Reverse Engineering of Graphical User Interfaces

Work partially supported by FCT under contract PTDC/EIA/66767/2006

Inês Coimbra Morgado, Ana C. R. Paiva
Department of Informatics Engineering,
Faculty of Engineering, University of Porto,
Porto, Portugal
{ei07040, apaiva}@fe.up.pt

João Pascoal Faria
Department of Informatics Engineering,
Faculty of Engineering, University of Porto
INESC Porto
Porto, Portugal
jpf@fe.up.pt

***Abstract*—This paper describes a dynamic reverse engineering approach and the correspondent tool, ReGUI, developed to reduce the effort of obtaining visual and formal models of both the structure and the behaviour of a software application's graphical user interface.**

This paper describes the tool's architecture, the exploration process it follows, the outputs it generates and the rules used to generate a Spec# model, which can be used in the context of Model-Based Graphical User Interface Testing. The case study presents the results obtained by applying the tool to the Microsoft Notepad application.

***Keywords*—ReGUI, Reverse Engineering, GUI testing**

I. Introduction

This research work is part of a wider ongoing project called AMBER iTest. The main goal of this project is to "develop a set of tools and techniques to automate specification based Graphical User Interface (GUI) testing, solving the shortcomings found in previous work, and show their applicability in industrial environments" [1]. Model-Based Testing (MBT) can contribute to increase the systematisation and automation of the testing process. However, the manual construction of a formal model (required as input by MBT techniques) is a too time consuming and error prone activity. The challenge to be tackled in this research work is the automatic construction of part of the software model using reverse engineering techniques, easing the process of creating visual and formal models. To build these models, both structural and behavioural information are required. This information is extracted by the ReGUI tool. The visual models help to quickly understand the GUI. The formal model is written in Spec# [2] and it is necessary to automatically generate test cases inside the AMBER iTest project.

Once extracted, the formal model needs to be verified, completed and validated. This process is of the utmost importance in order to ensure the model describes the intended behaviour. In addition, the extracted model may reveal errors that must be fixed. In that case, the model should be updated in order to describe the intended behaviour and identify, later on, the conformance errors with the application under test. If this validation process is not performed, the extracted model may describe the implemented behaviour, which may be different from the intended behaviour, and be useless as a test oracle. The validated model is then used by the Spec Explorer Tool [3] to generate test cases. These tests are afterwards run over the GUI of the application under test using the GUI Mapping Tool [4].

This paper is divided as follows. Section II describes the state of the art on reverse engineering and Section III presents the developed tool, ReGUI, focusing on its architecture, functioning and artifacts produced. Finally, Section IV presents a case study and Section V presents some conclusions about this research work, along with the limitations of the approach.

II. State Of The Art

"Reverse engineering is the process of analysing a subject system to create representations of the system at a higher level of abstraction" [5]. This representation is usually presented as a model, which can help to better understand an application, can be used by a code generation process to change the platform of legacy systems and can be used to check if the system has the required properties. 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 [6]. 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 [7].

A. Static Reverse Engineering

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 [8].

There are several studies on static reverse engineering [9], [10], [11], [12], [13]. Bouillon et al. implemented a set of derivation rules in *ReversiXML* [9] to enable model extraction from web pages. Instead of extracting a single model it is also possible to extract several models of different abstraction levels or perspectives and it is even possible to obtain models in different abstraction levels from a previously extracted one. In order to support this, some graph grammars were implemented in *TransformiXML* [9], [14].

Vanderdonckt et al. describe a reverse engineering process, which enables the extraction of a model from a web appli-

 ISBN: 978-1-61208-165-6

cation, *VAQUISTA* [10]. This method was developed in order to enable the automatic migration of the web application into other platforms, such as pocket computers or mobile phones.

B. Dynamic Reverse Engineering

Dynamic approaches extract information from the Application Under Analysis (AUA) in run-mode. Unlike static techniques, dynamic approaches are able to extract information about concurrent behaviour, code coverage and memory management [6]. Initially, the data is collected by running the AUA under a debugger or a profiler. There are several strategies to analyse and represent this data [7].

