FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Reverse Engineering of GUI Models

André Macedo Pinto Grilo

FINAL VERSION

Report of Dissertation
Master in Informatics and Computer Engineering

Supervisor: Ana Cristina Ramada Paiva (PhD)

July 2009

Reverse Engineering of GUI Models

André Macedo Pinto Grilo

Report of Dissertation
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (Professor Associado)

____

External Examiner: José Francisco Creissac Freitas Campos (Professor Auxiliar)

Internal Examiner: Ana Cristina Ramada Paiva (Professor Auxiliar)

24th July, 2009

Resumo

Nos dias de hoje, as aplicações de software apresentam geralmente uma interface gráfica (GUIs). As GUIs são uma parte importante das aplicações de software e sua correcta execução é um requisito importante, a fim de garantir a utilização da aplicação. As GUIs são um factor importante na decisão do utilizador de usar ou não o sistema.

Para garantir a sua correcta execução, é necessário realizar testes sobre as GUI. No entanto, esta é uma actividade difícil, dispendiosa e extremamente morosa, com um número muito reduzido de ferramentas e técnicas para auxiliar no processo de execução de testes. Uma forma de encontrar defeitos em GUIs é efectuando casos de testes e verificando os resultados obtidos.

Estes casos de teste podem ser criados manualmente ou produzidos automaticamente a partir de um modelo do GUI. No entanto, é impraticável fazer manualmente testes extensivos porque é um processo extremamente moroso e, criar um modelo da interface gráfica a fim de gerar automaticamente casos de teste, é uma tarefa muito difícil.

Este trabalho aborda o problema de testes de GUI. O objectivo principal é automatizar o processo de testes de GUI, gerando o modelo da interface gráfica, através de um processo de engenharia reversa, de modo a diminuir o esforço necessário para a construção desse modelo. A técnica de engenharia reversa é dinâmica e exercita a GUI de modo a extrair informações sobre a estrutura da GUI e alguns dos seus comportamentos.

Os modelos extraídos são escritos na linguagem formal de especificação, Spec#. Estes modelos são utilizados pela ferramenta Spec Explorer (ferramenta de testes baseados em modelo) que permite a geração automática de casos de teste contendo, não só, os dados de entrada, mas também os resultados esperados.

A geração do modelo da GUI é feita por um algoritmo dividido em duas fases. Na primeira fase, a ferramenta reúne a informação estrutural das diferentes janelas e regista as acções efectuadas pelo utilizador, a fim de "aprender" como abrir novas janelas. Na segunda fase, a ferramenta tenta extrair comportamento da GUI.

Os casos de teste são gerados a partir do modelo extraído e são executados sobre a implementação. Os resultados obtidos a partir da interface gráfica são comparados com os resultados obtidos a partir da especificação (oráculo de teste). Sempre que se detecte um erro (inconsistência) este é reportado.

A abordagem proposta neste trabalho é ilustrada por um caso de estudo realizado sobre a ferramenta bloco de Notas que faz parte do sistema operativo Microsoft Windows.

A abordagem proposta neste trabalho é uma melhoria em relação às abordagens de engenharia reversa existentes porque combina uma exploração automática com manual e gera um modelo com a estrutura e comportamento de uma GUI em Spec#.

Abstract

Today's software systems usually feature Graphical User Interfaces (GUIs). GUIs are an important part of today’s software and their correct execution is a very important requirement in order to ensure the legitimacy of the overall application. They are an important factor in the user’s decisions to use or not the system.

To ensure their correct execution, it is necessary to perform GUI tests. However, this is a difficult, extremely time-consuming, and costly activity, with a very few tools and techniques to aid in the testing process. One way to find defects in GUIs is to test them by executing test cases and verifying the execution outputs.

Test cases can either be created manually or produced automatically from a model of the GUI. However, it is unpractical to do extensive manual testing because it is a very time-consuming process and creating a model of the GUI in order to generate automatically test cases is a very difficult task.

This research work addresses the GUI testing problem. The goal is to introduce more automation into the GUI testing process by generating the GUI model, through a reverse engineering process in order to diminish the effort required for constructing that model. The reverse engineering technique is dynamic and exercises the GUI extracting information about the structure of the GUI and some of its behaviour.

The extracted models are written in the formal specification language, Spec#. These models are used by Spec Explorer tool (model based testing tool) that allows the automatic generation of test cases containing not only the input data but also the outcomes expected.

The generation of the GUI model is done in a two phase algorithm. In the first phase, the tool gathers the structural information of the different windows and also records the actions performed by the user in order to “learn” how to open new windows. In the second phase, the tool tries to extract GUI behaviour.

Test cases are generated from the extracted model and are executed on the GUI implementation. The results obtained from the GUI are compared with the results derived from the specification (test oracle). Whenever there is a conformance error it is reported.

The approach proposed in this research is illustrated by a case study performed on the Notepad application that ships with the Microsoft Windows operating system.

The approach proposed in this research work is an improvement over the current reverse engineering approaches because it combines automatic with manual exploration and generates a model with the structure and behaviour of a GUI in Spec#.

Acknowledgments

I would like to thank my supervisor, Professor Ana Cristina Ramada Paiva Pimenta, from Engineering Faculty of Porto University, for her guidance, determined search of resources, unforgettable mentoring and encouragement that made this dissertation possible.

A special thank is due to my co-supervisor Professor João Carlos Pascoal de Faria, also from Engineering Faculty of Porto University, for his inputs, enthusiasm and his invaluable perceptiveness in the discussions we had that enriched my perspective.

I owe special thanks to Ricardo Ferreira, researcher of AMBER iTest, for the suggestions, feedback, the time we spent working together in Engineering Faculty of Porto University.

I wish to express my gratitude to my parents, João Grilo and Orlanda Macedo Pinto, and to my brother, Filipe Macedo Pinto Grilo, for all their support, comprehension, and love. Thank you.

Finally, a special thanks to my best friends, Andreia Sá and Stephanie Moreira, for their encouragement, support, and care.

André Macedo Pinto Grilo

Contents

Resumo V

Abstract VII

Acknowledgments IX

Contents XI

List of Figures XIII

List of Tables XV

Acronyms XVII

1. Introduction 1
- 1.1 Context 1
- 1.2 Motivation and Objectives 2
- 1.3 Dissertation's Structure 2

2. State of the Art 5
- 2.1 Model-based testing approaches 6
  - 2.1.1 Ostrand's work 6
  - 2.1.2 IDATG 6
  - 2.1.3 GUITAR 6
  - 2.1.4 Spec Explorer extensions 7
- 2.2 Reverse engineering approaches 7
  - 2.2.1 AUIDL 8
  - 2.2.2 CelLEST 9
  - 2.2.3 MORPH 11
  - 2.2.4 L. Csaba's Approach 12
  - 2.2.5 TaMeX 13
  - 2.2.6 Web Application Reverse Engineering 14
- 2.3 Conclusions 16

3. Overview of the Reverse Engineering and Testing Process 19
- 3.1 AMBER iTest 19
- 3.2 Reverse Engineering and Testing Process 20

4. REGUI Tool 23
- 4.1 REGUI Tool 23
  - 4.1.1 Objectives/Requirements 23
  - 4.1.2 Working Principles 24
  - 4.1.3 Instructions to use the application 25
  - 4.1.4 Internal structure of the application 26
- 4.2 Description of the Algorithm 28
- 4.3 Rules to Infer Behaviour 30

5. Case Study 33

5.1 Structural Information.. 33
5.2 Navigation Steps.. 34
5.3 Inferring Dependencies.. 35
5.4 The Spec# Model.. 35

6. Conclusions and Future Work.. 37
6.1 Future Work.. 37

7. REFERENCES.. 39

8. GLOSSARY.. 41

List of Figures

Figure 2.1 Reengineering Model 8
Figure 2.2 CelLEST Architecture 10
Figure 2.3 MORPH Reverse Engineering Process 12
Figure 2.4 TaMeX Architecture 14
Figure 2.5 WARE Tool Architecture 16
Figure 3.1 AMBER iTest Overall Schematic 20
Figure 3.2 Overview of REGUI Tool 22
Figure 4.1 REGUI Tool – Initial Window 25
Figure 4.2 REGUI Tool 26
Figure 4.3 REGUI Tool Package Diagram 27
Figure 4.4 REGUI Package Diagram Class 27
Figure 4.5 Recording Package Class Diagram 28
Figure 4.6 Playback Package class diagram 28
Figure 4.7 Gathering Structural Information 29
Figure 4.8 Gathering Behavioural Information 30
Figure 5.1 Find Dialog 33
Figure 5.3 Enable/Disable Behaviour in Find Dialog 35

List of Tables

Table 4.1 Formalization of the Behaviour Algorithm 31
Table 5.1 Structural Information of the Find Dialog 34
Table 5.2 Navigational Steps 34
Table 5.3 Dependencies between GUI Controls 35
Table 5.4 Spec# Specification for the behaviour identified 35
Table 5.5 Rule for the method naming 36
Table 5.6 Rule for the GUI Control naming 36
Table 5.7 Spec# Model 36

Acronyms

API	Application Programming Interface
AUIDL	Abstract UI Description Language
AUT	Application Under Test
CCS	Calculus of Communicating Systems
FSM	Finite State Machine
GUI	Graphical User Interfaces
GUITAR	GUI Testing Framework
HCI	Human Computer Interaction
IDATG	Integrated Design and Automated Test Case Generation
MBT	Model-Based Testing
REGUI	Reverse Engineering of Graphical User Interface
UI	User Interface
UML	Unified Modelling Language
V&V	Verification and Validation
XML	eXtensible Markup Language

To my parents and brother
João, Orlanda and Filipe

To my best friend Andreia

Chapter 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 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.

1.1 Context

Over the years there has been significant progress in the development of software systems, which are often difficult to meet the time or budget requirements. These requirements are sometimes so overwhelming that ultimately bring problems to the development and maintenance of systems that will ultimately not be fully tested. In this context, supported by development tools and abstractions is essential to achieve software quality. However the speed at which today takes place information is such that these tools and abstractions cannot mean a consumption of time for those who use, because when it is, they are easily discarded.

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.

The starting point of the work which leads to this dissertation was our analysis of current state-of-the-art methods for GUI (recall Chapter 2). As a rule, the testing activity is performed manually without systematization. Moreover, no guarantee of adequate coverage with respect to some predefined criteria is given.

Although there have been efforts in constructing tools to automate the GUI testing process and diminish the resources (time and money) required, they suffer from many drawbacks that make them unsatisfactory solutions for the problem.

This research work was done under the project AMBER iTest - An Automated Model-Based User Interface Testing Environment. It is a project of the Faculty of Engineer of Porto University (FEUP) with the collaboration of Critical Software (CSW). The main goal of this project is to develop a set of tools and techniques to automate specification based GUI testing, solving shortcomings found in previous work, and show their applicability in industrial environments.

