Dynamic Reverse Engineering of Graphical User Interfaces

Inês Coimbra Morgado and Ana C. R. Paiva
Department of Informatics Engineering,
Faculty of Engineering, University of Porto,
rua Dr. Roberto Frias, 4200-465 Porto, Portugal
{pro11016, apaiva}@fe.up.pt

João Pascoal Faria
Department of Informatics Engineering,
Faculty of Engineering, University of Porto
rua Dr. Roberto Frias, 4200-465 Porto, Portugal
INESC TEC, Porto, Portugal
jpf@fe.up.pt

***Abstract*—This paper presents a dynamic reverse engineering approach and a tool, ReGUI, developed to reduce the effort of obtaining models of the structure and behaviour of a software applications Graphical User Interface (GUI). It describes, in more detail, the architecture of the REGUI tool, the process followed to extract information and the different types of models produced to represent such information. Each model describes different characteristics of the GUI. Besides graphical representations, which allow checking visually properties of the GUI, the tool also generates a textual model in Spec# to be used in the context of model based GUI testing and a Symbolic Model Verification model, which enables the verification of several properties expressed in computation tree logic. The models produced must be completed and validated in order to ensure that they faithfully describe the intended behaviour. This validation process may be performed by manually analysing the graphical models produced or automatically by proving properties, such as reachability, through model checking. A feasibility study is described to illustrate the overall approach, the tool and the results obtained.**

***Keywords*-ReGUI; Dynamic Reverse Engineering; GUI Testing; Properties Verification; CTL; Model Checking; SMV**

I. Introduction

This paper extends the research work presented in [1], which describes a reverse engineering tool to extract a model from the execution of a Graphical User Interface (GUI). In particular, the state of the art is improved and the overall approach is described in more detail. Moreover, a new module was implemented in order to automatically generate a Symbolic Model Verification (SMV) model for model checking. The case study was extended in order to illustrate the additional features.

GUI models are key inputs for several advanced techniques, such as Model Based GUI Testing (MBGT) [2], [3], [4], which enables the automatic test case generation, increasing the systematisation and automation of the GUI testing process, and model checking, which enables an automatic verification and validation of some properties of the system. However, the manual construction of such models is a time consuming and error prone activity. One way of diminishing this effort is to automatically construct part of the model by a reverse engineering process.

The challenge tackled in this research work is the automatic construction of part of the software's GUI model (structure and behaviour) using a dynamic reverse engineering technique. The extracted information is presented in several formats that allow performing different types of analysis, such as, visual inspection, model checking and MBGT.

The term *reverse engineering* was firstly defined in 1985 by Rekoff [5] as *"the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system"*. Five years later, Chikofsky and Cross [6] adapted this definition to software systems: *"Reverse Engineering is the process of analysing a subject system to (1) identify the system's components and interrelationships and (2) to create representations of the system in another form or at a higher level of abstraction"*.

The origin of software reverse engineering lies on the necessity of improving and automating software maintenance. It is estimated that program comprehension [7], *i.e.*, understanding the structure and the behaviour of the software, corresponds to over 50% of software maintenance [8]. As such, developing tools which may aid software engineers on this task is of the utmost importance. Reverse engineering has already proved to be useful on this subject. For example, reverse engineering helped coping with the Y2K problem, with the European currency conversion and with the migration of information systems to the web and towards the electronic commerce [9]. For such reasons, the IEEE-1219 standard[1], which was replaced by the IEEE-14764 one[2], recommends reverse engineering as a key supporting technology to software maintenance [9].

In the last two decades, reverse engineering tools have evolved considerably and, nowadays, reverse engineering is useful for other fields of study rather than software maintenance, such as, software testing and auditing security and vulnerability.

According to Canfora *et al.* [10], nowadays, the main goals of reverse engineering are:

- recovering architectures and design patterns;

[1]IEEE Standard for Software Maintenance

[2]Standard for Software Engineering - Software Life Cycle Processes - Maintenance

- re-documenting programs and databases;
- identifying reusable assets;
- building traceability between software artefacts;
- computing change impacts;
- re-modularising existing systems;
- renewing user interfaces;
- migrating towards new architectures and platforms;
- testing and maintenance.

As every technology, reverse engineering techniques can also be used with malicious intent [11], like removing software protection and limitations or allowing unauthorised access to systems/data. However, the developers may use the same techniques in order to assure software's safety.

Yet in 1990, Chikofsky and Cross [6] divided the reverse engineering process in two parts: an *analyser*, which collects and stores the information, and an *abstractor*, which represents that information at a higher level of abstraction, *i.e.*, a model, either graphical or textual. Figure 1 depicts the representation of a common reverse engineering process.

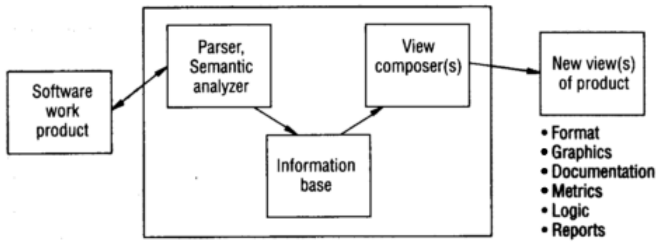


Fig. 1. Model of reverse engineering tools' architecture [6]

In Figure 1 the analyser is referred to as a *Parser, Semantic analyser* because, in the early years of reverse engineering, these were the most common techniques. Nowadays, besides static techniques [12], which extract information from the source code, there are other three different approaches [13], [14] for reverse engineering: dynamic, which extracts information from a running software system; hybrid, which mixes both static and dynamic approaches; and historical, which extracts information about the evolution of the system from version control systems, like SVN[3] or GIT[4]. Dynamic approaches have the advantages of being independent of the GUI implementation language and of not requiring access to source code. However, the existing dynamic approaches have limitations in terms of the behavioural information they are able to extract.

This paper describes a dynamic reverse engineering tool, ReGUI, developed to automatically extract structural and behavioural information from a GUI, including some behavioural information not obtained by other tools, like dependencies between GUI controls. ReGUI also distinguishes from other tools by producing multiple views and formats of the gathered information for different kinds of analysis: visual models enable a visual inspection of some properties, such as the number of windows of the GUI; textual models can be used for MBGT (Spec# [15] model) and to prove properties (SMV [16] model).

ReGUI v2.0 is fully automatic and uses a different approach from its previous version [17] so the results achieved, such as the extracted dependencies and the produced graphs, are different.

The rest of this paper is organised as follows. Section II presents the state of the art on user interface reverse engineering. Section III presents the proposed approach and the developed tool, ReGUI. Section IV presents the exploration process and the challenges faced during the development of ReGUI. Section V describes the outputs that can be obtained. Section VI presents a feasibility study on *Microsoft Notepad v6.1*, presenting the results obtained. Section VII presents some conclusions about this research work, along with the limitations of the approach and future work.

II. State of the Art

This Section presents the state of the art on reverse engineering, mainly on GUI reverse engineering, regarding the time interval from 2000 to 2011.

A. Static Analysis

As stated in Section I, static reverse engineering extracts information from textual sources, usually the source code, of the Application Under Analysis (AUA) [12]. The main techniques used in static analysis are code parsers, which are used to analyse the source code itself, query engines, which are used to verify certain facts of the code, and presentation engines, which are used to depict the query results [18].