Even though dynamic approaches are not as common as static ones, there are still some important works, which need to be mentioned.

Shehady et al. propose a reverse engineering method to automate part of the interface testing activity. It extracts the user interface's model representing it as a Variable Finite State Machine, which is later on transformed into a Finite State Machine (FSM) for testing purposes [15]. On top of the FSM, the W_p algorithm [16], which assumes the FSM is fully specified, is applied. This algorithm generates tests, which allow the identification of any discrepancies between the FSM and a model specifying the expected output values. The error diagnosis process is manual.

Chen and Subramaniam developed *VESP* (Visual Environment for manipulating test SPecifications) that works on GUI based applications in Java [17]. The *VESP*'s purpose is to obtain a FSM representation of a GUI coded in Java. Black box test cases [18] are generated from the FSM and afterwards executed on the GUI of the AUA. One aspect that differentiates this approach from more common processes is that the graphical environment provided enables the tester to modify the test specification by modifying the FSM itself, without needing to know any internal representation details.

Atif Memon developed a framework, *GUITAR*, which generates and runs test cases on a GUI, using both reverse engineering and model-based testing techniques [19]. *GUI Ripper*, a component of this framework, extracts a GUI model, representing the structure of the GUI as a Forest (graph, which relates the different windows to be opened in the AUA) and the behaviour as an event flow graph (EFG, graph, which relates the different events, which may take place in the AUA) and as an integration tree (tree, which relates the different components of the AUA) [20]. These are used by the remaining framework tools to generate and run the tests.

C. Conclusions

Some information can only be extracted using a dynamic reverse engineering approach, such as concurrency and memory management. Besides, when working with object oriented programs, it is hard to understand the behaviour and even which objects are instantiated through a static analysis, in which case a dynamic approach may be useful [8].

Most of the dynamic approaches presented in this Section generate a FSM model. However, such models lack data like the navigation map (the set of windows that it is possible to open and the actions needed to open such windows), information about whether or not a window is modal and dependencies among GUI elements.

GUITAR generates a GUI Forest, an EFG and an integration tree. GUI forest represents all the windows of the application. However, it does not describe the interaction steps required to open such windows. In the EFG, two events *e1* and *e2* are connected when *e2* can occur after *e1*. However, this graph does not describe when an event is initially disabled and when an event makes another event possible. So, the behaviour that test cases generated from such models may check during test execution is somehow limited.

III. REGUI

The problem at hand in this research work is to diminish the effort of producing visual and formal models of the GUI of a software application for testing purposes. The approach followed by this work is to extract the necessary information from the application while it is running, i.e., dynamic reverse engineering the AUA.

A. Architecture

Figure 1 depicts the architectural organisation of all the components used by the ReGUI tool.

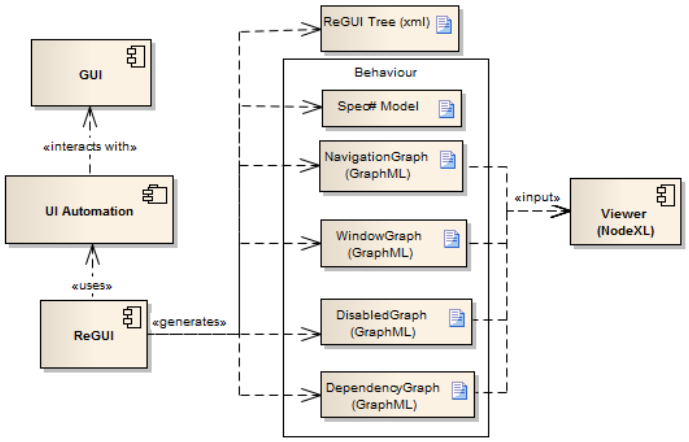

Fig. 1. Architecture of ReGUI

ReGUI uses UI Automation in order to interact with the GUI. UI Automation [21] is the accessibility framework for Microsoft Windows, available on all operating systems that support Windows Presentation Foundation. This framework represents all the applications opened in a computer as a tree (a *Tree Walker*), whose root is the Desktop and whose nodes are the applications opened at a certain moment. The GUI elements are represented as nodes, which are children of the application to which they belong. In the UI Automation framework each of these elements is an Automation Element.