1.2 Motivation and Objectives

The main goal of this research project is to improve current GUI testing methods and tools, as a way to contribute to the construction of higher quality graphical user interfaces and software systems. The integration of formal and empirical methods in software engineering, and the improvement of the usability of formal methods, is a key factor to achieve such goal. AMBER iTest also aims at strengthening the relationship between academia and software industry in Portugal most concerned with software quality, and contributing to the development of a competence centre for software testing and certification in Portugal.

This task will have as a major outcome the following:

- a GUI reverse engineering tool (REGUI tool) that is able to explore an interactive application through its GUI and produce a model of the structure and behaviour of that GUI;
- An algorithm to explore and extract the model of GUIs.

1.3 Dissertation's Structure

This report is divided in six chapters: the first one is an introduction to the subject of this research work; Chapter 2 is a review of the different model based testing and reverse engineering approaches and some existing and relevant projects about this subject; Chapter 3, presents Amber iTest and our approach; Chapter 4 describes the algorithm of the reverse engineering process used and also the formal specification of the algorithm to identify behaviours that are present in a GUI, the necessary rules to infer them, and it presents the

REGUI Tool; Chapter 5 presents a case study; the last section presents some conclusions and future work.

Chapter 2

State of the Art

Software testing is an empirical investigation conducted to evaluate the quality of the product or service under test, without forgetting the context in which it is intended to operate. The main purpose of software testing is to find software bugs by executing a program or application [1].

Software testing can also be known as the process of validating and verifying if software program, application, product respects the business and technical requirements that were guidelines to its design and development, assuring that it works as expected.

For the common user to interact with application it is necessary to implement a user interface. Nowadays Graphical User Interfaces (GUIs) are largely adopted. This may introduce more bugs into the overall application.

Performing software testing through GUIs, in order to find defects in the application and in the GUI, is more difficult and tough than testing a part of software by using its API (Application Programming Interface), because a high programming effort is required to simulate user actions on GUI objects, observe the outputs produced and check its correctness, even when using auxiliary libraries like UI Automation[2]. GUI testing represents a significant amount of the overall testing efforts. Numerous testing tools have been developed to reduce the GUI testing effort, from those that only automated test execution till those that also automate test generation. They will be described in the sequel.

Capture/replay tools, like WinRunner, make easier the generation of test case by recording the user interaction into test scripts, so it can be reproduced later. These kinds of tools are efficient for regression testing, but are not the best to find defects beforehand, as they do not help in test case design [3].

Reverse engineering is defined by Chickofsky and Cross[4] as the process of analysing 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 artefacts and building or synthesizing abstractions that are less implementation-dependent.

A new approach to software testing that has receiving an increased attention is the model-based approach, because of its potential to automate test case generation and the growing embracing of model driven software engineering approaches. The most important characteristics of four model-based approaches rehearsed in the literature to test software applications through their GUI are described below.

2.1 Model-based testing approaches

2.1.1 Ostrand's work

Ostrand's work is a mixed of capture/replay tools with model-based testing concepts[5]. He uses the capture functionality to build a preliminary model of the GUI which is automatically converted into a visual model for generalization. This generalization is obtained from two main concepts: path variations (modelling alternative sequences of actions and iterations) and data variations (fixed values are replaced by variables that take values within defined domains). From the generalized test scenarios constructed using these concepts, several test scripts are then generated in the scripting language supported by the capture/replay component of the test environment for being replayed and tested over the GUI.

2.1.2 IDATG

Another work is IDATG (Integrated Design and Automated Test Generation environment)[6]. In this project the test cases generated, that cover all the edge, are created from Task Flow Graphs (TFGs) that illustrate typical usage scenarios. Each atomic task step is mapped interactively to a GUI object in a design time view of the GUI under test. Test cases generated are converted to a script language supported by a capture/replay tool like WinRunner for test execution. However, it is not clear from the documentation if test data (input data and outputs expected) are supplied manually by the tester or generated by the tool. The authors also mention another test generation technique, based on a formal behavioural specification of the user interface, but do not describe it further.

2.1.3 GUITAR

Memon developed GUITAR, a GUI testing framework. With this framework, test cases are generated from a GUI model comprising event flow graphs and an integration tree[7]. The former represents allowed orderings between pairs of events within a GUI component (window). The latter identifies the hierarchy of GUI components. Memon defines intra and inter component coverage criteria and uses planning techniques from Artificial Intelligence to automatically generate test cases[8]. The GUI model is generated in an internal format by a reverse engineering tool (ripping tool) directly from an executable GUI [9]. However, this internal format is not explained, and it is neither possible to construct the models manually nor to refine the ones generated. The ripping tool extracts the model from a correct GUI and

incorrect versions of the same GUI are tested based on that model. This limits the applicability of the approach in industrial environments.

2.1.4 Spec Explorer extensions

As last example of a model-based GUI testing approach is the approach of some of the proponents of the Amber iTest, in cooperation with the Foundation of Software Engineering Group of Microsoft Research (FSE/MR) [10]. To specify the atomic user actions and composite usage scenarios are written in Spec#[11]. Then with the Spec Explorer tool test cases are automatically generated [12]in a two-step process. The first step is to extract a Finite State Machine (FSM) from the Spec# specification throughout bounded exploration of its (usually infinite) state space; the second step is the test cases generation from the FSM according to a coverage criteria.

2.2 Reverse engineering approaches

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.

For fully understanding existing software it is necessary to extract both static and dynamic information. Static information embraces usually software artefacts and their relations [13]. Examples of artefacts 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 artefacts, 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 analysed without executing the system[14]. 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;
- a dynamic approach, in which the system is executed and its external behaviour is analysed[15][16]. 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 behaviour, but are more difficult to automate. We focus on dynamic approaches because our goal is to extract information for black-box testing purposes.

For modelling GUIs, there are numerous examples of graphical notations. Some are based on UML[17] and its extensions mechanisms. However, UML is not the best for modelling

several particularities of interactive systems, like task modelling and navigation[18]. Normally, to represent task models in UML use cases and activity diagrams are used, but these models do not show clearly task decomposition like the ConcurTaskTree[15] notation does. Navigation between screens can be described by UML state charts, but details such as backtracking, history and concurrency are difficult to represent. UMLi (UML for Interactive Applications) is an example of a UML extension aiming to integrate the design of applications and their user interfaces[19]. It introduces a graphical notation for modelling presentation aspects, and extends activity diagrams to describe collaboration between interaction and domain objects, providing five specialised object flows to model relationships between objects and activities.

Even using a graphical notation, the construction of a GUI model may require a lot of time and effort. For testing already existing GUIs, preliminary models can be constructed by automated reverse engineering processes based on static and/or dynamic analysis techniques

There are several examples of projects that use both approaches applied not only to stand alone applications but also to web applications and are detailed in the next sections of this chapter.

2.2.1 AUIDL

The first known example of UI re-engineering is the Abstract UI Description Language (AUIDL) environment[20]. In this approach, the interface is represented in an object oriented manner while the interface behaviour is described using Milner's process algebra[1]. The first step is to translate the original UI in the AUIDL language able to represent design objects in terms of both structure and behaviour. Different levels of abstractions are defined, leading to a description of dependencies and interaction of UI components obtained from the abstract syntax tree and from the related syntactic information of a particular implementation, as you can see in the Figure 2.1.

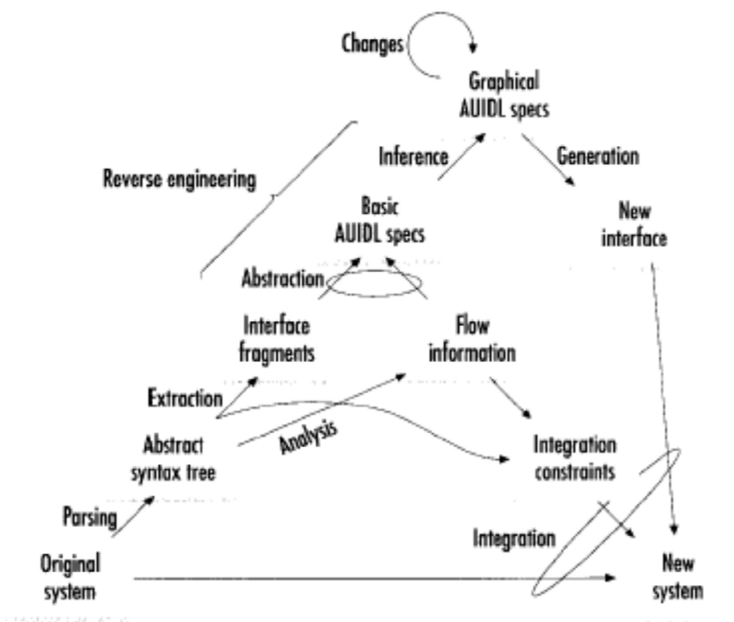

Figure 2.1 Reengineering Model[1]

In order to translate the interface in AUIDL, 'user actions' and 'systems responses' have to be identified using pattern matching techniques. The Milner's process algebra is used to map out the behaviour of the system. Different tools were used to determine the equivalence of two systems with respect to various notions of behavioural equivalence.

According to CCS (Calculus of Communicating Systems) algebra (Milner), two systems are equivalent when they have observably equivalent behaviours or if either can emulate the steps of the other. In CCS, the behaviour of a system is defined as either its entire communication capabilities, or by what is observable is such a system.

AUIDL follows the class hierarchy described by Motif. The spatial organization of display objects is explicitly described with two mechanisms: containment and importation.

This system was developed to migrate from a UI in COBOL into IBM 3270 Environment, so the toolkit for implementing these concepts has been developed using Refine on a workstation. Using Refine/Cobol and the BMS parser, an abstract syntax tree is obtained from the source code of the system. A module extracts AST UI fragments (I/O system calls or user interaction). Syntactic pattern matching and control flow analysis (for the behaviour analysis) are used to construct the abstract specification of the UI. This last step is semi-automated, a part of the code is parsed and automatically abstracted, and the remaining of the sliced code is left to the programmer. Constraints on data and control flow graphs have to be respected at the cut points (in the code) to obtain an equivalent interface.

In the last step, the AUIDL specifications are translated in the EASEL language. The EASEL language allows the automated generation of screens for the IBM 3270 environment. Finally, the generated code is linked with application core.

2.2.2 CelLEST

CEL Legacy Enhancement Software Technologies (CelLEST) is composed of two middle-ware tools: the recorder and the pilot[21]. The main purpose of the CelLEST is facilitating the migration and optimization of the uses of a legacy system on a new platform. In order to extract the necessary information to complete the possible tasks on the old system, Cellest adopts a transversal approach of applications.

