VNU Journal of Science: Comp. Science & Com. Eng., Vol. xx, No. x (2018) 1–10

A method of Automated User Interface Testing for Windows-based Applications

Duong Dinh Tran*, Hung Manh Nguyen, Pham Ngoc Hung

Faculty of Information Technology,
VNU University of Engineering and Technology,
E3 Building, 144 Xuan Thuy Street, Cau Giay, Hanoi, Vietnam

Abstract

This paper proposes a method of automated user interface testing for Windows-based applications. Given an application under test, the key idea is to generate new test scenarios from its widgets and its specification. These widgets are extracted during the execution of the applications, and the specification is generated by combining the interactions of widgets. Besides, the paper contributes a technique to detect hidden widgets which is considered as one of the most challenging problems in user interface testing. Currently, a tool named GTA has been implemented to demonstrate the effectiveness of the proposal in practice and it has been tested with several industrial projects. The experiment has illustrated that this proposed method could detect more hidden widgets in comparison with that of Ranorex Spy and UI Spy.

Received May 2018, Revised , Accepted

Keywords: Automated testing, graphical user interface testing, Windows application

1. Introduction

User interface (UI) provides the ability to allow interactions between humans and underlying systems. Thus, checking the correctness of UI is considered as an important phase in the software quality assurance. However, UI is becoming more and more complicated, hence manual testing has difficulties in both design and execution steps. Furthermore, it must be performed not only one time but also every time when developers modify the source code of applications [1]. Therefore, UI testing should be performed automatically instead of manual testing to reduce the efforts of testers and increase productivity. Because Windows operating system accounts for a larger market share than others such as Linux or Mac OS, hence this paper focuses on presenting a method of automated UI testing for Windows-based applications.

There are two main approaches to automatically check the correctness of UI including *Model-based testing* and *Record & Replay* [2]. The former uses

* Corresponding author. Email: 14020084@vnu.edu.vn

models to represent system behaviors and user interactions. The graph model is the popular one which each node represents an interaction of a widget. The test scenarios are created by traversing all possible paths in the graph. Some well-known tools applying this approach can be mentioned UFT [3], PETTool [4], PATH [5]. The later is another technique used to check the correctness of UI interactions. This method consists of two major phases. In the first phase, all events that testers performed are recorded. After that, these interactions can be replayed multiple times automatically. Currently, there are many automated UI testing tools which are developed based on Record & Replay technique. The most famous free tools can be referred to as Mouse Recorder[1], Reran[2], etc. There are also several commercial tools such as TestComplete [6], Ranorex[3].

However, both two mentioned approaches still contain some problems. Firstly, in *Model-based testing*, testers need to represent user scenarios using the formal specification method. It means that testers should have the mathematical knowledge, hence they find it hard to use when applying this approach in automated UI testing. Secondly, in *Record & Replay* technique, although testers do not need to have mathematical knowledge, they must record all test scenarios manually to create test scripts. However, in practice, the number of test scenarios is usually large, hence recording all scenarios is a task which is waste of time, cost, and efforts.

[1]`http://www.mouserecorder.com/`
[2]`http://www.androidreran.com/`
[3]`https://www.ranorex.com/`

This paper introduces a method of automated UI testing for Windows-based applications to mitigate the mentioned problems. There are three main phases of the proposed approach including widgets inspection, test scenarios specification analyzer and test scripts generation, test execution. The goal of the first phase is to extract all widgets and their properties from the Application Under Test (AUT). From these widgets, testers would specify the desired test scenarios. After that, the test scenarios specification is analyzed to generate test scripts to produce a test project. Finally, this test project is executed on a specific compiler to obtain the results of test scripts (e.g., *pass/fail*).

The rest of this paper is organized as follows. At first, Section 2 introduces some of the basic concepts used in this research. Next, Section 3 presents the widgets inspection phase. After that, Section 4 describes test scenarios specification and test scripts generation. The details of test execution are presented in Sect. 5. Section 6 describes a tool that was implemented from the proposed method and experimental results. Finally, the conclusion of the paper is shown in Sect. 7.

2. Related work

Many works have been proposed for the test scenario generation by several authors. Focusing on the most recent and the closest ones, we can refer to [2] [7] [8] [9] [10].

Record and replay [2] is a technique used to check the correctness of interactive applications with graphical user interfaces. In the UI testing, this technique is used more widely than Model-based UI testing because

it does not require mathematical knowledge. Record and replay consists of two phases corresponding to its name. Firstly, all the events that be performed by users are recorded (e.g., keys pressed, mouse movements, etc.). After that, these events can be exactly replayed multiple times automatically. This is a great advantage of this technique because it assists to save much testers' efforts in executing test. Testers find it easy to use record and replay tools for UI testing. Record and replay technique is effective in practice for UI testing in comparison with traditional manual methods. Thus, there was much attention in developing automated UI testing tools based on record and replay approach for applications on various platforms [7] [8] [9] [10].

Microsoft provides a library named UI Automation on .NET framework allows for programmatic access to most widgets on the Windows-based applications. It enables assistive technology products, such as screen readers, to provide information about the UI to end users and to manipulate the UI by means other than standard input. By using UI Automation and following accessible design practices, developers can make applications running on Windows more accessible to many people with vision, hearing, or motion disabilities. Also, UI Automation is specifically designed to provide robust functionality for automated testing scenarios. This library exposes every piece of the UI to client applications as an "*Element*". *Elements* are contained in a tree structure, with the desktop as the root element. Clients can filter the raw view of the tree as a control view or a content view.