Staiger's Approach: Staiger [19] presented, in 2007, an approach for the Bauhaus tool suite[5] [20] to statically reverse engineer the source code of a GUI, in order to support program understanding, maintenance and standards' analysis. Staiger claimed researchers had not focused their work on the static analysis of GUIs, even though most applications provided one. Staiger's approach is divided in three different phases: detecting the GUI elements, detecting widget hierarchies and detecting event connections. For the first phase, Staiger detects which data types on the source code had any connection to the GUI. Having detected these data types, he identifies the variables and respective parameters, as well as the functions or methods that are part of the GUI. After the source code elements are identified, the next phase finds the actual GUI elements, by identifying when each of the elements was created and the relationships among them. This provides the hierarchy of the elements. The third phase of the approach detects the different event handlers, the event they are handling and the element which triggers them. Along the execution, a window graph is generated containing all the extracted information on the several windows of the GUI. This approach

[3]svn.apache.org
[4]git-scm.com

[5]http://www.bauhaus-stuttgart.de/

was intended for C/C++ applications with a GUI implemented with GUI libraries, such as GTK[6] or Qt[7].

Lutteroth's Approach: Lutteroth [21] presented, in 2008, an approach whose goal was to automatically improve the layout of hard-coded GUIs. The approach extracts the GUI's structure and transforms it into a formal layout, Auckland Layout Model (ALM), which was defined by Lutteroth and Weber [22] in 2006. This approach extracts the structure of the GUI, by identifying its root element and navigating through its descendants. During this process, the position of each of the elements is mapped to a *tabstop*, which is a ALM property that represents a position in the coordinate system of a GUI.

Afterwards, the properties of each element, such as size and position, are updated, according to what best fits the GUI. For example, if an element has static content, a button for instance, then its size remains unaltered; otherwise, the best size is calculated according to its possible contents. In the end, the obtained layout may even be automatically improved. This may be done, for example, with the aid of some layout standards.

GUISurfer: In 2010, Silva *et al.* [23] developed the GUISurfer framework to test GUI-based Java applications, by following a static reverse engineering approach. The framework is composed by three tools: *File Parser*, *ASTAnalyser* and *Graph*. The first tool is responsible for the reverse engineering process, by parsing the GUI's source code and extracting behavioural information into an Abstract Syntax Tree (AST) [24]. Then, the second tool slices the information contained in the AST, focusing on the interface layer. In order to do so, the *ASTAnalyser* requires, besides the AST, the entry point of the application (the main method) and the set of GUI elements, which are part of the slicing process. In the end of this second phase, two files are generated: one containing the initial state of the GUI and one containing the events that may occur from the initial state. The third tool processes these two files and generates two Haskell specification files, which map the different events and conditions to actions on the GUI.

B. Dynamic Analysis

This Section describes existing dynamic reverse engineering approaches without and with code instrumentation.

1) Approaches Without Instrumentation:

GUIRipper: In 2003, Memon *et al.* [25] presented GUIRipper, a dynamic reverse engineering tool, which extracts behavioural information from the GUI of Java systems for testing purposes [26].

GUIRipper automatically interacts with the system's GUI, attempting to open as many windows as it can and, during this process, it extracts the GUI's structure and behaviour, producing three different artefacts. The GUI Forest is a graph representing the structure of the GUI. Each node represents a window of the GUI, containing the structure of its elements; an edge from a node *a* to a node *b* indicates that the window represented by the node *b* is accessible from the window represented by the node *a*. The second artefact is an event flow graph (EFG), which represents the behaviour of the GUI. Each node represents an event, such as *click on the button OK*; an edge from a node *a* to a node *b* indicates event *b* can follow event *a*. The third artefact is an integration tree, which relates the different components of the GUI. This last artefact is necessary to rip off the GUI into several components, generating EFGs for each one.

Amalfitano et al.'s Approach: Amalfitano *et al.*'s [27] presented, in 2008, an approach to reverse engineer Ajax [28] based Rich Internet Applications (RIAs) [29] as they claimed the problematic of modelling and validating this type of applications had not yet been explored thoroughly. They intended to fill this gap by dynamically extracting the behaviour of an application and representing it as a finite state machine (FSM) [30].

The analyser runs the RIA under analysis within a controlled environment and an event analysis takes place, *i.e.*, information on the sequence of events is extracted. The state chart diagram depicted in Figure 2 models this first phase, which includes two main states: waiting for an event to occur (*Event Waiting*) and waiting for an event handler to be complete (*Event Handling Completion Waiting*). Whenever an event is raised, information such as the type of the event, at what time it occurred and on which element it occurred is recorded. This process begins when the application starts.

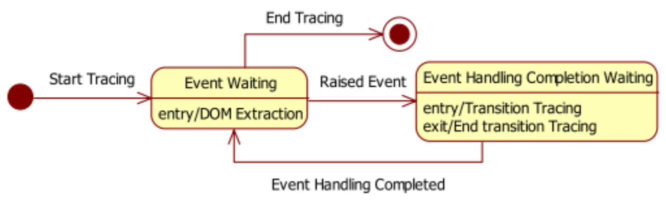


Fig. 2. The trace activity for the extraction step [27]

The second phase, abstraction, is composed by three steps. Initially, the information is transformed into a graph (a transition graph), which models the flow of client interfaces. The second step consists on using a clustering technique to analyse the transition graph in order to group equivalent nodes and edges. The clustering is based on the evaluation of several interface equivalence criteria, such as, the DOM structures including the same set of active element nodes and offering the same interaction behaviour to the users. This way, the issue of state explosion is dealt with. Finally, the FSM is generated with each state corresponding to each node of the clustered transition graph. This approach was validated by the development of a Java tool, RE-RIA, which implements both phases of the process.

2) Approaches With Instrumentation: There are some dynamic techniques that require source code (or byte code) instrumentation. Code instrumentation consists in inserting logging code into the existing one. As this is achieved during

[6]www.gtk.org
[7]qt.digia.com/

run time and without access to the code, they are considered dynamic approaches instead of hybrid ones.

Briand et al.'s Approach: Briand *et al.* proposed, in 2006 [31], an approach to dynamically extract behavioural information from Java distributed communications, namely Remote Method Invocation applications. The extracted information is represented as UML sequence diagrams. Even though there may be several applications to these diagrams, they intended to test the consistency of the code with the design.

Briand *et al.* divided their approach in two phases. The first phase consists in the instrumentation of the source code. In order to make their approach as little intrusive as possible, Briand at al. used Aspect Oriented Programming [32]. The second phase analyses the execution traces, creates the corresponding models and transforms them into scenario diagrams.

As in every dynamic analysis strategy, the extracted information is limited to the extent of the system's exploration and to the context in which each action was executed. This way, Briand *et al.* defined two meta-models: one to describe the information extracted from the execution traces and another to describe what they called scenario diagrams, which are UML sequence diagrams but limited to the context (scenario) of the execution. In order to transform the first meta-model into the second, they defined rules in the Object Constraint Language.

Finally, Briand *et al.* claimed that one of the biggest advantages of their approach was the usage of meta-models and transformation rules as these are formalised and can be easily improved and compared to others.