The architecture of the CelLEST can be seen in the Figure 2.2.

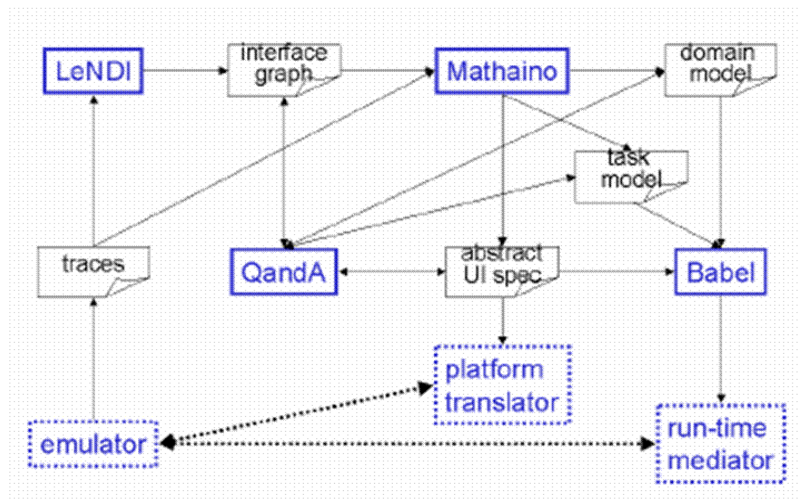

Figure 2.2 CelLEST Architecture

Initially, the recorder identifies aspects of the legacy system relevant to a specific task, instead of having to deal with the whole body of the system code. In this approach, an interface is viewed as a collection of uniquely identifiable screens, each of which allows a set of possible actions to transition from this screen to other screens. The screens are captured by the recorder that records each session (visited screens, actions in order to achieve specific tasks). Then, the pilot translates these actions in functionally equivalent actions in a new GUI. Its role is the same as an application model: to bind the new UI with the old legacy system.

The system that supports the task of mapping the interface is the Legacy Navigation Domain Identifier (LeNDI) System[22]. In the first step of this process, snapshots of the system screens and all the user-system interactions are collected by the Recorder. Then, the interface mapping process proceeds to recognize identifying features in the screen snapshots of the trace, in order to cluster several screen snapshots together as instances of a single unique system screen. To do so, some commonality between these snapshots should be recognized with the help of different techniques (commonly used strings, field positions, projection profiles, keywords and cursor position). LeNDI needs the user to extract some features to use for correlating like snapshots. Then, the user configures the clustering process by deciding which features to use and their relative weights in deciding the similarity between snapshots. The right set of discriminating features must be chosen for every legacy system, and the relative weights of them must be decided after careful study of the recorded traces. The snapshot is considered an instance of the screen with highest matching value if this value is above a user defined threshold. Finally, the user actions that enable the transition (and their preconditions) from one screen to another have to be identified and modelled.

The output of this process is the interface graph whose nodes correspond to the individual screens of the system and its edges correspond to the user action sequences that enable the transitions of the system from one screen to another. The interface graph is a specification of how the legacy system is currently being used by its users. This specification is stored in a database.

In order to be possible the migration of the legacy systems into multiple platforms was created the Mathaino system[23]. The main purpose of Mathaino is to reverse engineer the

information-exchange plan between the legacy system and the user and to specify an optimized abstract UI to accomplish this task. The reverse engineering step: Traces stored (recorded) by the recorder, which were also used by the LeNDI, are the inputs of Mathaino. In picture 2.1, it's it is possible to see the different relations between components, that belongs to CELLEST.

The process starts with the output field recognition. The user has to select each output field and associate them with a name.

After the outputs fields have been identified, the analyser constructs automatically a navigation plan between the different screens and the variable that must appear on these screens. With the help of the plan navigator, the plan has to be validated. For each screen of the trace, the user has to enter an appropriate custom wait pattern in order to synchronize the new interface with the processing time of the old legacy system. The last step of the trace analysis is the association of fields with domain objects (named by the user) in order to generate an object-oriented domain model. With the information collected in the previous steps, an abstract GUI can be created for each task, either automatically or manually. The task and domain model contain enough information to select the best "widget class" for each output/input operation. As output, it's created an abstract UI, which can be translated in one of these two languages (XHTML and WAP)[23].

2.2.3 MORPH

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[14]. Nowadays, many organizations are choosing to reengineer their critical applications to better fit their needs and to take advantage of the new technologies. So another project of UI reengineering is the Model Oriented Reengineering Process for HCI (MORPH). MORPH is a technique and a toolset that supports user interface reengineering. Its main purpose was to migrate from text-based user interface into graphical user interfaces[24].

MORPH starts to identify basic user interactions tasks and associated attributes in legacy code by applying static program analysis techniques, including control flow analysis, data flow analysis, and pattern matching. The resulting model is then used to transform the detected abstractions in the model to a specific graphical widget toolkit.

The MORPH process is composed of three steps[25] and can be seen the representation of the reverse engineering process in the Figure 2.3:

- **The Detection** – This first step can be also called program understanding. In this step the source code is analysed in order to identify user interaction components in the legacy system, through the detection engine.
- **The Representation** – In this step is created a model of the existing user interface, drawn from the previous step. This model is stored in the knowledge base.
- **The Transformation** – As last step, and thanks to transformation engine, it is possible to manipulate, augment and restructure the resulting model to a graphical environment. The human analyst can refine the model in this stage. This step suggests specific graphical implementations and integrates them for user interface abstractions into the legacy code.

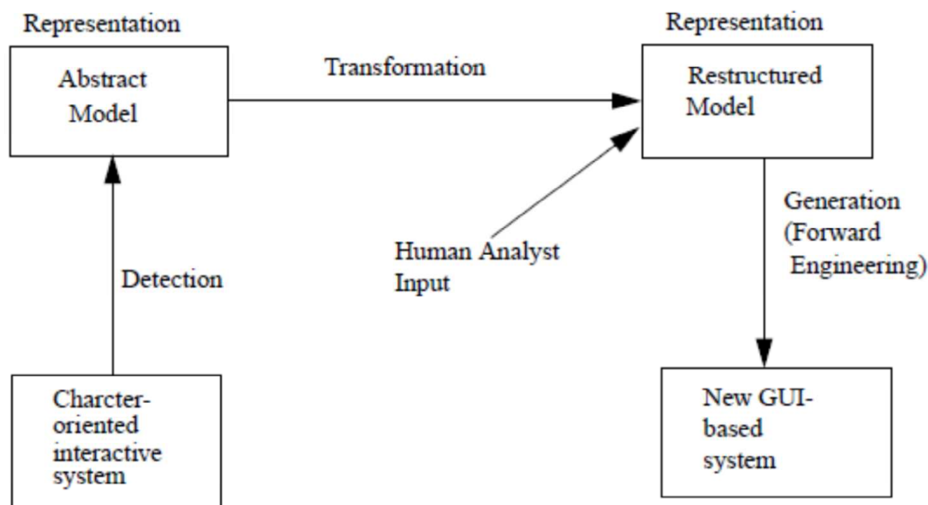

Figure 2.3 MORPH Reverse Engineering Process[26]

2.2.4 L. Csaba's Approach

Another project that was necessary to migrate from a text-based interface (DOS application) into a new graphical interface (Windows based). The original (source) program was written in COBOL and was running in a mainframe environment. It had its own data management system built in and was using the ADIS (ACCEPT/DISPLAY System) run-time module of Micro Focus COBOL[16].

The program could not be easily converted into another language, because of the code structure: the old code was of the 'spaghetti code' type, and should be converted into an event oriented code. This task is very hard, especially for huge monolithic mass of code which was developed gradually over many years.

The best solution is the 'remote controlling'. First the DOS program had to be modified in two ways: replacing the I/O calls by functions communicating with a shared memory area, and making it loadable by Windows.

The replacement functions copy the information into a buffer and another program (Virtual User Program) program that manages the communication to the shared memory from the GUI point of view) reads this information out of the buffer and use it to build its own dialog. Once the real user has made his changes in the Windows dialog, the VUP places the new data into the memory image and pushes the virtual key by putting the appropriate key code into the shared return area and ringing the bell. The buffer is used in order to avoid a continuous refresh of the GUI. The DOS program redrew the screen for each keystroke, but this way of doing is inappropriate in a Windows environment (causes blinking and relatively long processing time).

The realization had 3 steps:

- In step one, the I/O and I/O setups calls were extracted and replaced with calls to the functions of the new interface module (in a search/replace procedure).
- In step two, the interface module was created (the shared memory and communication module). The interface module has two sets of functions, one to support the artificial user program and another called by the modified DOS

program as a replacement for ADIS IO functions. This makes the communication possible between the run time tasks.

- The third step provides the VUP and the new interface. The new interface is generated with the S-Prog Tool. The VUP is a state machine which tries to keep the working engine in a well-defined state as well. To do so it uses “key macros” or key sequences. Macros were developed to gather information from the COBOL source line describing the 25x80 text screen, and convert them into an S-Prog field descriptor table for every DOS Panel. Some transformation from the original UI have been done, in order to benefit from the new display capabilities.

The presentation of the GUI is done manually.

All the previous projects referred were all applied to standalone applications, and their main purpose was the migration from old User Interfaces (UI) – normally text interfaces – to new UI – Graphical User Interface, using also new technologies; while the following examples are web based (web sites, web applications), but with the same purpose, the reverse engineering as a way to update the user interface.

2.2.5 TaMeX

The first example is TaMeX: Task-structure Based Mediation for Information Integration through XML[27].

In this project it was developed a task-structure based mediation framework for the integration of WEB applications within a domain. The approach followed by TaMeX is based on the concept of task-specific mediation: information sources within an application domain are encapsulated within wrapper agents that interact with an intelligent intermediary agent, the mediator. TaMeX, the architecture is represented in the picture Figure 2.4 consists of:

- Wrapper agents: drive and extract information from a set of corresponding Web Applications (within a domain);
- A mediator agent: whose task structure drives the interaction of the aggregate application with the users and controls the flow of requests and information to and from the wrappers.

TaMeX uses two models: the domain and task models. XML is used as an intermediate data structure for information exchange and as a modelling language for the mediator’s domain ontology and task structure.

The information extraction is done with an XPath-based algorithm for generating extraction rules from HTML.

The Domain structure consists in a Hierarchy of Entities, which contains 3 types of information:

- Names and attributes of entities;
- Their relationship;
- The variants of these entities.

In the task structure is modelled the agent’s behaviour. There are three different types of tasks:

1. User interaction tasks: implementing the mediator's interaction with the user;
2. Information-Collection Tasks: the mediator identifies the application that supplies the necessary information and sends the information to the corresponding wrapper;
3. Internal Tasks: internal functions to process information.