At the end of the execution, the ReGUI tool generates six documents: one to represent the structure of the GUI (*ReGUITree.xml*) and five others to represent its behaviour.

 ISBN: 978-1-61208-165-6

Four of these files are GraphML [22] files: *NavigationGraph.xml*, which represents the navigation map of the GUI, *WindowGraph.xml*, which represents the window graph, *DisabledGraph.xml*, which represents the disabled graph, and *DependencyGraph.xml*, which represents the dependency graph. These files are used as inputs for NodeXL [23], which is a template for Microsoft Excel, that enables the visualisation of the graphs. There is another specification file, written in Spec#, which is input of the test case generation within the AMBER iTest project, and is a specification model read by a MBT tool. These files are described in more detail in Section III-D.

B. Front-End

The ReGUI front-end is shown in Figure 2. This tool is based on the development of a previous one presented in [24]. ReGUI Tool v2.0 is fully automatic and uses a different approach from the previous version of the tool so the results achieved, such as dependencies and graphs produced, are different.

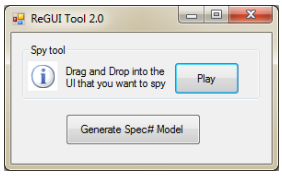

Fig. 2. ReGUI front-end

In order to start the extraction process, it is necessary to identify the GUI to be analysed. In order to do so, it is necessary to drag the *Spy Tool* symbol and drop it on top of the GUI. Following, the user must press the button *Play*, which will start the exploration process. The name of this button changes to *Playing* during the execution and to *Again* at the end. Finally, the user may press the *Generate Spec# Model* button in order to generate the Spec# model. If, for some reason, the user intends to run the ReGUI on the same AUA once more, pressing the button *Again* (button *Play* at the end of the execution) will restart the process.

C. Exploration Process

The exploration process is divided in two phases. The first one navigates through every menu option in order to verify which GUI elements are enabled and which are disabled in the beginning of the execution, i.e., the initial state of the GUI. The second phase also navigates through all the menus but this time the ReGUI tool interacts with all the menus that are enabled at that point. After interacting with each menu item, the ReGUI tool verifies if any window opened, closing it afterwards. Following, ReGUI opens all the menus again in order to verify if any state changed, i.e., if an element previously enabled became disabled or vice-versa.

During the development of the ReGUI tool, it was necessary to face some challenges that are described next:

1) Identification of GUI elements: GUI elements may have dynamic properties, i.e., properties which vary along the execution, such as the *automationId* and the *RuntimeIdProcess*. During the exploration process, the identification of an element is performed through an heuristic based on different properties of the element. This heuristic assigns a percentage of similarity to the different elements according to the ressemblance between their properties. The properties to be compared may be configurable at the beginning of the execution. Nevertheless, there are properties that allow to differentiate two GUI elements. For instance, when two GUI elements have a different *ControlType* value, they are undoubtedly different. These properties are very useful for the identification.

2) Exploration order: In general, the extracted information depends on the order by which the GUI is explored. Currently, ReGUI follows a depth-first algorithm, i.e., all the options of a menu are explored before exploring the next menu and the exploration of each node's children follows the order in which they appear on the GUI. However, if the exploration followed a different order, the dependencies extracted could be different. An example of such may be found in Microsoft Notepad v6.1. The menu item *Select All* requires the presence of text in the main window in order to produce any results. Since, in the beginning, there is no text in the main window, interacting with this menu item does not have any effect. After interacting with the *Time/Date* menu item, that writes the time and date in the main window, the *Select All* menu item would produce visible results (selecting the text and enabling the menu items *Cut*, *Copy* and *Delete* and disabling the menu item *Select All* itself).