Safyallah and Sartipi: In 2006, Safyallah and Sartipi [33] presented an approach to identify the features of a system by identifying sequential patterns in the execution traces of the system. In order to do so, Safyallah and Sartipi divided their approach into two phases. In the first phase, the execution traces are extracted. This is achieved by setting scenarios, which are based in the domain of the application, the documentation and the familiarity of the user with the system, to examine each feature, and by source code instrumentation (inserting the name of the function at the beginning and at the end of each of them). Executing the scenarios provided the execution traces. In the second phase, a sequential pattern mining algorithm was applied to the extracted traces in order to obtain the most frequent sequential patterns. Figure 3 depicts the type of patterns identified: with this type of analysis, it is possible to identify, for example, that a *lock* is eventually followed by an *unlock*.

```
1  A B C D E A X B C
2  A G X B C
3  A X B C
```

Fig. 3. Sequential pattern: the sequence *ABC* is repeated [34]

This enabled the identification of generic functionalities (common to the different features of the system) and the ones that were feature-specific. With this approach, Safyallah and Sartipi were able to ease program comprehension and feature to source code assignment.

Alafi's Approach: In 2009, Alalfi [35] also presented an approach and a tool (PHP2XMI), which intended to extract behavioural information by instrumentation of the source code of the AUA and analysing the generated event traces. The ultimate goal of this approach is to ease the security analysis and testing of PHP-based[8] web applications.

The PHP2XMI tool functions in three steps. The first step corresponds to the code instrumentation. This step enables the extraction of information on page *URL*s, *http* variables, sessions and cookies. The second step executes the application, generating the execution traces, which are then filtered to ignore redundant information, and storing the relevant information in a SQL database. The third and final step transforms the stored data into UML 2.1 sequence diagrams [36]. These diagrams can be depicted by any UML 2.1 tool set.

C. Hybrid Analysis

Hybrid analysis provides an improvement of the completeness, scope and precision of the extraction as it mixes both static and dynamic approaches, trying to maximise the amount of extracted information [13]. This Section presents some of the works that follow this line of research.

Systä's Approach: In 2000, in her dissertation, Systä [37] presented an approach combining the advantages of both static and dynamic analyses, with special focus on the dynamic part, for reverse engineering a Java software system.

The static part consists in parsing the system's byte code with a byte code extractor in order to extract the system's structure. This information is represented as a graph, which can be visualised with the Rigi reverse engineering environment, developed by Müller *et al.* [38]. Afterwards, the system is run under a customised jdk debugger, JDebugger, producing dynamic event trace information, control flow data, which was represented as scenario diagrams. These diagrams could be visualised with the SCED dynamic modelling tool [39], which transforms them into a single state diagram.

Systä developed a prototype, Shimba, which applies the described approach, integrating the Rigi system with the SCED tool. Without disregarding the other applications of the extracted information, debugging is presented as being a very useful one.

Frank et. al's Approach: In 2001, Frank *et al.* [40] presented an approach to dynamically reverse engineer a mobile application for Android, iOS or Java ME. They extract a model of the life cycle, which can be used to detect errors, like verifying if an application's information is saved when it has to be interrupted, *e.g.*, save the text of an e-mail when an incoming call occurs. Even though the reverse engineering process itself is processed during run-time, it is necessary to previously alter the source code, which makes this an hybrid approach.

Frank *et al.*'s approach was divided in four phases. The first and second ones are of the responsibility of the developer as

[8]http://www.php.net/

they consist in programming the life cycle's code, overwriting every call-back method called in life cycle changes, and inserting logging code to all the overwritten methods. In the third phase, black-box tests [41] are applied to the mobile application in order to identify the different triggers. For this, it is of the utmost importance that the first two phases have been processed thoroughly. In the fourth phase, the information extracted in the previous phase is used to derive an application's life cycle model and to identify properties of the application at certain states of the life cycle. The model is a state diagram of the application. The states can be, for example, *running*, *paused*, *background*. The transitions are labelled according to the corresponding actions, *e.g.*, *onCreate()* and *onStop()*. This model may be useful in several contexts, such as verifying the consistency of the application's life cycle or identifying properties of the application at a given state.

D. Discussion

Apart from the work of Lutteroth *et al.* [21], which only extracts structural information from GUIs to improve layout, the analysed reverse engineering approaches are similar to ours because they extract both structural and behavioural information. However the purpose of each approach may be different: program comprehension (for testing and/or maintenance) [19], [23], [25], [27], [31], [42]; debugging [37]; properties verification [40]; feature to source code mapping [33]; and security analysis [35]. The purpose of the presented approach is program comprehension and properties verification.

Regarding the information extracted, there are similarities regarding structural information (GUI elements, their properties and hierarchy relations) the set of approaches extract but varieties regarding behavioural information. Some approaches extract events (their handlers and the relations between them) [19], [23], [25], [27]; sequence of actions [27], [31], [33], [35], [37]; sequential patterns [33]; and lifecycle of the system [40]. The presented approach extracts structural information, alike the remaining approaches, and behavioural information on navigation and on dependencies between the different GUI controls.

Another characteristic that distinguishes the several studied approaches is the representation (abstractor) of the extracted information. Most of the approaches represent their information in only one structure: sequence diagrams [31], [35], [37]; state diagrams [27], [40]; specification file [21]; graphs [19] or sequence patterns [33]. There are only two approaches which opt to represent the information in more than one way: Silva *et al.* [23] extracted both a specification file and an AST and Memon *et al.* [25] extracted a window graph and an event flow graph. As far as the authors know, there is only one approach that enables verification of properties [40], but it focuses strictly on the life cycle of mobile applications.

In addition, most of the approaches can only be applied to one platform (web [27], [35] or mobile [40]), or to one language (Java [23], [25], [31], [37] or C/C++ [19]). The approach described in this paper uses UI Automation that allows extracting information from desktop and web applications, which increases the range of supported platforms.

III. ReGUI Overview

The goal of this research work is to diminish the effort of producing visual and formal models of the GUI of a software application. The approach followed is the extraction of the information from the GUI under analysis by a dynamic reverse engineering approach. This way, this approach is independent of the programming language in which the GUI was written, broadening its applicability. This Section presents an overview of the proposed approach.

A. Architecture and Outputs

Figure 4 depicts an overview of the approach proposed in this paper.

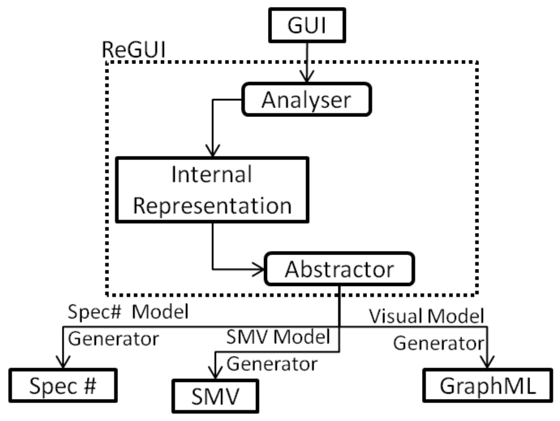


Fig. 4. Architecture and outputs obtained with the ReGUI tool

The analyser is responsible for the exploration of the GUI and extraction of the structural and behavioural information. The exploration process is discussed in more detail in Section IV-A. The abstractor is responsible for representing the extracted information in different ways: visual models, which enable a quick visual inspection; a Spec# model, which can be used for MBGT; and a SMV model, which enables the automatic verification of properties. These models are explained in detail in Section V.

B. Extracted Information

Figure 5 represents the information extracted by ReGUI. *GUI Elements* can be *Windows* or *Controls* that may be initially enabled or disabled. *Windows* may be *modal* (in which case it is not possible to interact with other windows of the same application while this one is opened) or *modeless* (it is possible to interact with other windows). *Windows* are composed of *Controls*, which may be *menu items* or others. The elements in the diagram (classes and relationships) are annotated with graphical symbols used in the visual models generated by ReGUI.