The high level tasks are implemented by a menu-driven interface and are activated and decomposed by the users. The low level tasks correspond to the mediator's interaction with either the user or the wrappers to invoke the underlying information sources. The task model represents the control of information exchange and the interaction between users and applications.

In the Figure 2.4 you can see that the architecture is composed of 3 layers (in the left the UI, the mediator in the middle and in the right the wrappers).

The mediator provides a UI generated by applying XSLT to the domain model and the task model. The mediator contains a JAVA servlet which connects the browser with the wrappers and processes the collected information. There is a wrapper for each HTML source and they map the requests in the appropriate protocol. The extracted information is converted in XML.

The wrapper construction is done in 2 phases[28]:

1. The demonstration phase: traces of the interaction between the user's browser and the resource servers are recorded by proxy servers.
2. The learning phase: traces are compared against equivalent XML example request (composed in accord with the domain model) to learn the application request protocol.

The extraction grammar is generated by following a hierarchical approach based on the tree structure of HTML documents. Same concepts are in the same sequence of HTML tags. The learner observes the first XPath to the contents in the HTML page and reproduces it.

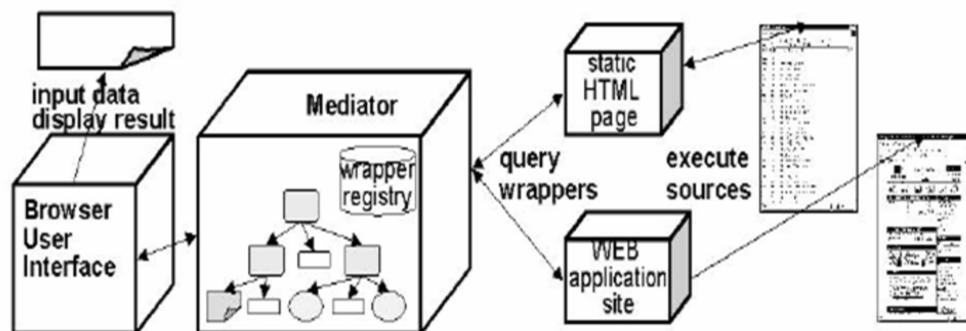

Figure 2.4 TaMeX Architecture[29]

2.2.6 Web Application Reverse Engineering

The main purpose of Web Application Reverse Engineering (WARE)[30] if to provide support to the recovery, from existing Web Applications (WA), of UML diagrams dealing with both static and dynamic content. This tool was developed to support maintenance and site evolution.

In order to reverse engineering the WA, the links (static or dynamic) are mapped as relations. They have defined three different kinds of relations: 1- relations related to the submit button; 2- redirection relations (between a script and a page or between a link and a built page); 3- inclusion relations.

A second classification step helps to clarify the classification of the page objects (images, forms...). From user actions or from the code control flow derives events that trigger interactions between components.

The tool is divided in three different layers. A representation of the architecture of the developed tool can be seen in the Figure 2.5. The first layer is the Interface layer, its purpose is to visualise the results. The second one is the Service Layer, it's the core of the tool, here it's where the extractor analyses and classifies the HTML objects and scripts that modify the WA, then it's stored in IRF files (Intermediate Representation Form = specific ML). Another component of this layer is the Abstractor; the Abstractor is composed of a translator that translates IRF files into a Relational Database (RDB), a Query Executer that permits predefined search in the RDB and produces information based on queries, the UML diagram abstractor produces class diagrams and provides necessary data for the construction of sequence and use case diagrams. The last layer is the Repository, is a database in MS Access, and data is stored in the three formats of the Service Layer. The Taxonomy is represented by class diagram, but they have to be done manually.

The entire process is automated until the dynamic analysis. At this point, the user has to create a use case scenario for each class graph. The dynamic aspect and parameters involved are found back by a RDB search (query). The sequence diagrams are constructed by tracing the events in the objects of the WA.

To sum up, there are 4 main steps in the process:

1. WA static analysis and class diagram recovery
2. Identification of notable sub-graphs in the class diagram, where each sub-graph will be responsible for Web Application functionality
3. Use cases recovery by associating each set of classes to a single use case
4. Sequence diagrams recovery, for obtaining several scenarios of using the WA, by analysing the dynamic interactions among the WA components.

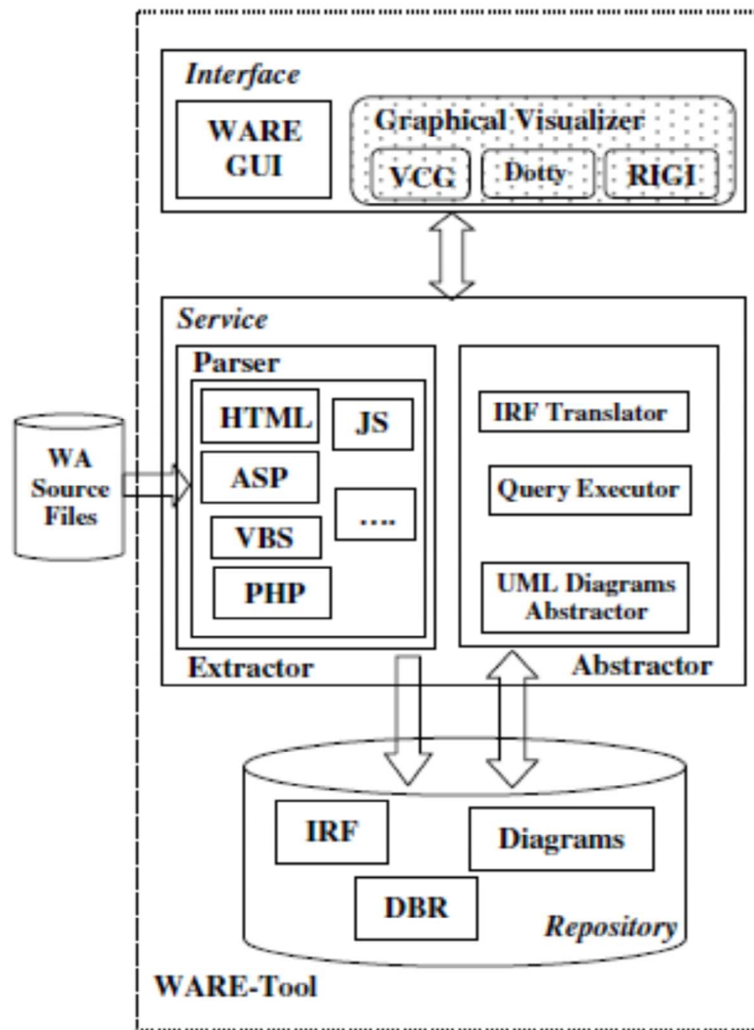

Figure 2.5 WARE Tool Architecture[31]

2.3 Conclusions

GUI testing can be performed manually or with the help of tools. Manual tests are good for exploratory or initial tests, and for those tests performed by the end user. They can find more bugs per test cases executed when performed by experts. Bugs found can provide hints to find other bugs. Manual tests are particularly well suited for usability tests performed by real users. One of the problems with some approaches for manual testing is their lack in systematization. This problem can be reduced by using checklists of standard tests and application of specific tests.

Even so, manual tests require too much effort while providing weak coverage criteria. Test cases are difficult to reproduce and the success of test case execution (number of errors found) is very dependent on the capabilities of the tester. In addition, experienced test specialists are hard to find.

Automated testing is faster than the manual one. The increase of execution speed makes it possible to run more tests in less time, more often, and covering more functionality. One

example is the testing of a strange sequence of events where bugs can be found and that are usually not covered by manual tests. In addition, automated tests may be reused and repeated every time a bug is found. Although automated tests are more efficient in terms of time needed and better use of resources, they may be a source of false sense of security. It is known that "program testing can be used to show the presence of bugs, but never to show their absence"[32].

Model-based testing tools lead to a higher degree of automation. In addition to the automatically generation of test cases, these tools also provide support for automatically executing those tests. This requires a model of the application under test. More time is spent with this activity when compared with the other automated approaches but no time is spent on the generation of test cases since they are calculated automatically. Some of these tools reduce the time spent in constructing the model by reverse engineering existing applications.

For fully understanding existing software it is necessary to extract both static and dynamic information. Static information embraces usually software artefacts and their relations [13]. Examples of artefacts 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 artefacts, but also contains sequential information, information about concurrency, code coverage, etc.

Static approaches are particularly well suited for extracting information about the internal structure of the system and dependencies among structural elements from the source code. 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 behaviour, but are more difficult to automate. We focus on dynamic approaches because our goal is to extract information for black-box testing purposes.

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. It will be better detailed in the next chapters.

Chapter 3

Overview of the Reverse Engineering and Testing Process

3.1 AMBER iTest

The research work described in this document is one of the tasks of a bigger project called AMBER iTest which is described in this chapter in order to better understand the context of this work.

As explained in the previous chapter, one of the main problems with model based GUI testing is the time and effort needed to build the model of the GUI of the application under test. In cases where the application under test already exists, one way to diminish this effort is by automatically deriving (by reverse engineering) a partial "as is" model from the existing application. Such a model will only capture the coarse structure and behaviour of the application; nevertheless, it can serve as a starting point for further manual modelling. The model will have to be manually checked for correctness; defects in the application may be discovered in this process.

In order to help solving these problems, AMBER iTest project aims to construct a set of tools and techniques to automate specification based GUI testing to contribute to the construction of higher quality graphical user interfaces and software systems.

- Development of a visual GUI modelling front-end – hide as much as possible formal modelling details from users, via a visual modelling front-end that is integrated with the formal modelling environment and is supported by appropriate translation tools;
- Generalization of the model to implementation mapping tool – ease the task of mapping the abstract actions described in the model to concrete actions on physical objects in the implemented GUI for several platforms, via an enhanced GUI mapping tool;

- Development of GUI test coverage analysis and enforcement tools–help the generation of higher quality test cases, via the identification of the most appropriate test coverage criteria for model base GUI testing and the development of tools to evaluate the quality and control and guide the generation of test cases; and validate the overall approach by a set of industrial case studies;
- Development of a GUI reverse engineering tool - one way to alleviate the time and effort needed to construct the model of the GUI of the application under test is by automatically deriving (by reverse engineering) a partial "as is" model from the existing application. Such a model will only capture the coarse structure and behaviour of the application; nevertheless, it can serve as a starting point for further manual modelling. The model will have to be manually checked for correctness; defects in the application may be uncovered in this process.

The main goal is to improve current GUI testing methods and tools, as a way to contribute to the construction of higher quality graphical user interfaces and software systems. The integration of formal and empirical methods in software engineering, and the improvement of the usability of formal methods, is a key factor to achieve such goal. The project also aims at strengthening the relationship between academia and software industry in Portugal most concerned with software quality, and contributing to the development of a competence centre for software testing and certification in Portugal. The overall schematic of AMBER iTest can be observed in Figure 3.1.