3) Synchronisation: To automatically interact with a GUI, it is necessary to wait for the interface to respond after each action. One way to solve this problem is to add a waiting time long enough to ensure the GUI is able to respond. However, this would make the exploration process too slow. In order to surpass this problem, ReGUI checks (with event handlers) when any changes occurred in the UI Automation tree (which reflects the state of the screen in each moment) and continues after that. It is yet possible to assign a waiting time to any action, in order to check if the correspondent result could eventually take more time to occur. However, this approach does not allow the detection of a sequence of timely spaced events that are the result of the same action.

4) Closing a Window: During the execution it is necessary to close windows that are eventually opened, in order to continue with the exploration process. However, there is no standard way of closing them. Windows usually have a top right button for closing purposes but when this is not available it is necessary to interact with another button, which would close the window. The selection of such button is done according to configurable guidelines.

D. Outputs

A tree (ReGUI tree) and four graphs are used internally to store and represent the extracted information. Every node of

 ISBN: 978-1-61208-165-6

these four graphs corresponds to a node in the ReGUI tree. The information stored in these structures is used to generate the formal model in Spec#. The outputs are:

1) ReGUI Tree: The ReGUI tree merges all the UI Automation trees produced during the exploration process. Initially, the ReGUI tree has only the elements visible at the beginning of the exploration and, at the end, it has every element which has become visible at some point of the exploration, such as the content of the windows opened along the process and submenu options.

2) Window Graph: The window graph shows which windows may be opened in the application. Figure 5 is a visual representation of this graph, in which each node is a window. A window may be modal or modeless, being modal if it does not allow interaction with other windows of the same application while opened and modeless otherwise. An egde between two nodes *w1* and *w2* means that it is possible to open *w2* by interacting with elements of *w1*.

3) Navigation Graph: The navigation graph represents the nodes which are relevant to the navigation, i.e., this graph stores information about which user actions must be performed in order to open the different windows of the application. The visual representation of this graph is depicted in Figure 6. A solid edge between a window *w1* (represented by a square) and a GUI element *e1* (represented by a circle or a triangle) means *e1* is inside of *w1* whilst a dashed edge between two GUI elements *e1* and *e2* means it will be possible to interact with *e2* after interacting with *e1*.

4) Disabled Graph: The disabled graph's purpose is to show which nodes are accessible but disabled in the the first phase of exploration process described in Section III-C, i.e., which nodes are disabled (represented by a filled triangle) at the beginning of the exploration. An example of this graph is depicted in Figure 7.

5) Dependency Graph: A dependency between two elements *A* and *B* means that interacting with *A* modifies the value of a property of *B*. After interacting with an element ReGUI looks for any changes in the properties of the different elements (dependencies), as described in section III-C.

Figure 8 is the visual representation of the dependency graph obtained during the exploration process. A solid edge between a window *w1* and a node *n1* means *n1* is inside *w1* and a dashed edge between two nodes *n1* and *n2* means there is a dependency between *n1* and *n2*.

6) Spec# file: The Spec# model is obtained by applying the rules of Figure 3 to the navigation graph. Each window generates a namespace and each edge generates a method annotated with [Action]. Action methods in Spec# are methods that will be used as steps within the following generated test cases. Methods without annotations are only used internally. An example of such model is shown in Figure 9.

IV. Case Study

In this Section, the results of running the ReGUI tool on Microsoft Notepad v6.1 are presented. For this execution, the properties taken into consideration to compare the elements