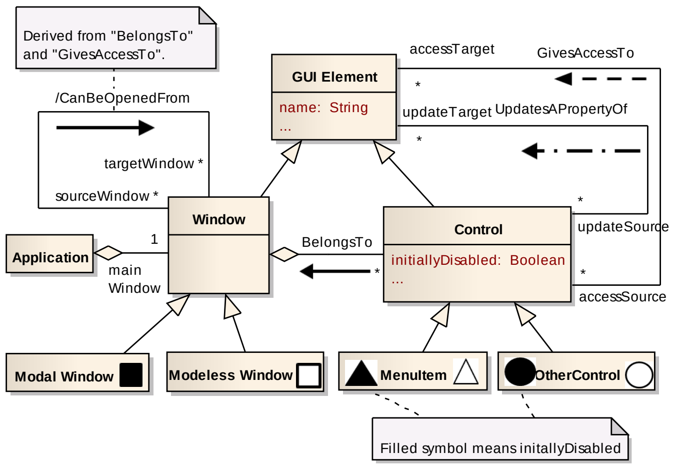


Fig. 5. Annotated metamodel of the models generated by the ReGUI tool (see Figure 4)

The associations between the objects represent the extracted behaviour. When interacting with a control, there are five possible identifiable outcomes:

- *open* - a window is opened;
- *close* - a window is closed;
- *expansion* - new controls become accessible. For instance, the expansion of a menu;
- *update* - one or more properties of one or more elements are updated. For instance, the name of a window is modified or an enabled control becomes disabled (or *vice-versa*);
- *skip* - nothing happens.

The first three outcomes are represented by the *GivesAccessTo* relation, while the third one is represented by the *UpdatesAPropertyOf* relation. If a control of a window (*BelongsTo* relation) opens another window (*GivesAccessTo* relation), then there is a *CanBeOpenedFrom* relation between the second and the first windows. This makes the last relation a derivation from the two previous ones.

C. Front-End

The ReGUI front-end is shown in Figure 6.

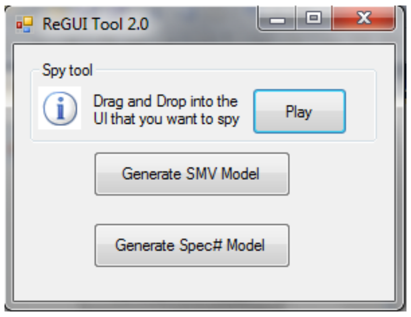


Fig. 6. ReGUI front-end

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. At the end of the execution, all models except the Spec# and the SMV ones have already been generated. As such, the user may press the *Generate SMV Model* button in order to generate the SMV model, as well as, the *Generate Spec# Model* button in order to generate the Spec# model.

IV. ReGUI Analyser

In this Section, the analyser process is described, namely the exploration process and the challenges tackled during the implementation.

A. Exploration Process

In general, a dynamic exploration can be classified according to its automation, manual or automatic, and to whether or not it is guided. If the exploration is automatic not being guided means it is random whilst a guided exploration would require some heuristic to determine which control should be explored at each instance. If, otherwise, it is manual, being guided means the user actions are driven by a particular goal, whilst not being guided would just indicate the user has complete freedom in choosing the next control.

Given the goal of this approach, the exploration must be performed automatically, remaining the choice of being guided or not. If the random exploration took long enough, eventually the entire interface would be explored. However, the amount of time a company has is limited and, thus, this approach is not ideal. On the other hand, the guided exploration depends completely on the algorithm used for the exploration.

As in most situations, the best solution would be to get the advantages of each approach: follow a guided approach mixed with a random one and, if and when the exploration hits a breakpoint, the tool should ask the user to interact with the GUI in order to move forward with the automatic exploration. ReGUI follows a guided exploration based on the order of the elements.

The exploration process is divided in two phases. First, ReGUI navigates through every menu option in order to extract the initial state of the GUI, *i.e.*, which GUI elements are enabled/disabled at the beginning of the execution, in the main window. For the second phase, ReGUI navigates through all the menus and interacts with the ones enabled at that instance. After each interaction, ReGUI verifies if any window has opened. If so, ReGUI extracts its structure, closing it afterwards. Following, ReGUI goes through all the menus again in order to verify if any state changed, *i.e.*, if a previously enabled element became disabled or *vice-versa*. All the information extracted is organised in internal structures, which are described in Section V.

In order to interact with the GUI, ReGUI uses UI Automation [43], which 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, children of the application to which they belong. In the UI Automation framework each of these elements is an *Automation Element*.

B. Challenges

During the development of ReGUI, it was necessary to face some challenges:

1) Identification of GUI elements: GUI elements may have dynamic properties, *i.e.*, properties which may vary along the execution, such as the *RunTimeIdProcess* and the *Name*, and do not have a property which uniquely identifies them. During the exploration process, the identification of an element is performed by comparing its properties with the ones of other elements. There are some that, when used for comparing two elements, undoubtedly distinguish them when their values are different. For example, if two controls are a *button* and a *menu item*, then they are necessarily different. However, this sort of properties may not be sufficient. As such, an heuristic based on some properties was implemented to compare two elements: an element *a* is considered to be the same as an element *b* when it is the one which most resembles element *b*, considering a minimum threshold. The properties to be used in the comparison can be configured in the beginning of the execution.

2) Exploration order: In general, the extracted information depends on the order in 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 one. The exploration of the children of a node follows the order in which they appear on the GUI. However, if the exploration followed a different order, the dependencies extracted would be different. An example of such may be found in the *Microsoft Notepad v6.1* application and is depicted in Figure 7. The menu item *Select All* requires the presence of text in the main window in order to produce any results. Since there is no text in the main window, in the beginning, interacting with this menu item does not have any effect. However, after interacting with the *Time/Date* menu item, which writes the time and date in the main window, the *Select All* menu item would produce visible results: it would select the text, enabling the menu items *Cut*, *Copy* and *Delete* and disabling the *Select All* menu item itself.

3) Synchronisation: To automatically interact with a GUI, it is necessary to wait for the interface to respond after each action. 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. For example, after expanding a menu, its submenus are added to the *UI Automation* tree as its children, launching an event. The event handler catches it and ReGUI acts accordingly. When verifying whether or not a window opened, there is an event handler similar to the one used to catch a menu expansion. However, when invoking an element for the first time, there is no way of previously knowing if any event will occur. This way, after invoking an element, ReGUI waits either for the event handler to catch the event or for a defined amount of time.

4) Closing a Window: During the execution it is necessary to close windows that are eventually opened, in order to continue 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, ReGUI looks for one of these buttons to close it: *cancel*, *no*, *close*, *ok*, *continue* or *x*.

V. ReGUI Abstractor

ReGUI generates different views on the extracted information. Each of these views represents different aspects of the structure and behaviour of the GUI under analysis, enabling a rapid visual inspection of such aspects. The current views ReGUI is able to extract are a tree representing the structure of the GUI and the hierarchy between the different elements, and four graphs representing its behaviour. Every node of these four graphs corresponds to a node in the tree. The information stored in these structures is used to generate the formal models both in Spec# and in SMV. The next sub-sections describe these different outputs, explaining the type of information represented in each of them. The Figures referred along this Section are examples of outputs and can be depicted along Section VI.

A. Structural Information: 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. An examples is depicted in Figure 12.

B. Behavioural Information

Extracting behavioural information is useful for different purposes, such as modelling the GUI behaviour, generating test cases, proving properties or usability analysis. This Section describes the different views generated by ReGUI on the behavioural information extracted.

