUI testing process is to generate automatically test cases from GUI models. Our knowledge with GUI testing shows us that such models are very costly to be manually created and the specifications of software applications are rarely available in a way that models used by testing approaches can be automatically created from them.

We have presented a new technique that, through a reverse engineering process, allows obtaining a model of the GUI's structure and some of its behavior. This model is kept in a XML file from which a Spec# specification is generated for testing purposes.

The reverse engineering process proposed combines automatic with manual exploration which solves some of the "blocking problems" found in the approaches described in the state of the art section.

The algorithm used to infer the GUI's behavior follows a specific order of exploration that does not guarantee finding all possible dependencies among GUI controls. Dependencies may exist that show up only after a specific sequence of user actions different from the sequence used by the exploration process.

In the future, it is our intention to implement new algorithms to extend the set of dependencies among GUI objects found automatically by the tool (such as, dependencies that involve creating GUI objects); and improve the Spec# produced making it more complete.

REFERENCES

1. Chikofsky, E.J. and J.H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy*, in *Software, IEEE*. 1990. p. 13-17.
2. Systa, T., *On the relationships between static and dynamic models in reverse engineering Java software*. Reverse Engineering - Working Conference Proceedings, 1999: p. 304-313.
3. Moore, M.M. *Rule-based detection for reverse engineering user interfaces*. in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. 1996.
4. Mori, G., F. Paterno, and C. Santoro, *CTTE: support for developing and analyzing task models for interactive system design*. *Software Engineering, IEEE Transactions on*, 2002. **28**(8): p. 797-813.
5. Csaba, L., *Experience with User Interface Reengineering Transferring DOS Panels to Windows*, in *Proceedings of the 1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)*. 1997, IEEE Computer Society.
6. Stroulia, E., et al., *Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach*, in *Proceedings of the Sixth Working Conference on Reverse Engineering*. 1999, IEEE Computer Society.

7. Ricca, F., P. Tonella, and I.D. Baxter. *Restructuring Web applications via transformation rules*. in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*. 2001.
8. Barnett, M., Rustan, and W. Schulte, *The Spec# Programming System: An Overview*.
9. Veanes, M., et al., *Model-based testing of object-oriented reactive systems with spec explorer*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2008. **4949 LNCS**: p. 39-76, ISBN 978-3-540-78916-1.
10. Paiva, A.C.R., *Automated Specification-Based Testing of Graphical User Interfaces*. Department of Electrical and Computer Engineering, 2007. **Ph.D.**
11. Microsoft. *UI Automation*. msdn 2009 [cited 2009]; Available from: <http://msdn.microsoft.com/en-us/accessibility/bb892133.aspx>