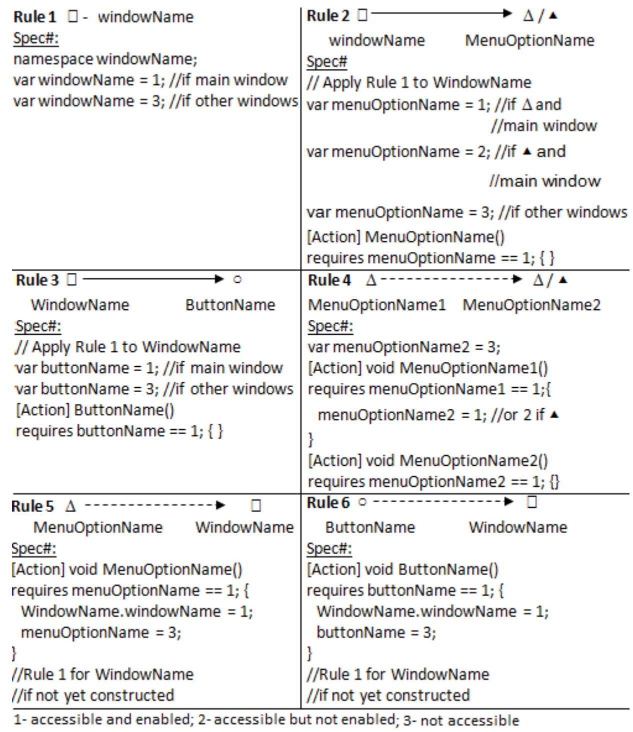

Fig. 3. Rules for the Spec# generation

were the *ControlType*, the *Name*, the *AcceleratorKey*, the *AccessKey*, the *HelpText*, the *ProcessID* and the position. In order to sucessfully close the windows, ReGUI looked, in this order, for buttons whose name was *Cancel*, *No*, *Close*, *Ok*, *Continue* or *X*.

Figure 4 is a simplified representation of the ReGUI tree after exploring the first menu, the menu *File*.

```
Untitled - Notepad
        File
                New
                Open...
                Save
                Save As...
                Page Setup...
                Print...
                Exit
        Edit
        Format
        View
        Help
```

Fig. 4. Part of the ReGUI tree when exploring the menu item *File*

The visual representation of the window graph is represented in Figure 5. In this case, it is possible to conclude that the window *Open*, which is modal, and the window *Windows Help and Support*, which is modeless, may both be opened from the main window of the AUA.

Figure 6 shows the visual representation of the navigation graph. In this example, it is possible to depict that to open the *Save As* window, it is necessary to interact with the menu item *File* and then interact with the menu item *Save* or with the menu item *Save As*. Clicking on the button *Close*, belonging to the window *Save As*, this window is closed and the main

 ISBN: 978-1-61208-165-6

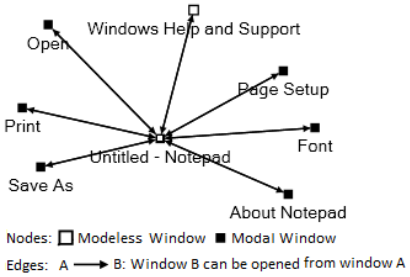

Fig. 5. Visual representation of the window graph

window gets the focus again.

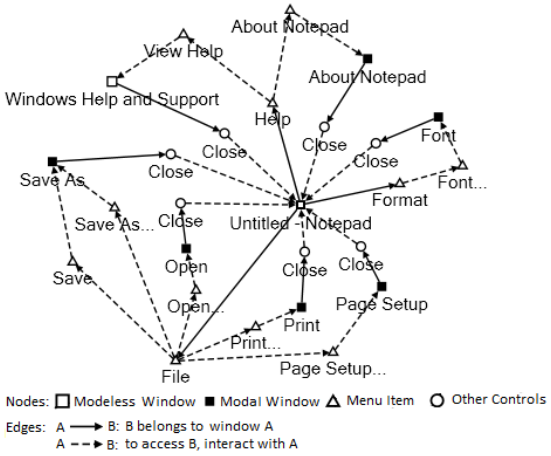

Fig. 6. Visual representation of the navigation graph

Analysing Figure 6 it is possible to verify the window *Find* is not opened during the exploration process as it requires previous insertion of text in the main window.

Figure 7 is the visual representation of the disabled graph, obtained during the first step of the exploration process. In this Figure, the set of menu items *Paste*, *Undo*, *Cut*, *Delete*, *Find Next*, *Find...* and *Copy* are initially disabled. The menu item *Edit* is represented only because it is the father of these menu items.