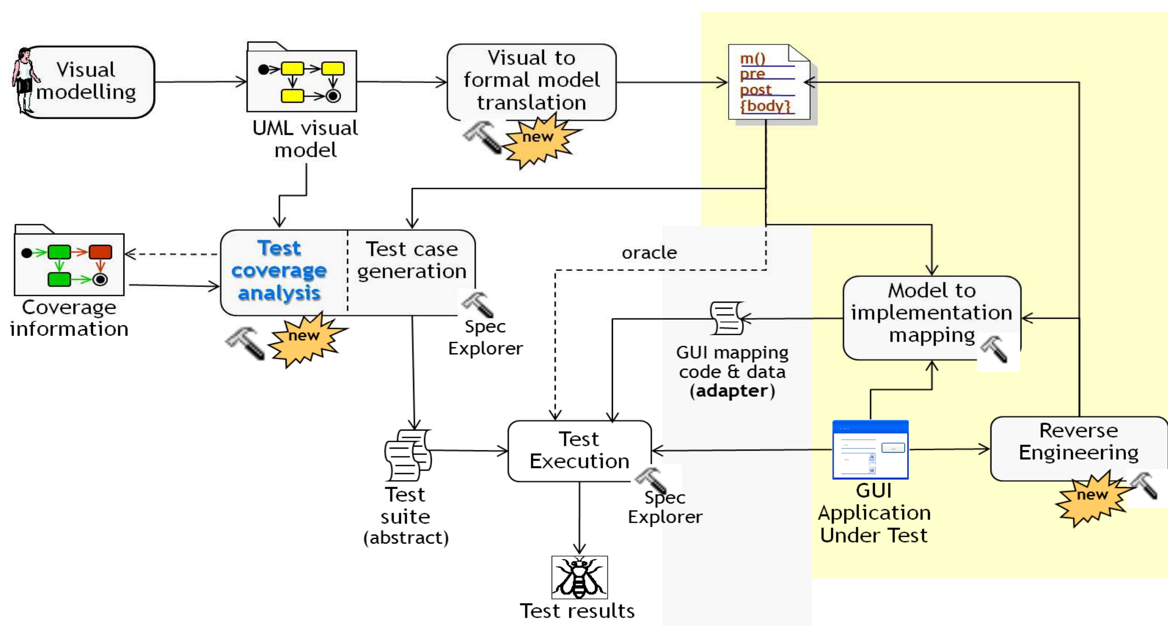

Figure 3.1 AMBER iTest Overall Schematic

3.2 Reverse Engineering and Testing Process

This research work contributes with the REGUI Tool (Reverse Engineering of Graphical User Interface Tool), intents to automate the reverse engineering process. The REGUI tool will explore the application through its GUI, simultaneously with the help of the user interaction with the application (in a way similar to the Memon's tool[9]), and will produce a model of the GUI structure and behaviour as complete as possible, represented in the formal specification language used (Spec#).

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

The tool described – REGUI tool – in this document is capable of building a preliminary model in Spec# - a pre/post specification language [33]– by interacting with the existing GUI, as seen in the Figure 3.2. 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 behavioural 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 behaviour, 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 behaviour of the GUI) and then it is used to generate a test suite automatically, using the Spec Explorer tool[12]. 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.

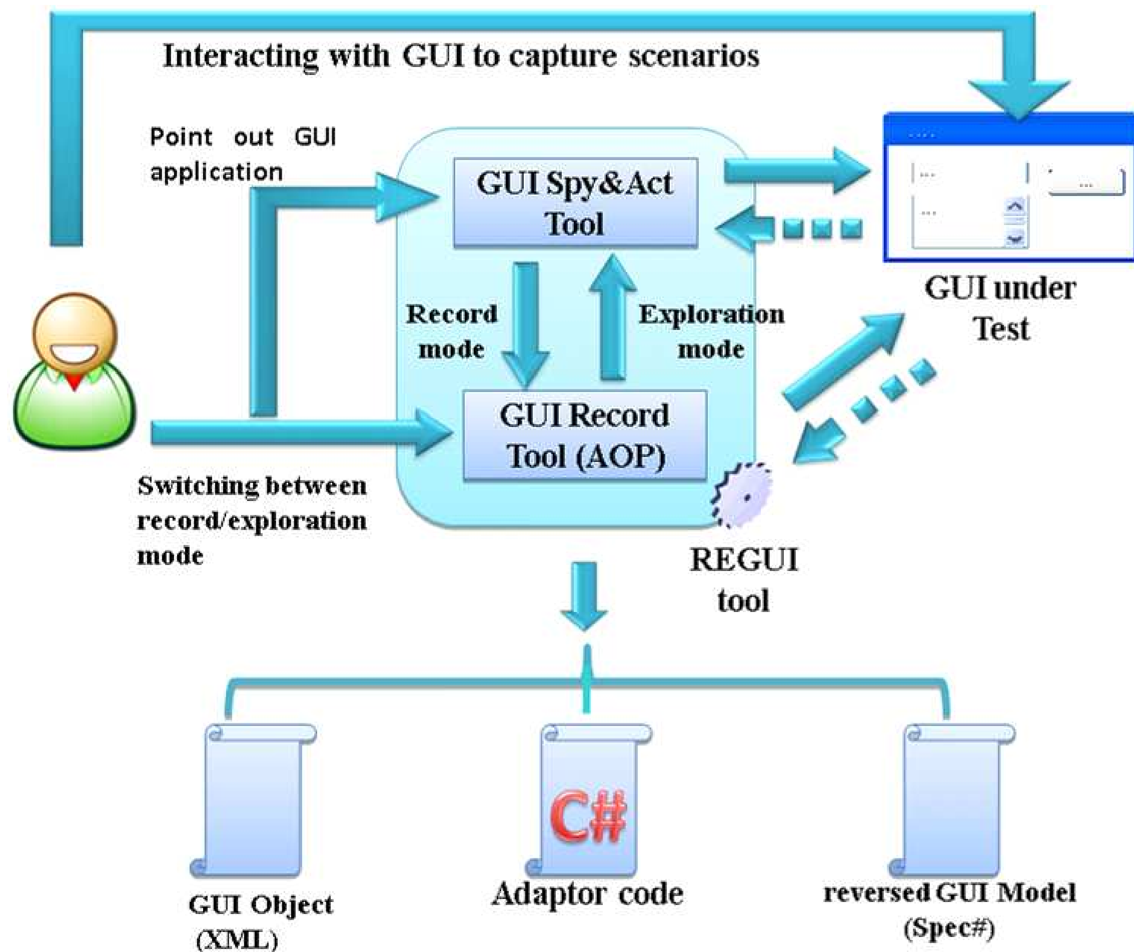

Figure 3.2 Overview of REGUI Tool

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[34] 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 done through the dynamic approach by executing the application under test; it is extracted the structure and identified some behaviours inside the GUIs. The algorithm of the reverse engineering process is explained in the next chapter with more detail.

Chapter 4

REGUI Tool

In this Chapter it will be explained, in the first section, the REGUI Tool, the requirements of the tool, how it works, and the internal structure

In the second section, the Reverse Engineering algorithm that was implemented in the tool to extract the necessary information to create the model to be used to create the Test cases is described in detail in this chapter.

In the third and last section, it is described the algorithm to infer the different kinds of behaviour that is possible to find in GUIs.

4.1 REGUI Tool

This section describes the REGUI Tool. The basic requirements of the tool, its working principles and the main working algorithm are presented. Also, the organization of the output models and a manual to use the application are shown. Finally, a more detailed architecture of the system is explained.

4.1.1 Objectives/Requirements

The main requirements of the REGUI tool are divided in three major categories:

- Supported platforms and controls – indicates the different kind of platforms that applications under test are developed, and GUI controls that are supported by the REGUI tool, and the framework that is the used to develop the REGUI tool;
- Exploration Process – defines how the exploration process is handled;
- Outputs – defines the expected outputs generated by the REGUI tool;

The requirements are numbered, have a priority and a name, and sometimes when necessary, a brief description is made, so for each category are:

- Supported platforms and controls
    - R1 (Essential) Support for Win32, Windows Forms, Windows Presentation Foundation (WPF), Java AWT/Swing and Web GUI applications
        - Detect, extract properties and simulate user actions on GUI applications built using these frameworks
    - R2 (Essential) Support a large set of GUI controls
        - So it can be able to test a wide range of application functionalities
    - R3 (Essential) Use a well-known framework for programmatically simulate user actions on a GUI
        - Well Known, well documented and preferably free
- Exploration Process
    - R4 (Essential) Explore the application through its GUI
    - R5 (Essential) Automatic exploration
        - The application should be explored automatically as much as possible in order to extract the maximum information
    - R6 (Essential) Combine automatic with manual exploration in order to prevent locking situation
        - E.g., when it is necessary to reach a new window through a user/pass situation
- Outputs
    - R7 (Essential) Extract navigational map of the application
    - R8 (Essential) Extract dependencies among GUI controls
        - E.g., enable/disable dependency, calculated values, …
    - R9 (Essential) Generate a visual model
        - In the selected notation
    - R10 (Essential) Generate an intermediate Model in XML
        - To export to other programming/spec language
    - R11 (Essential) Generate mapping information in XML
        - In a format that can be used later to map abstract to concrete actions and test cases

4.1.2 Working Principles

The main purpose of this application is to create a Spec# model, through a reverse engineering process. This tool needs to have the application under test (AUT) as input. This input is provided by pointing the main window of the AUT. After that the tool extracts the window structure. Then the application switches to record mode in order to "learn" how to achieve new windows. When a new window is reached, the tool extracts automatically the structure of the new window, switching again to record mode. These steps are repeated until the user closes the tool or closes the AUT.

4.1.3 Instructions to use the application

Before starting the Reverse Engineering Process, it is necessary to initialise the application under test (AUT). After that the user "Drags-and-Drop" the symbol identified with number 1 in Figure 4.1 to the main window of the AUT in order to indicate the application under test. At this moment, the REGUI tool automatically extracts the physical information about GUI controls and switches again to the Record/Manual mode.

The region identified with the number 2 in Figure 4.1 is used to present information about the GUI controls of the window. The region with number 3 in Figure 4.1shows the tree structure of the GUI controls inside the Window of the AUT.

The region indicated by the 4 in the Figure 4.1, presents the steps and actions performed by the user in the AUT, in order to open new windows. These actions are recorded in the XML file. When the new window is reached, the tree represented in the region 3 is updated.