1) Navigation Graph: The navigation graph represents the nodes 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. A visual representation of this graph is depicted in Figure 13. A solid edge between a window *w1* and a GUI element *e1* means *e1* is inside of *w1* whilst a dashed edge between two GUI elements *e1* and *e2* means *e2* becomes accessible after interacting with *e1*.

Figure 8 is a subset of Figure 5 of Section III and depicts the information extracted by ReGUI that is represented in this graph, as well as the graphical symbols used.

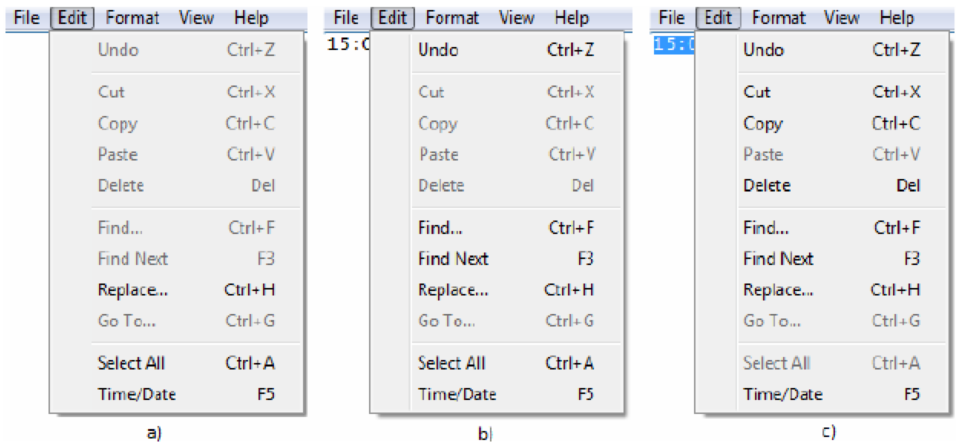


Fig. 7. Menu item *Edit* on *Microsoft Notepad v6.1*: a) after invoking the menu item *Select All* and before invoking the menu item *Time/Date*; b) after invoking the menu item *Time/Date*; c) after invoking again the menu item *Select All*

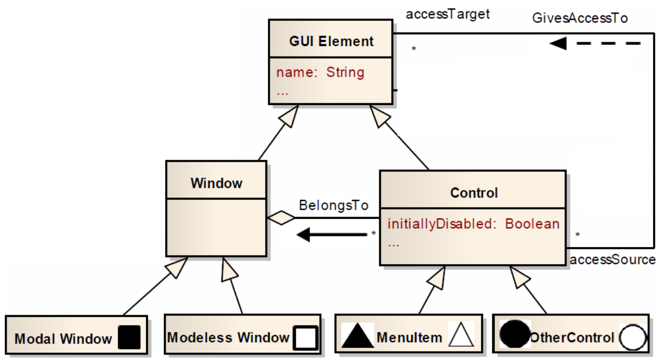


Fig. 8. Representation of the different elements and their relationships in the navigation graph

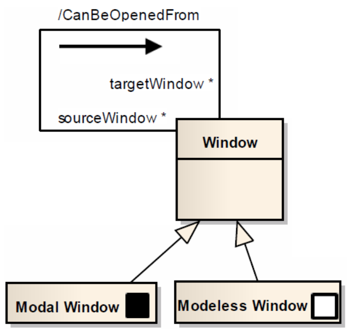


Fig. 9. Representation of the different elements and their relationships in the window graph

2) Window Graph: The window graph shows a subset of the information represented by the navigation graph. It describes the windows that may be opened in the application. Figure 14 is a visual representation of this graph. A window may be modal or modeless. An edge between two nodes *w1* and *w2* means that it is possible to open window *w2* by interacting with elements of window *w1*.

Figure 9 is a subset of Figure 5 of Section III and depicts the information extracted by ReGUI that is represented in this graph.

3) Disabled Graph: The disabled graph's purpose is to show which nodes are accessible but disabled int he beginning of the execution (obtained during the first phase of the exploration process described in Section IV-A). The enabled property of an element may vary during the second phase but that modification is not represented in this graph. An example of this graph is depicted in Figure 15. The nodes correspond to some GUI elements, being filled when disabled and empty when enabled. A solid edge between two nodes *n1* and *n2* means that *n2* belongs to *n1*. On the other hand, a dashed edge between those nodes means *n2* is accessible after interacting with *n1*.

Figure 8 is also applicable to this graph as the relations between the controls have the same meaning, even though the information represented in both graphs is different.

4) Dependency Graph: A dependency between two elements *A* and *B* means that interacting with *A* modifies the value of a property of *B*. An example of a dependency would be if interacting with *A* enabled a previously disabled *B*. Figure 16 is the visual representation of a dependency graph obtained during an exploration process. A solid edge between a window *w1* and a node *n1* means *n1* is accessed from *w1* and a dashed edge between two nodes *n1* and *n2* means there is a dependency between *n1* and *n2*.

Figure 10 is a subset of Figure 5 of Section III and depicts the information extracted by ReGUI that is represented in this graph.

C. Spec# Model

Spec# is a formal specification language that can be used as input to the model-based testing tool Spec Explorer [44], for automatic test generation.

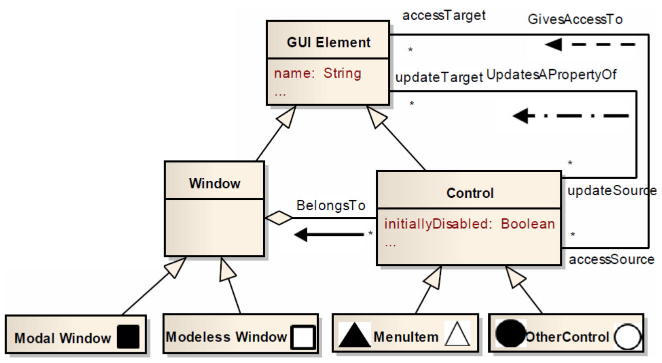


Fig. 10. Representation of the different elements and their relationships in the dependency graph

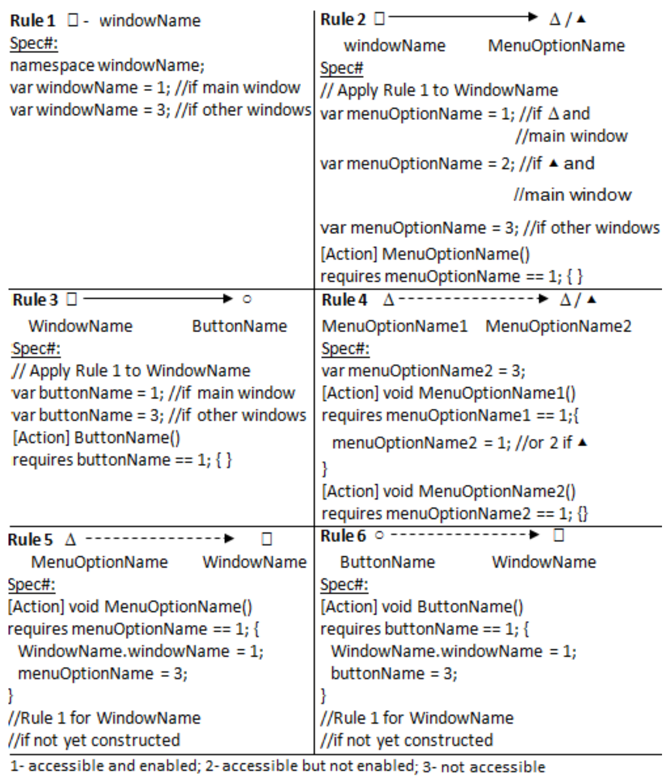
The Spec# model is obtained by applying the rules in Figure 11 on the navigation graph. Each window generates a namespace and each edge generates a method annotated with *[Action]*. Action methods in Spec# are methods that will be used as steps within the following generated test cases. Methods without annotations are only used internally. All the elements relevant to the navigation are represented as variables (*var*) having three possible values: *1*, if the element is accessible and enabled; *2*, if the element is accessible but disabled; and *3*, if the element is not accessible. At the beginning of the execution, the only possibly accessible elements are the ones belonging to the main window, as every other window is not accessible itself. Every variable and method corresponding to elements belonging to a certain window must be placed under the namespace representing that window. An example of a model generated by the application of these rules is in Figure 17.