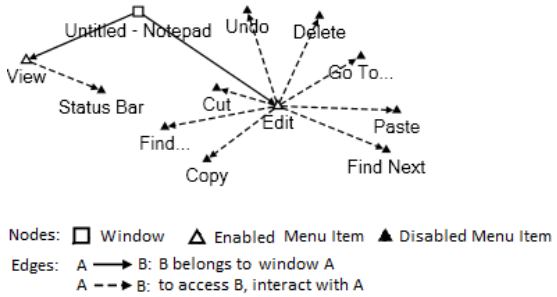

Fig. 7. Visual representation of the disabled graph

Figure 8 shows the visual representation of the dependency graph. When interacting with the menu item *Time/Date*, the menu item *Undo*, which was initially disabled, as depicted in Figure 7, becomes enabled. Thus, there is a dashed arrow from *Time/Date* to *Undo* in the graph.

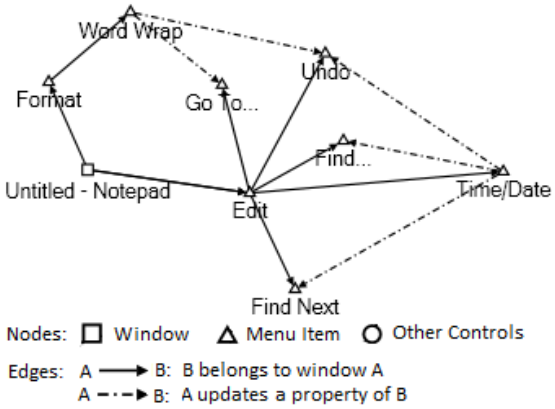

Fig. 8. Visual representation of the dependency graph

Finally, Figure 9 depicts a small sample of the generated Spec# model. The rules applied to generate this Spec# model are in comments. The first namespace corresponds to the main window of the Notepad software application. The two methods within this namespace describe the behaviour when interacting with the menu item *File* and with the menu item *Save*. The second namespace corresponds to the window *Save As* and its method describes the interaction with the button *Close* inside that window.

```
namespace Untitled___Notepad; //Rule 1
var untitled___Notepad = 1;
var menu_itemFile = 1; //Rule 2
var menu_itemSave = 3; //Rule 4
[Action] void Menu_itemFile () //Rule2
requires menu_itemSave == 1;{
  menu_itemSave = 1; //Rule 4}
[Action] void Menu_itemSave ()
requires menu_itemSave == 1; {
 menu_itemSave = 3;  //Rule 5
 WindowSave_As.windowSave_As = 1;}

namespace WindowSave_As; // Rule 1
var windowSave_As = 3;
var buttonClose = 3; // Rule 3
[Action] ButtonClose()
requires buttonClose == 1;{
 buttonClose = 3; //Rule 6
 Untitled___Notepad.untitled___Notepad = 1;
}
```

Fig. 9. Sample of the Spec# formal model generated

For this sample of the Spec# model, no modifications should be necessary upon the manual verification.

V. Conclusions

The ReGUI tool is capable of extracting important information about the behaviour of the AUA, such as navigational information and which GUI elements become enabled or disabled after interacting with another element. The exploration

process is fully automatic. The user just has to point out the AUA.

ReGUI generates graphs, which are useful to quickly visualise the structure and behaviour of the AUA in order to understand its functioning. Also, an important part of the Spec# model is already generated by the tool.

When comparing with the *GUITAR* framework described in Section II-B, it is possible to verify that there is a similarity between the information stored in its GUI Forest and the information stored in both the Window graph and the ReGUI tree as the GUI Forest has information about which window may be opened from another window, along with the structure of each of those windows. The main advantage of the approach described in this paper is that it collects important behavioural information, such as dependencies, and the actions needed to open the several windows of the AUA.

ReGUI has still some limitations. For instance, currently, it only supports interaction through the *invoke pattern* [21] but it may evolve to interact through other *patterns*. In addition, it just tries to open windows from the main window and there are still other dependencies that may be explored. Nevertheless, these limitations could be overcome in a following version of the tool. It is yet objective of the authors to analyse the tools response to more complex systems in order to accurately evaluate the quality of the extracted dependency model.

One of the main difficulties faced during the development of ReGUI was the lack of GUI standards. For example, generally, an opened window is, in the UI Automation tree, child of the main application. However, there are some which are siblings of the application. Furthermore, although each window should have an element called *system menu bar*, which corresponds to the top bar where you can usually find the *minimise*, *maximise* and *close* buttons, some windows do not have that element.