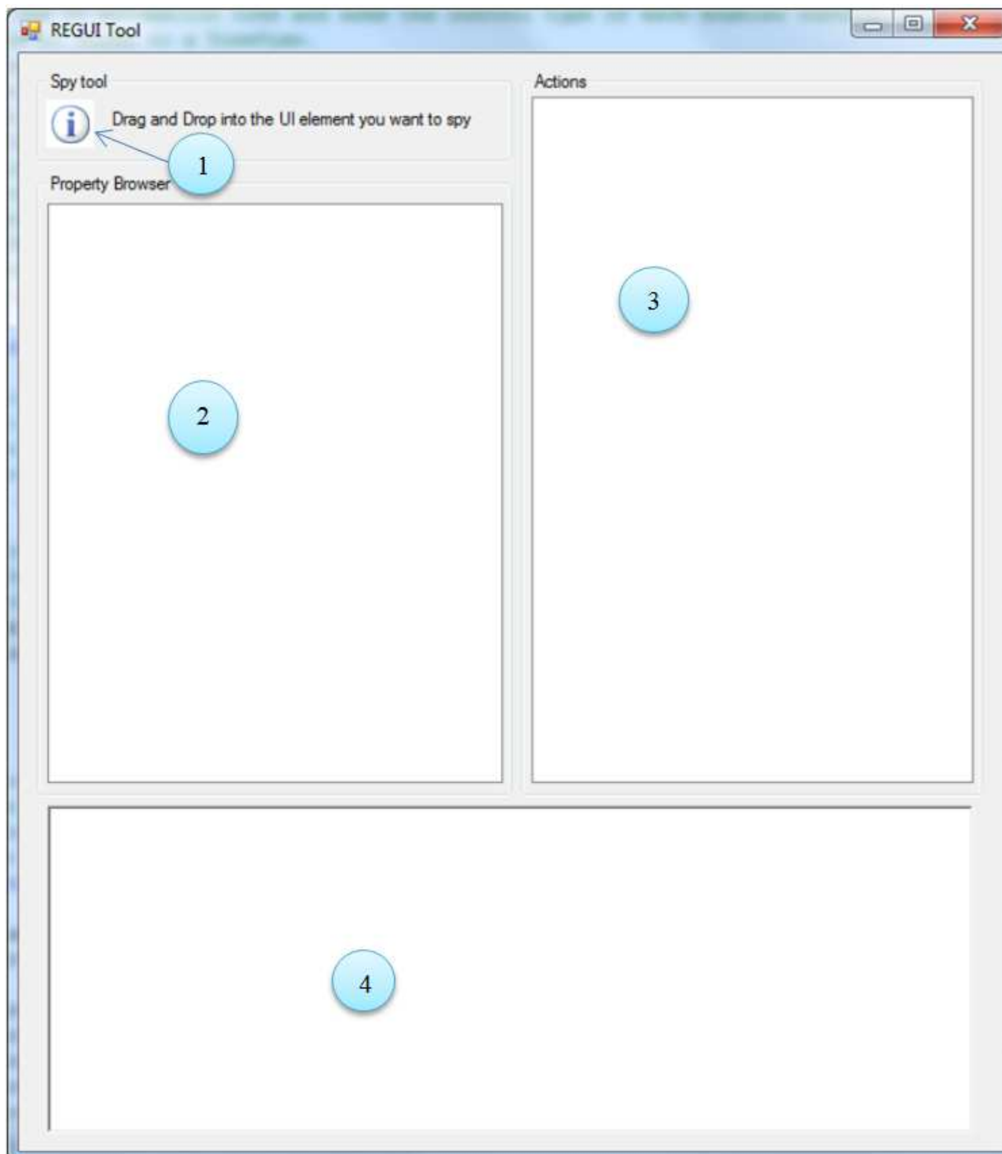

Figure 4.1 REGUI Tool – Initial Window

After executing the REGUI Tool is possible to identify, in Figure 4.2, the tree of GUI Controls presents in the window of the AUT, and some actions performed by the user.

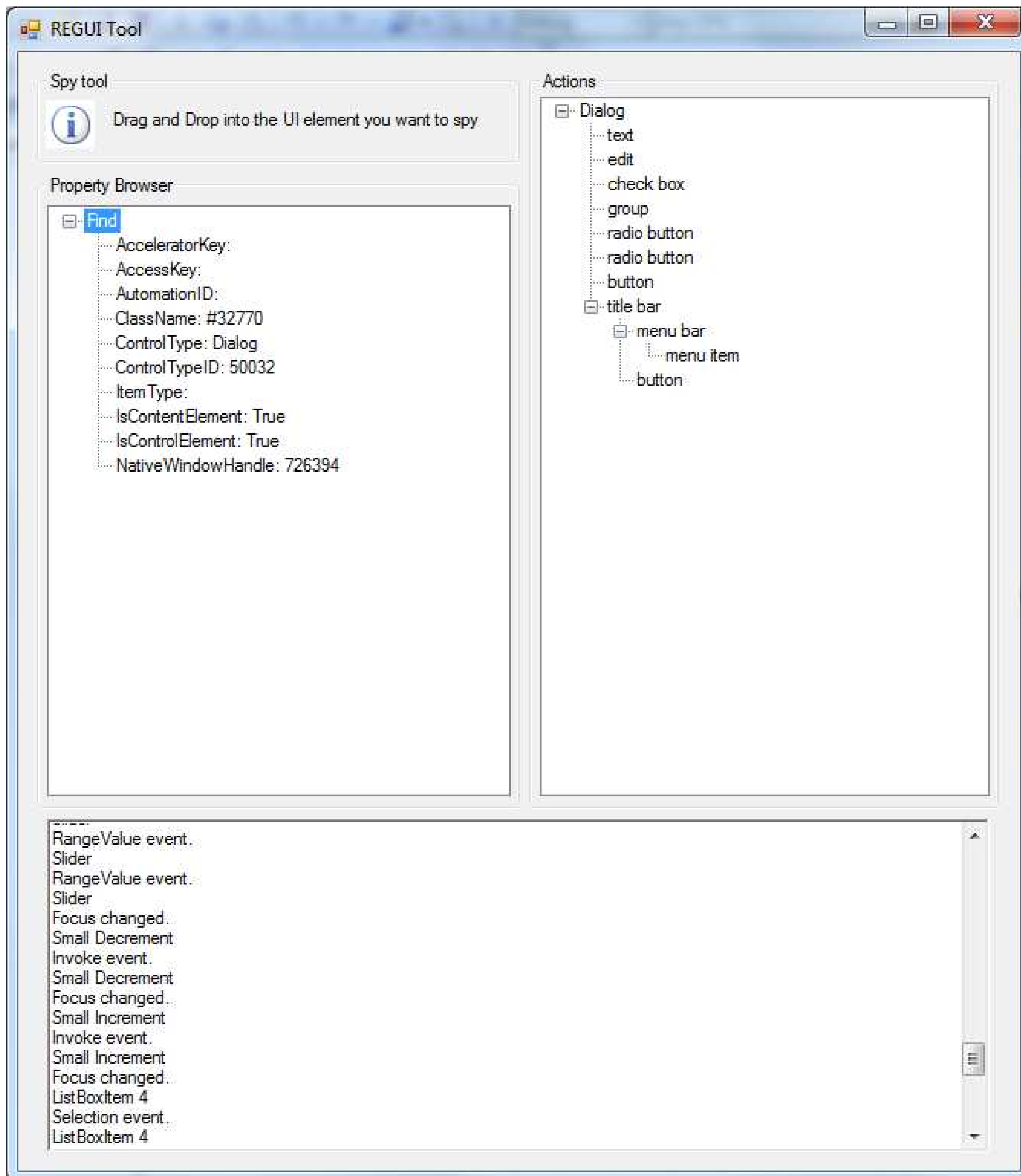

Figure 4.2 REGUI Tool

4.1.4 Internal structure of the application

In this chapter, the internal structure of the application will be detailed a little more. The Figure 4.3 shows the package diagram of the tool developed. The application consists in 3different file classes. Each file has a specific purpose and the main one – REGUI – can access to methods of the other files. The main file is the one that has the reverse engineering algorithm implemented. This file has several classes that are grouped as shown in Figure 4.4.

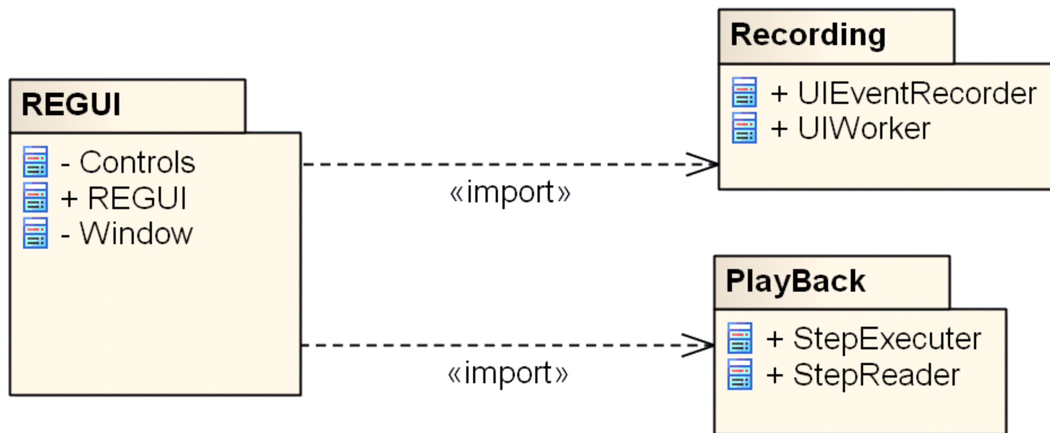

Figure 4.3 REGUI Tool Package Diagram

The only public class accessible by the application is the REGUI class. This includes functions that are used by the application to extract the structural information, to create and update the XML file that contains the extracted information, and the function that controls the recording mode. Other of the Package is the Recording, Figure 4.5.