D. SMV Model

After obtaining a model one problem that may rise is how to verify properties on it. This may be tackled by model checking techniques, which verify if given properties are valid on the model under analysis. The verification of properties can be very useful, for example, in usability analysis and improvement [45], [46]. The one used in this approach is symbolic model checking [47]. Properties are expressed in Computation Tree Logic, which is a propositional temporal logic, and the system is modelled as a FSM.

Some of the properties that can usually be verified are *reachability*, *i.e.*, if it is possible to reach every node, *liveness*, *i.e.*, under certain circumstances, something will eventually happen, *safety*, *i.e.*, under some circumstances, something will never happen, *fairness*, *i.e.*, under certain conditions, something will always happen, and *deadlock-freeness*, *i.e.*, the system does not get into a cycle from which it cannot come out.

With this approach, the state machine representing the system is generated automatically based on the navigation graph. Alike this graph, each state represents a GUI element, which is represented by a unique *id*. The first state corresponds



Rule 1 □ - windowName
Spec#:
namespace windowName;
var windowName = 1; //if main window
var windowName = 3; //if other windows
	Rule 2 □ ⟶ Δ / ▲
	windowName MenuOptionName
	Spec#
	// Apply Rule 1 to WindowName
	var menuOptionName = 1; //if Δ and
	//main window
	var menuOptionName = 2; //if ▲ and
	//main window
	var menuOptionName = 3; //if other windows
	[Action] MenuOptionName()
	requires menuOptionName == 1; { }

Rule 3 □ ⟶ ○
WindowName ButtonName
Spec#:
// Apply Rule 1 to WindowName
var buttonName = 1; //if main window
var buttonName = 3; //if other windows
[Action] ButtonName()
requires buttonName == 1; { }
	Rule 4 Δ - - - -▸ Δ / ▲
	MenuOptionName1 MenuOptionName2
	Spec#:
	var menuOptionName2 = 3;
	[Action] void MenuOptionName1()
	requires menuOptionName1 == 1;{
	menuOptionName2 = 1; //or 2 if ▲
	}
	[Action] void MenuOptionName2()
	requires menuOptionName2 == 1; {}

Rule 5 Δ - - - -▸ □
MenuOptionName WindowName
Spec#:
[Action] void MenuOptionName()
requires menuOptionName == 1; {
WindowName.windowName = 1;
menuOptionName = 3;
}
//Rule 1 for WindowName
//if not yet constructed
	Rule 6 ○ - - - -▸ □
	ButtonName WindowName
	Spec#:
	[Action] void ButtonName()
	requires buttonName == 1; {
	WindowName.windowName = 1;
	buttonName = 3;
	}
	//Rule 1 for WindowName
	//if not yet constructed

1- accessible and enabled; 2- accessible but not enabled; 3- not accessible

Fig. 11. Rules for the Spec# generation

to the main window, having *id 1*. The relations *belongsTo* of the navigation graph were eliminated for this representation because they do not describe user actions.

This model is imported to the SMV tool[9] and is composed of three modules:

- *getInfo(id)*, where further information about the states may be represented. In this case, the type of GUI elements can be *1* for window, *2* for menu item and *3* for other controls;
- *getNextState(id)*, which represents information about the next state (*e.g.*, how many and which states follow a given state);
- *main*, in which the state machine is described along with the specification of the properties to be verified.

Figure 18 depicts an example of the SMV description of a state machine.

VI. Feasibility Study

In order to check and test the feasibility of the approach presented in Section IV, ReGUI was run on *Microsoft Notepad v6.1*. In this Section the different outputs resultant from this experiment are presented and analysed. The window, navigation, disabled and dependencies graphs were visualised with a template for Microsoft Excel, NodeXL[10].

[9]http://www.cs.cmu.edu/~modelcheck/smv.html
[10]http://nodexl.codeplex.com/

A. Structural Information

Figure 12 is a simplified representation of the menu structure of *Notepad* upon the exploration of its menu item *File*. At this point, the ReGUI tree has more information than the presented in this Figure as the *UI Automation* tree contains plenty of elements. However, these do not add any relevant information to the structure and were, therefore, removed from the example provided in this document.

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

Fig. 12. Part of the ReGUI tree when exploring the menu item *File*

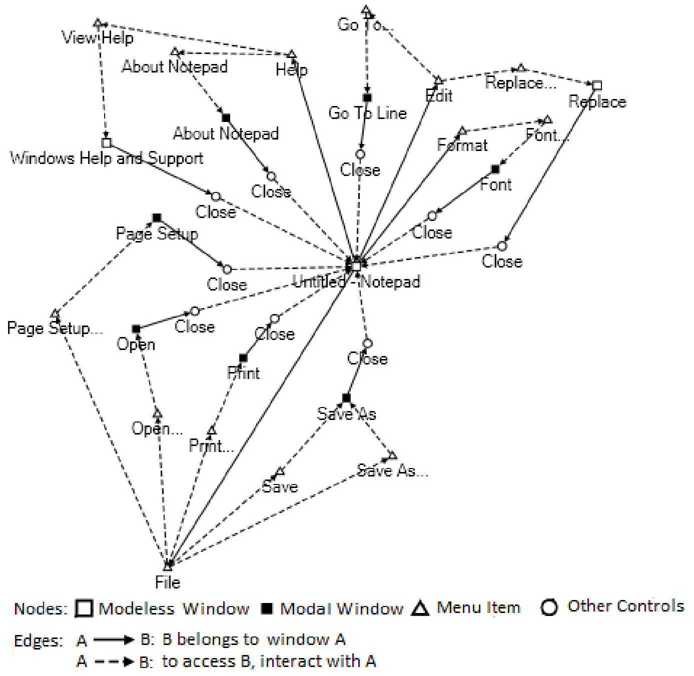


Fig. 13. Visual representation of the navigation graph

B. Behavioural Information

Apart from the structure, behavioural information is also extracted and stored in four internal structures. In this Section, an example of each of the graphs corresponding to those structures, which have already been described in Section V, is presented.

Figure 13 shows the visual representation of the navigation graph. In this example, it is possible to depict that 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* in order to open the *Save As* window. Clicking on this window's button *Close* closes it and the main window gets the focus again.

The visual representation of the window graph is represented in Figure 14. In this case, it is possible to see 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, which is modeless.

Figure 15 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 parent of these menu items.