This research work was developed on the context of a project with testing purposes, the AMBER iTest. However, once the model is generated and verified it is possible to use it for other purposes. For instance, to use this model to generate code in languages different from the original one, such as transforming a C# application into a Java application or the other way around.

REFERENCES

[1] S. E. Group, "Amber itest - an automated model-based user interface testing environment," October 2008, http://paginas.fe.up.pt/šofteng/wiki/doku.php?id=projects:amber_itest:start, last access on December 2010.

[2] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: An overview," in *CASSIS International Workshop*, March 2004.

[3] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Formal methods and testing," R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Model-based testing of object-oriented reactive systems with spec explorer, pp. 39–76.

[4] A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *7th International Conference on Formal Engineering Methods*, November 2005.

[5] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, pp. 13–17, January 1990.

[6] T. Systä, "Dynamic reverse engineering of java software," in *Proceedings of the Workshop on Object-Oriented Technology*. London, UK: Springer-Verlag, 1999, pp. 174–175.

[7] M. J. Pacione, M. Roper, and M. Wood, "A comparative evaluation of dynamic visualisation tools," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 80–.

[8] T. Systä, "Static and dynamic reverse engineering techniques for java software systems," Ph.D. dissertation, Faculty of Economics and Administration of the University of Tampere, Kalevantie 4, FI-33014 University of Tampere, Finland, 2010.

[9] L. Bouillon, Q. Limbourg, J. Vanderdonckt, and B. Michotte, "Reverse engineering of web pages based on derivations and transformations," in *Proc. of 3 rd Latin American Web Congress LA-Web2005 (Buenos Aires, October 31-November 2, 2005), IEEE Computer Society Press, Los Alamitos, 2005*, 2005, pp. 3–13.

[10] J. Vanderdonckt, L. Bouillon, and N. Souchon, "Flexible reverse engineering of web pages with vaquista," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, ser. WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 241–248.

[11] J. C. C. J. C. Silva and J. A. Saraiva, "Gui inspection from source code analysis," *Electronic Communications of the EASST*, 2010, to appear.

[12] Y. farn R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach, "Ciao: A graphical navigator for software and document repositories," in *In International Conference on Software Maintenance*. IEEE Computer Society, 1995, pp. 66–75.

[13] M. P. Chase, S. M. Christey, D. R. Harris, and A. S. Yeh, "Managing recovered function and structure of legacy software components," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, ser. WCRE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 79–.

[14] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lpez-Jaquero, "Usixml: A language supporting multi-path development of user interfaces." in *EHCI/DS-VIS*, ser. Lecture Notes in Computer Science, R. Bastide, P. A. Palanque, and J. Roth, Eds., vol. 3425. Springer, 2004, pp. 200–220.

[15] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, ser. FTCS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 80–.

[16] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 591–603, June 1991.

[17] J. Chen and S. Subramaniam, "A gui environment to manipulate fsms for testing gui-based applications in java," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9 - Volume 9*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 9061–.

[18] L. Williams, "Testing overview and black-box testing techniques," 2006.

[19] D. R. Hackner and A. M. Memon, "Test case generator for guitar," in *Companion of the 30th international conference on Software engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 959–960.

[20] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of The 10th Working Conference on Reverse Engineering*, Nov. 2003.

[21] R. Haverty, "New accessibility model for microsoft windows and cross platform development," *SIGACCESS Access. Comput.*, pp. 11–17, June 2005.

[22] U. Brandes, M. Eiglsperger, and J. Lerner, "Graphml primer," April 2007, http://graphml.graphdrawing.org/primer/graphml-primer.html, last access on July 2011.

[23] Microsoft, "Nodexl: Network overview, discovery and exploration for excel," March 2011, http://nodexl.codeplex.com/, last access on May 2011.

[24] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria, "Reverse engineering of gui models for testing," in *5a Conferencia Ibrica de Sistemas y Tecnologias de la Informacin*, July 2009.

 ISBN: 978-1-61208-165-6