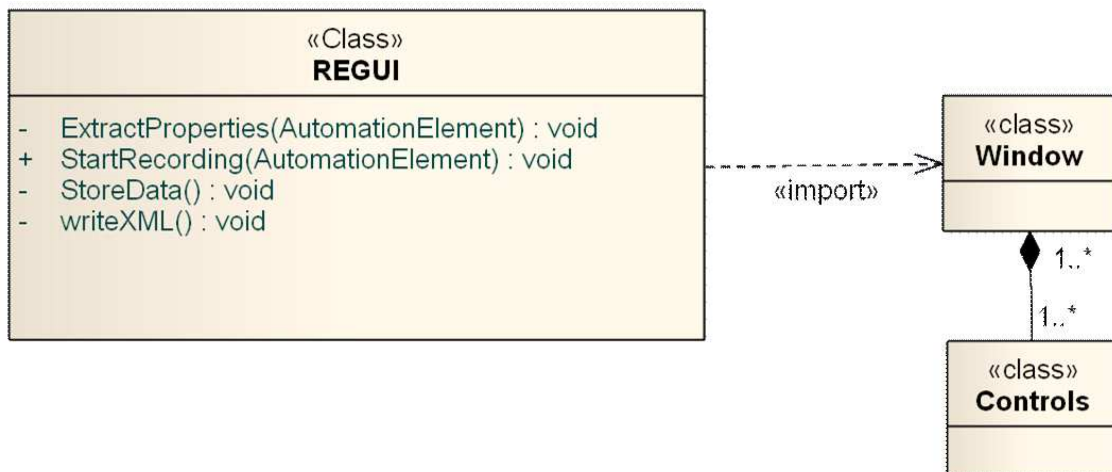

Figure 4.4 REGUI Package Diagram Class

This package includes two classes that give support for the recording mode; this means that both classes have methods to identify and save the different actions performed by the user over the AUT, in the XML file.

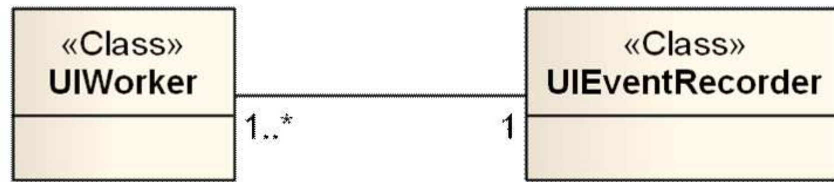

Figure 4.5 Recording Package Class Diagram

In the opposite way, the package Playback is responsible to replay the steps stored previously in the XML. The Playback package is composed by two main classes: The StepReader – where are the methods to read the steps from the XML file; StepExecuter – responsible for the simulation of the user actions (shown in the Figure 4.6).

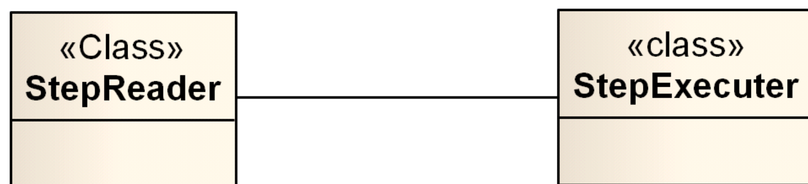

Figure 4.6 Playback Package class diagram

4.2 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, Figure 4.7:

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.

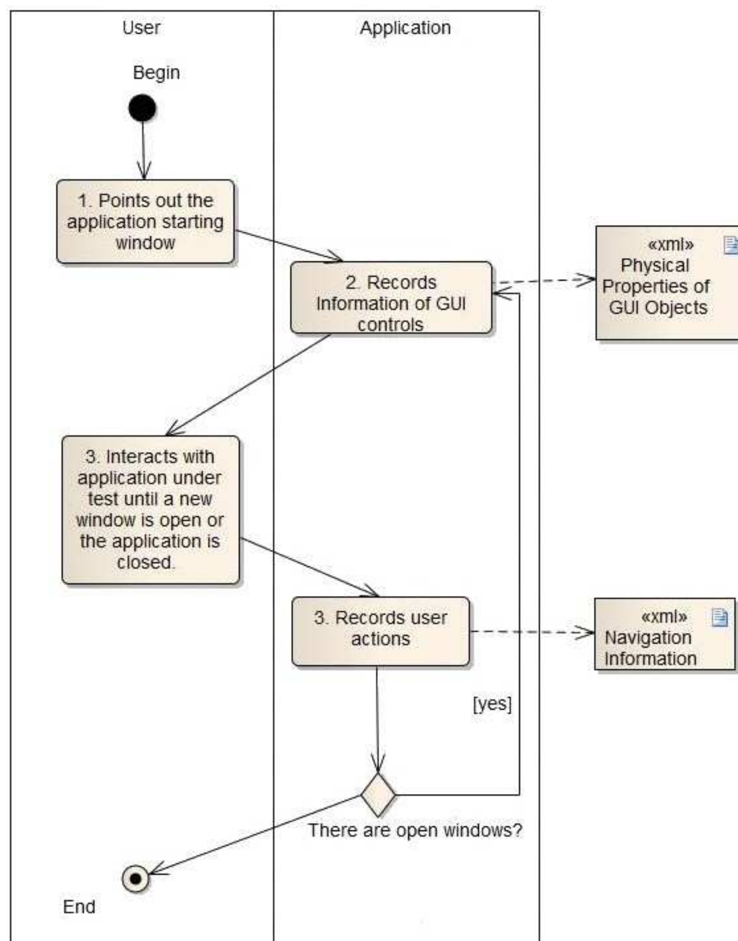

Figure 4.7 Gathering Structural Information

The second phase of the reverse engineering process aims to extract behavioural 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 behavioural information, Figure 4.8:

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.

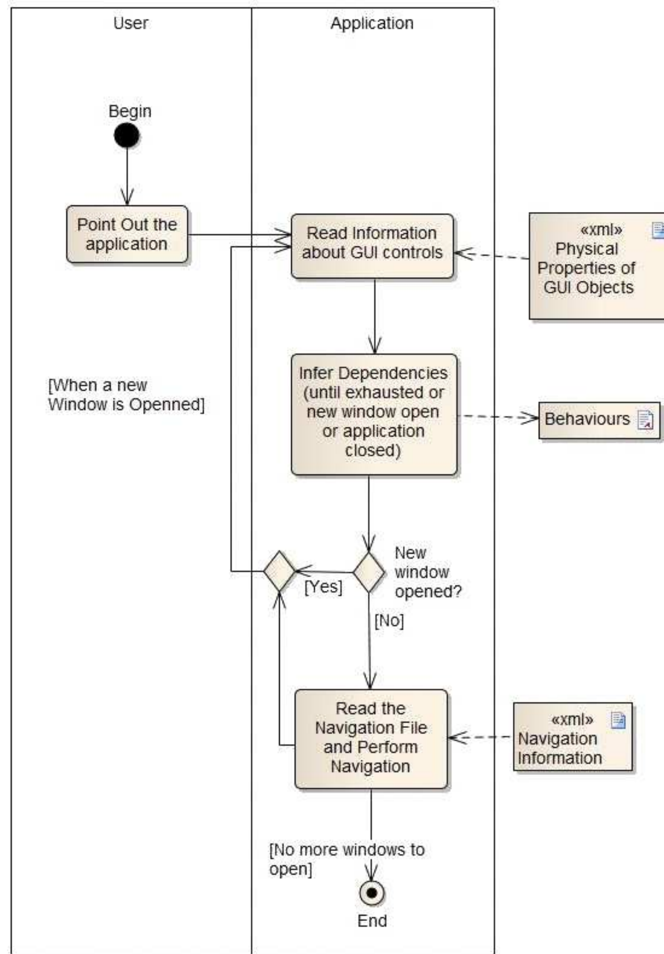

Figure 4.8 Gathering Behavioural Information.

4.3 Rules to Infer Behaviour

In order to identify the different kinds of behaviours that are common in GUIs, it is necessary to define some rules to infer the behaviour. To discover the different behaviours 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 $O = \{o_1, ..., o_N\}$ will be the Cartesian product of the domain values of the GUI controls properties, i.e., $S = dom(o_1.p_1)\times dom(o_1.p_2)\times...\times dom(o_1.p_m)\times...\times dom(oN.p1)\times dom(oN.p2)\times...\times dom(oN.pk)$ [35]. 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 `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 `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×PAR×O×S->S.

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= {<a,o,p,o'>|a:A, o:O, o':O, p is property of o', o≠o'}.

The algorithm to extract these dependencies is as follows:

Table 4.1 Formalization of the Behaviour Algorithm

```
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 ∪ {<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 ∪ {<a,o,p,o'>|o':O, p:Properties(o'.type)
            and s'.o'.p ≠ s.o'.p and o ≠ o'}
        s := 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 ParamValues (setText) = {'a','A','1'}.

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 in next section).
- 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|o.type=textbox}).
- Master detail dependency. An example of this behaviour is when updating a Combo Box or List box causes changes to the content of a grid or list view.

The next chapter will present a case study where it will be possible to see the results obtained by the application of the REGUI tool to extract the model of the Microsoft Notepad application.

Chapter 5

Case Study

This chapter presents a case study in which the model of the Microsoft Notepad application is extracted by the REGUI tool.

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>`).

5.1 Structural Information

Starting by the extraction of the structural information of the different windows of the application under test and saving that information in a XML file. The REGUI tool, after saving the structural information of the main window of the Notepad and recording the steps from this window until the Find window, extracts the structural information of this new window.

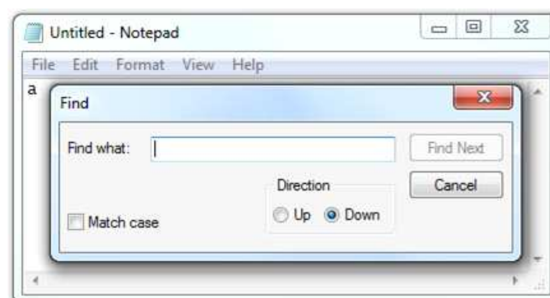

Figure 5.1 Find Dialog

An excerpt of the XML generated for the Figure 5.1 (Find Dialog) is shown in the Table 5.1.

Table 5.1 Structural Information of the Find Dialog

```
    <? xml version="1.0" encoding="utf-8”?>
    <application xmlns: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>
```

The structural information of the windows will be used in future phases of the Reverse Engineering algorithm.

5.2 Navigation Steps

In order to be possible to REGUI tool to know how to reach the different windows in the second phase of the reverse engineering algorithm, the navigational steps performed by the user are recorded in XML file. An excerpt of the XML file that contains the navigational steps between the main window and the Find Dialog (Figure 5.1) is shown in the Table 5.2.

Table 5.2 Navigational Steps

```
    <navigation>

      <Nav1>
            <source>Notepad</source>
            <destination>Find</destination>
            <steps>