Figure 16 is the visual representation of the dependency graph. With this graph it is possible to detect dependencies among GUI controls and analyse whether or not it behaves as expected. For instance, interacting with the menu item *Word Wrap* provokes a modification on the *isEnabled* property of the menu items *Undo* and *Go To...* (there is a dashed edge from

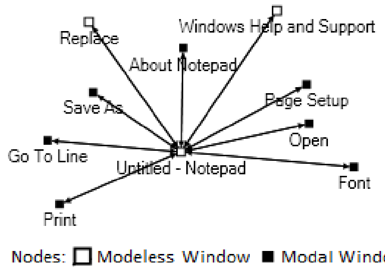


Fig. 14. Visual representation of the window graph

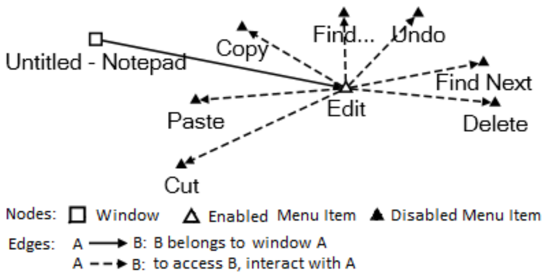
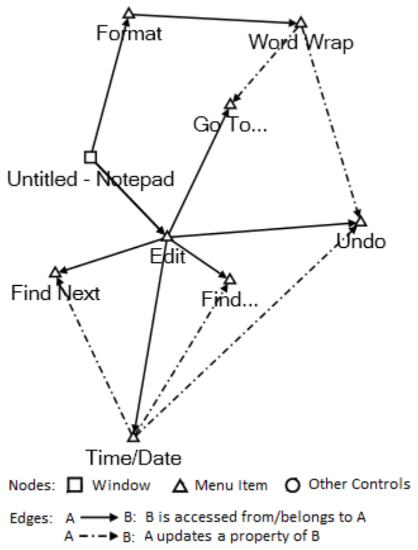

Find...
Undo
Copy
Untitled - Notepad
Find Next
Edit
Paste
Delete
Cut
Nodes: Window Enabled Menu Item Disabled Menu Item
Edges: A → B: B belongs to window A
A - - → B: to access B, interact with A

Fig. 15. Visual representation of the disabled graph

Word Wrap to *Undo* and to *Go To...*) and interacting with the menu item *Time/Date* may alter the *isEnabled* property of the menu item *Undo* (there is also a dashed edge between these nodes). As such, a visual inspection over the graph may be enough for the tester to detect some abnormalities in the GUI's behaviour.



Format
Word Wrap
Go To...
Untitled - Notepad
Edit
Undo
Find Next
Find...
Time/Date
Nodes: □ Window △ Menu Item ○ Other Controls
Edges: A ⟶ B: B is accessed from/belongs to A
A -·-▸ B: A updates a property of B

Fig. 16. Visual representation of the dependency graph

C. Spec# Model

Using the extracted information it is possible to obtain another kind of model: a Spec# model, which is a formal representation of the behaviour of the GUI. At this moment, the Spec # model only represents the information gathered on the navigation graph.

Figure 17 depicts a small sample of the generated Spec# model for this case study. The rules applied to generate this Spec# model, presented in Figure 11, are enumerated 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.

After validating and completing the model, it can be used, for example, as input for the MBGT approach described in [48].

D. SMV Model

In order to verify properties on the extracted information, it goes through a transformation process to SMV. Until now, only the information represented by the navigation graph is used to verify properties. The navigation graph is automatically transformed into a SMV state machine (see Figure 18).

Figure 18 depicts the representation of the state machine in the SMV model for this case study. The state machine has an initial state (*init(state)*) and transitions (*next(state)*). The meaning of each transition is described in comments (−−). Three variables have been declared: *state*, which corresponds

```
namespace WindowUntitled___Notepad;
var windowUntitled___Notepad = 1;
var menu_itemFile = 1;
var menu_itemSave = 3;
[Action] void Menu_itemFile ()
      requires menu_itemFile == 1;{
            menu_itemSave = 1;
      };
[Action] void Menu_itemSave()
      requires menu_itemSave == 1;{
            menu_itemSave = 3;
            WindowSave_As.windowSave_As = 1;
      };

namespace WindowSave_As;
var windowSave_As4 = 3;
var buttonClose = 3;
[Action] ButtonClose ()
      requires buttonClose == 1;{
            buttonClose = 3;
            WindowUntitled___Notepad.
                  windowUntitled___Notepad = 1;
      };
```

Fig. 17. Sample of the Spec# formal model generated

to the *id* of the control represented by that state; *follow*, which has two attributes: *state*, which corresponds to the set of possible next states, and *num*, which indicates the number of possible next states; and *moreInfo*, which has the attribute *type* that represents the type of the control corresponding to that state.

With this state machine, it is possible to verify several properties to evaluate, for instance, usability properties, such as:

- regardless of the current state, it is always possible to reach the main window (state =1):
  AF state = 1;
- check the presence of deadlocks:
  !(EF (AG (follow.num = 1 & state in follow.state))).
  It checks if when there is only one out transition, the next state is different from the current state;
- regardless of the current state, it is possible to go back to the main window in *x* steps (three, *e.g.*):
  EBF 1..3 state = 1;
- when on the main window, there is always a window *x* steps away (three, *e.g.*):
  state = 1 − > ABF 1..3 moreInfo.tipo = 1.

Running the SMV model for the *Microsoft Notepad* application, it is possible to state that:

- it is always possible to reach the main window, regardless of the state;
- no deadlocks were detected;
- it is always possible to get to the main window in three steps;
- there is always a window three steps away.

VII. Conclusions and Future Work

ReGUI is capable of extracting important information about the behaviour of the AUA, such as navigational information

```
MODULE main
    VAR
        state: 1..20;
        follow: getNextState(state);
        moreInfo: getInfo(state);

       ASSIGN
        init(state) := 1;
        next(state):=
            case
                state = 1: {2, 4, 6, 7, 9, 11, 13, 15, 17, 19};
                --from state 1, it is possible to go to states 2 (Open...),
                --4 (Save), 6 (Save As...), 7 (Page Setup), 9 (Print…),
                --11 (Replace...), 13 (Go To...), 15 (Font...),
                --17 (View Help), 19 (About Notepad)
                state = 2: 3;  --goes to window Open
                state = 3: 1;  --goes to the main window
                state = 4: 5;  --goes to window Save As
                state = 5: 1;  --goes to the main window
                state = 6: 5;  --goes to window Save As
                state = 7: 8;  --goes to window Page Setup
                state = 8: 1;  --goes to the main window
                state = 9: 10;  --goes to window Print
                state = 10: 1;  --goes to the main window
                state = 11: 12;  --goes to window Replace
                state = 12: 1;  --goes to the main window
                state = 13: 14;  --goes to window Go To Line
                state = 14: 1;  --goes to the main window
                state = 15: 16;  --goes to window Font
                state = 16: 1;  --goes to the main window
                state = 17: 18;  --goes to window Windows Help and Support
                state = 18: 1;  --goes to the main window
                state = 19: 20;  --goes to window About Notepad
                state = 20: 1;  --goes to the main window
            esac;
```

Fig. 18. State machine in SMV

and which GUI elements become enabled or disabled after interacting with another element. The exploration process is fully automatic, with the user just having to point out the AUA.

The outputs generated by the ReGUI tool are extremely useful for program comprehension and for program verification as the graphs can be used to verify some important properties, such as reachability and deadlock-freeness, and the Spec# model can be used for test case generation and platform migration, for example. Even though the ReGUI tool does not generate the totality of the Spec# model, it already provides an important part of it.