                  <Nsteps>6</Nsteps>

                  <0>setText document = "aaa"</0>

                  <1>FocusChange Edit</1>

                  <2>Mouse Click</2>

                  <3>FocusChange Find</3>

                  <4>Mouse Click</4>

                  <5>FocusChange FindWhat</5>

            </steps>
```

```
    </Nav1>

    ...
  </navigation>
  </application>
```

5.3 Inferring Dependencies

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 behaviours in the several windows. One of the behaviours 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 Figure 5.2 and an excerpt of the generated information kept in XML file is reproduced in the Table 5.3.

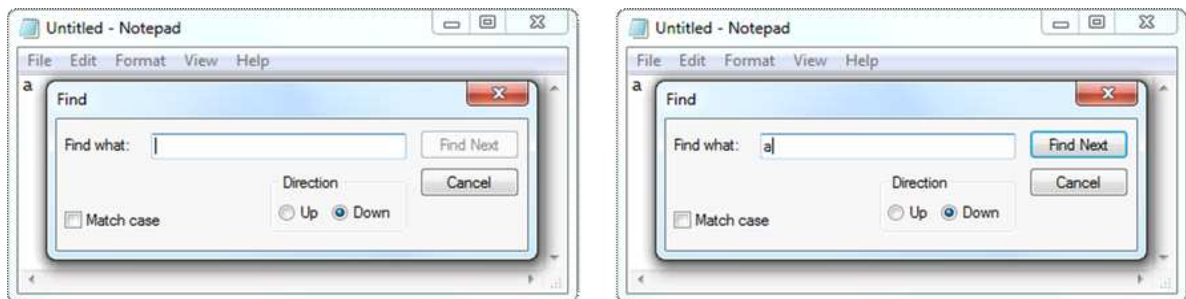

Figure 5.2 Enable/Disable Behaviour in Find Dialog

Table 5.3 Dependencies between GUI Controls

```
<dependency>
    <dep1>
        <window>Find What</window>
        <source>textbox Find what:</source>
        <destination>button Find Next</destination>
        <action>setText("a", Find What)</action>
        <property>FindNext.IsEnabled = true</property>
    </dep1>
    ...
</dependency>
```

5.4 The Spec# Model

After the creation of the XML file with the dependencies, the REGUI tool starts to translate the information into Spec#. The specification generated to represent the dependency identified in Figure 5.2 is shown below.

Table 5.4 Spec# Specification for the behaviour identified

```
[Action] void setTextFindWhat(string str)
```

```
modifies ButtonFindNext.IsEnabled;

{...}
```

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

Table 5.5 Rule for the method naming

```
<Action><caption>
```

and the name of the GUI controls follows the rule:

Table 5.6 Rule for the GUI Control naming

```
<classeType><caption>+ '.' + property
```

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

An excerpt of the Spec# model generated that represents the information extracted using the REGUI tool is available in the Table 5.7.

Table 5.7 Spec# Model

```
//Notepad window
[Action] void SetTextDocument(string typedText)
modifies Document.text && MenuItemFind.IsEnabled;
requires IsEnabled("Notepad");
{ //TODO}

[Action] void MenuItemFind()
requires text!="" && IsEnabled("Notepad") ;
ensures IsOpen("Find"));
{ //TODO}
…
//Find dialog
[Action] void SetTextFindWhat(string str)
modifies TextFindWhat.text && ButtonFindNext.IsEnabled;
requires IsEnabled("Find");
{
  TextFindWhat.text = str;
}
[Action] void FindNext()
requires IsEnabled("Find");
{ //TODO}
…
```

Chapter 6

Conclusions and Future Work

This chapter presents a summary of the main contributions of the work reported in this dissertation in the fields of reverse engineering of GUI applications, and points out topics that deserve future attention.

The theme of this research work is the automation of GUI testing. In the state of the art, we have concluded that the use of model-based testing tools lead to a higher degree of automation. However, this testing approach requires the model of the application under test, and the construction of this model requires a huge effort of time and resources. In order to diminish this effort, it is possible to extract a model of the GUI under test through a reverse engineering process. There are two different reverse engineering approaches: static (when the source code of the GUI under test is available) and dynamic (when the model extraction is performed by interacting with the GUI under test).

This dissertation describes an application that through a reverse engineering process, allows obtaining a model of the GUI's structure and some of its behaviour. This model is kept in a XML file from which a Spec# specification is generated to make GUI testing more systematic, thus improving overall GUI quality.

Pragmatically, we hope that the approach developed in this research work will be used effectively in industrial environments and henceforth contribute to higher quality interactive software. However, we are aware that the specification-based testing technique is not yet widely understood by testers and their managers. May this dissertation be also a contribution to disseminate the knowledge about methodologies and techniques to make testing activities more systematic, automatic, and less resource demanding.

6.1 Future Work

The algorithm presented to infer the GUI's behaviour 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. So this algorithm may be need a few tune up in order to become more efficient.

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; and improve the Spec# produced making it more complete.

A research paper was accepted in a national conference, and another one was submitted in an international conference for validation by peers.

REFERENCES

1. Merlo, E., et al., *Reengineering user interfaces.* Software, IEEE, 1995. **12**(1): p. 64-73.
2. Microsoft. *UI Automation*. msdn 2009 [cited 2009; Available from: http://msdn.microsoft.com/en-us/accessibility/bb892133.aspx.
3. Hendrickson, E., *Making the Right Choice*, in *Software Testing & Quality Engineering*. 1999. p. 21-25.
4. Chikofsky, E.J. and J.H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy*, in *Software, IEEE*. 1990. p. 13-17.
5. Ostrand, T., et al., *A visual test development environment for GUI systems.* SIGSOFT Softw. Eng. Notes, 1998. **23**(2): p. 82-92.
6. Beer, A., S. Mohacsi, and C. Stary. *IDATG: an open tool for automated testing of interactive software*. in *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*. 1998.
7. Memon, A.M., M.L. Soffa, and M.E. Pollack, *Coverage criteria for GUI testing*, in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. 2001, ACM: Vienna, Austria.
8. Memon, A.M., M.E. Pollack, and M.L. Soffa, *Using a goal-driven approach to generate test cases for GUIs*, in *Proceedings of the 21st international conference on Software engineering*. 1999, ACM: Los Angeles, California, United States.
9. Memon, A., I. Banerjee, and A. Nagarajan. *GUI ripping: reverse engineering of graphical user interfaces for testing*. in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*. 2003.
10. Paiva, A., et al., *A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing*, in *Formal Methods and Software Engineering*. 2005. p. 450-464.
11. Barnett, M., et al., *The Spec# Programming System: Challenges and Directions*, in *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. 2008, Springer-Verlag. p. 144-152.
12. Veanes, M., et al., *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, in *Formal Methods and Testing*. 2008. p. 39-76.
13. Systa, T., *On the relationships between static and dynamic models in reverse engineering Java software.* Reverse Engineering - Working Conference Proceedings, 1999: p. 304-313.
14. Moore, M.M. *Rule-based detection for reverse engineering user interfaces*. in *Proceedings of the Third Working Conference on Reverse Engineering*. 1996.
15. 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.
16. 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.
17. Group, O.M. *UML® Resource Page*. 1997 2009/01/08 [cited 2009; Available from: http://www.uml.org/.
18. P. Markopoulos, P.M. *UML as a representation for Interaction Design*. in *Proceedings of the OZCHI'00*. 2000.

19. Pinheiro da Silva, P. and N. Paton, *UML i : The Unified Modeling Language for Interactive Applications*, in *«UML» 2000 — The Unified Modeling Language*. 2000. p. 117-132.
20. Merlo, E., P.Y. Gagne, and A. Thiboutot. *Inference of graphical AUIDL specifications for the reverse engineering of user interfaces*. in *Software Maintenance, 1994. Proceedings., International Conference on*. 1994.
21. 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.
22. El-Ramly, M., et al., *Modeling the System-User Dialog Using Interaction Traces*, in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. 2001, IEEE Computer Society.
23. Kapoor, R.V.a.S., E.*Mathaino: Simultaneous legacy interface migration to multiple platforms*. in *In Proceedings of the 9th International Conference on Human-Computer Interaction*. 2001. New Orleans, LA, USA.
24. Moore, M., S. Rugaber, and P. Seaver. *Knowledge-based user interface migration*. in *Software Maintenance, 1994. Proceedings., International Conference on*. 1994.
25. Moore, M. and S. Rugaber. *Issues in User Interface Migration*. in *Proceedings of the Third Software Engineering Research Forum*. 1993.
26. Moore, M. and S. Rugaber, *Using Knowledge Representation to Understand Interactive Systems*, in *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*. 1997, IEEE Computer Society.
27. Lab, S.E.R., *TaMeX: Task-structure Based Mediation for Information Integration through XML*, U.o. Alberta, Editor. 1998, Department of Computing Science.
28. Stroulia, E., J. Thomson, and G. Situ. *Constructing XML-speaking wrappers for WEB applications: towards an interoperating WEB*. in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. 2000.
29. Situ, Q. and E. Stroulia, *Task-Structure Based Mediation: The Travel-Planning Assistant Example*, in *Proceedings of the 13th Biennial Conference of the Canadian Society on Computational Studies of Intelligence: Advances in Artificial Intelligence*. 2000, Springer-Verlag.
30. Lucca, G.A.D., et al., *An Approach for Reverse Engineering of Web-Based Applications*, in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. 2001, IEEE Computer Society.
31. Lucca, G.A.D., et al., *WARE: A Tool for the Reverse Engineering of Web Applications*, in *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*. 2002, IEEE Computer Society.
32. Dijkstra, E.W., *Chapter I: Notes on structured programming*, in *Structured programming*. 1972, Academic Press Ltd. p. 1-82.
33. Barnett, M., Rustan, and W. Schulte, *The Spec# Programming System: An Overview*.
34. W3C. *Extensible Markup Language (XML)*. 2009/04/16 [cited 2009; Available from: http://www.w3.org/XML/.
35. Paiva, A.C.R., *Automated Specification-Based Testing of Graphical User Interfaces*. Department of Electrical and Computer Engineering, 2007. **Ph.D**.

GLOSSARY

A

algorithm, 20, 21, 23, 24, 27, 31, 33

C

Cellest, 8
COBOL, 7, 10, 11
code coverage, 5, 15

D

DOS, 10, 11, 36
dynamic, 5, 12, 13, 15, 19, 35
dynamic approach, 5

G

Graphical User Interfaces, 1, 3, 37
GUI, 1, 2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 30, 32, 33, 34, 35,
GUI models, 2
GUI testing, ii, 1, 2, 3, 4, 14, 16, 17, 33, 35
GUITAR, iv, vii, 4

H

HTML, 11, 12, 13

I

IDATG, iv, vii, 4, 35

J

JAVA, 12

L

LeNDI, 8, 9

M

Mathaino, 9, 36
Model-based testing, 1, 4, 15

R

Reverse engineering, 2, 3, 5

S

Spec#, 5, 17, 18, 25, 31, 32, 33, 34, 35, 36
static, 5, 9, 12, 13, 15, 35
static approach, 5

T

TaMeX, iv, v, 11, 12, 36

U

UI, 3, 7, 8, 9, 11, 12, 35
UI re-engineering, 7
UML, 5, 12, 13, 36
UMLi, 5

W

Web Application Reverse Engineering, 12

X

XML, 11, 12, 18, 20, 21, 25, 28, 29, 30, 33, 36, 37