The static and hybrid approaches have, by definition, a different purpose than the one presented in this paper as they require the source code, contrary to dynamic approaches. Comparing with other dynamic approaches, it is possible to conclude this approach extracts more information. Memon's approach [25] extracts information on the structure and the relation between the different events, which is represented by the *ReGUI Tree* and the navigation and window graphs, whilst this approach also extracts information on the dependency between the different controls. Similarly, Amalfitano's approach [27] is focused on the events (when they are raised and completed) and not on the dependency part. Briand *et al.* [31], Safyallah and Sartipi [33] and Alafi's [35] approaches require instrumentation, even though they are considered dynamic approaches, which makes these approaches more intrusives than the approach presented in this paper. Moreover, the behavioural information extracted enables proving different properties through model checking. None of the analysed approaches provides such analysis.

The main difficulties faced during the development were the lack of GUI standards and the necessity of synchronisation. ReGUI has still some limitations. For instance, currently, it only supports interaction through the *invoke pattern* 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.

The future work will be focused on solving these limitations, on improving the exploration of the GUI, *i.e.*, interact with the different controls more than once and in different orders and on improving the Spec# generation. It is also intended to apply this approach to other platforms, such as web and mobile.

VIII. Acknowledgements

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

References

[1] I. Coimbra Morgado, A. Paiva, and J. Pascoal Faria. Reverse Engineering of Graphical User Interfaces. In *The Sixth International Conference on Software Engineering Advances (ICSEA '11)*, number c, pages 293–298, Barcelona, 2011.

[2] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, November 2007.

[3] Ana C. R. Paiva, João C. P. Faria, and Raul F. A. M. Vidal. Specification-based testing of user interfaces. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *10th International Workshop on Interactive Systems. Design, Specification, and Verification (DSV-IS '03)*, pages 139—-153, Funchal, Portugal, 2003.

[4] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes. Reverse engineered formal models for GUI testing. In *The 12th international conference on Formal methods for industrial critical systems*, pages 218–233, Berlin, Germany, July 2007. Springer-Verlag.

[5] MG Rekoff. On Reverse Engineering. *IEEE Trans. Systems, Man, and Cybernetics*, (March-April):244 – 252, 1985.

[6] E.J. Chikofsky and J.H. Cross. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[7] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *The 15th international conference on Software Engineering (ICSE '93)*, pages 482–498, May 1993.

[8] Thomas A. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[9] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, and Margaret-Anne Storey. Reverse engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering - ICSE '00*, pages 47–60, New York, New York, USA, May 2000. ACM Press.

[10] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142, April 2011.

[11] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

[12] David Binkley. Source Code Analysis: A Road Map. In *Future of Software Engineering (FOSE '07)*, pages 104–119. IEEE, May 2007.

[13] Thoms Bell. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, November 1999.

[14] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, March 2007.

[15] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec\# Programming System: An Overview. In *International Conference in Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*, pages 49–69, Marseille, France, 2004. Springer.

[16] Kenneth L. McMillan. *Getting Started with SMV*. Cadence Berkley Labs, 2001 Addison St., Berkley, CA, USA, 1999.

[17] A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria. Reverse engineering of GUI models for testing. In *The 5th Iberian Conference on Information Systems and Technologies (CISTI '10)*, number July, pages 1–6. IEEE, 2010.

[18] Alexandru Telea, Lucian Voinea, and Heorhiy Byelas. Architecting an Open System for Querying Large C and C++ Code Bases. *S. African Computer Journal*, 41(December):43–56, 2008.

[19] Stefan Staiger. Static Analysis of Programs with Graphical User Interface. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 252–264. IEEE, 2007.

[20] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus A Tool Suite for Program Analysis and Reverse Engineering. *In Reliable Software Technologies, Ada Europe 2006*, page 71, 2006.

[21] Christof Lutteroth. Automated reverse engineering of hard-coded GUI layouts. In *The 9th conference on Australasian user interface (AUIC '08)*, pages 65–73. ACM, January 2008.

[22] Christof Lutteroth and Gerald Weber. User interface layout with ordinal and linear constraints. In *The 7th Australasian User Interface Conference (AUIC '06)*, pages 53–60, January 2006.

[23] João Carlos Silva, Rui Gonçalo, João Saraiva, and José Creissac Campos. The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code. In *2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 181–186, Berlin, 2010. ACM.

[24] Nicola Howarth. Abstract Syntax Tree Design. Technical Report August 1995, Architecture Projects Management Limited, 1995.

[25] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *The 10th Working Conference on Reverse Engineering (WCRE '03)*, 2003.

[26] Daniel R. Hackner and Atif M. Memon. Test case generator for GUITAR. In *Companion of the 13th international conference on Software engineering (ICSE Companion '08)*, ICSE Companion '08, page 959, New York, New York, USA, 2008. ACM Press.

[27] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Reverse Engineering Finite State Machines from Rich Internet Applications. In *The 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 69–73. IEEE, October 2008.

[28] Jesse James Garrett. Ajax: A New Approach to Web Applications. *Adaptive Path*, 2005.

[29] Cameron O'Rourke. A Look at Rich Internet Applications. *Oracle Magazine*, 2004.

[30] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[31] L.C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.

[32] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *The 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[33] H. Safyallah and K. Sartipi. Dynamic Analysis of Software Systems using Execution Pattern Mining. In *The 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 84–88. IEEE, 2006.

[34] Tao Xie, Suresh Thummalapenta, and D Lo. Data mining for software engineering. *IEEE Computer*, 42(8):55–62, 2009.

[35] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. A verification framework for access control in dynamic web applications. In *The Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering (C3S2E '09)*, page 109, New York, New York, USA, May 2009. ACM Press.

[36] Object Management Group. UML 2.1.2, 2012.

[37] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Phd, University of Tampere, 2000.

[38] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering (CASCON '93)*, CASCON '93, pages Volume1: 217–226. IBM Press, 1993.

[39] K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto. Automated support for modeling OO software. *IEEE Software*, 15(1):87–94, 1998.

[40] Dominik Franke, Corinna Elsemann, Stefan Kowalewski, and Carsten Weise. Reverse Engineering of Mobile Application Lifecycles. In *18th Working Conference on Reverse Engineering (WCRE '11)*, pages 283–292. IEEE, October 2011.

[41] Glenford J. Myers. Art of Software Testing. March 1979.

[42] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automated Reverse Engineering of UML Sequence Diagrams for Dynamic Web Applications. In *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*, pages 287–294. IEEE, April 2009.

[43] Rob Haverty. New accessibility model for Microsoft Windows and cross platform development. *SIGACCESS Access. Comput.*, (82):11–17, 2005.

[44] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, Lev Nachmanson, Robert Hierons, Jonathan Bowen, and Mark Harman. Model-based testing of object-oriented reactive systems with spec explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Models and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[45] Fabio Paternò and Carmen Santoro. Integrating Model Checking and HCI Tools to Help Designers Verify User Interface Properties. In *7th International Workshop on Interactive Systems Design, Specification and Verification*, Limmerick, Ireland, 2001.

[46] Nadjet Kamel, Sid Ahmed Selouani, and Habib Hamam. A Model-Checking Approach for the Verification of CARE Usability Properties for Multimodal User Interfaces. *International Review on Computers & Software*, 4(1):152—-160, 2009.

[47] E Clarke, K McMillan, S Campos, and V Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer Berlin / Heidelberg, 1996.

[48] Ana C. R. Paiva, João C. P. Faria, Nikolai Tillmann, and Raul A. M. Vidal. A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing. In *7th International Conference on Formal Engineering Methods (ICFEM '05)*, pages 450–464, 2005.