3. Widgets inspection

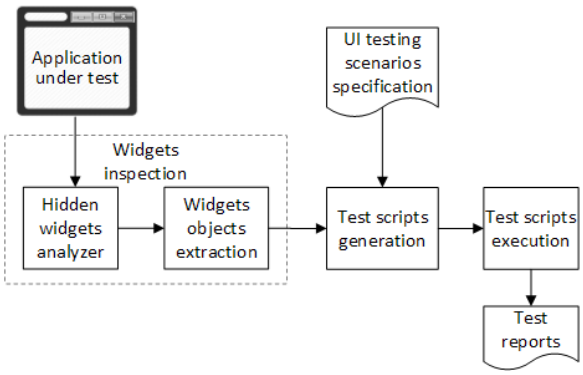

Figure 1. The overview of the proposed method

The proposed method can be divided into three main phases (Fig. 1). Firstly, all widgets are extracted from AUT during the execution of it. One of the challenges in this phase is that some hidden widgets cannot be detected, hence a technique is introduces to overcome this problem. Then, a collection of UI test scenarios are generated by combining the interactions of each widget from the specification. After that, the specification of test scenarios is analyzed to generate test scripts and export a test project. Finally, the test project is built and run to export UI test reports.

3.1. Widgets objects extraction

After AUT was launched, the process identifier (PID) of the application is retrieved. By using this PID, widgets are found recursively from the root elements corresponding the windows of the application. For each widget, the following properties need to be retrieved.

- **type:** there are several kinds of widgets such as window, button, text box, radio button, tree view, etc.
- **id:** this is an attribute which was assigned value by the programmer in software developing process
- **name:** a string, it is usually the same as the content of this widget.
- **parent** and **children**
- **size:** height, width of widget
- **location:** relative location of widget
- **screen capture of widget**
- other properties: e.g., state (enable or disable), visibility, etc.

3.2. Hidden widgets in opened windows analyzer

When a window of AUT is opened, all widgets in this window need to be extracted. However, this objective may not be achieved because it contains some hidden widgets. For example, in a combo box at the normal state, except the current selection which is being selected, the remain selections are invisible. In order to deal with this problem, several appropriate actions are performed automatically whenever encountering a certain widget type to visible widgets which were hidden before. The details of the solution are shown in Table 1. In the mentioned example, the applicable action is to click on the drop-down button on the right of this combo box.

Table 1. Handling hidden widgets in opened windows

Widget type	Widgets are maybe hidden	Actions
tree view	children nodes when parent is not expand	expand all nodes that contain children
tab	widgets in unselected tab items	in turn select each tab item, extract all elements in this tab item
combo box	selections that are not being selected	click drop-down button on the right

3.3. Hidden widgets in unopened windows analyzer

In fact, a Windows desktop application contains more than one window, but not all of them are opened immediately after the application is launched. However, there is no way to find widgets in the windows that have not be opened yet. Thus, all windows of AUT should be opened before extracting process in order to avoid ignoring widgets in these windows. This research proposes modification of AUT from source code of the project as a solution to display all windows in it. The approach can be divided into three phase as the Fig. 2.

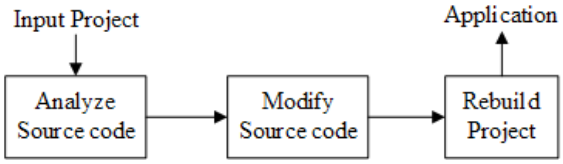

Figure 2. Solution to display all windows in AUT

In the first phase, all windows of the application are found by analyzing the project's source code. For example, if the

source code is written in C#, this mission will be done by fetching all classes, a class inherited from one of the two following classes will be a corresponding window:

- *System.Windows.Window*
- *System.Windows.Forms.Form*

After that, several statements which open windows are inserted into appropriate locations. There are some conditions need to be satisfied in the insertion process: all windows of AUT need to be opened, each window is called open one time, so if a window is displayed, it does not need to insert new scripts to open again. Algorithm 1 describes the proposed method to overcome these problems. The main idea of this method is to store the opened windows by using a temporary text file *temp_file.txt*. Each class is corresponding to a window, in its constructor function, a statement writing its name is inserted at the first, and another statement calling to *Open_All_Windows* function is added at the end. The content of the *Open_All_Windows* function contains scripts that open all windows of AUT except itself. However, before opening a window, it must be sure that it has not opened yet by checking the existence of its name in the temporary text file. Finally, the modified source code is rebuilt to create a new application.

Algorithm 1: Modify source code of project

```
   Input  : Project's source code
   Output : Modified application
1  begin
2      create empty list
3      foreach class in project do
4          if class inherits Window or Form
             then
5              add class to list
6      foreach class in list do
7          name = name of class
8          tempFile = "tem_file.txt"
9          insert statement
             "AppendText2File(tempFile,name);"
             into the first of constructor function
10         insert statement
             "Open_All_Windows();"
             into the end of constructor function
11 function Open_All_Windows(temp_file):
12     if temp_file not contains SecondWindow
         then
13         open SecondWindow
14     if temp_file not contains ThirdWindow
         then
15         open ThirdWindow
       /* so on with remains windows  */
```

4. Test scenarios specification and test scripts generation

From the widgets retrieved before, testers add a list of interactions for each one. Then, these actions are combined to generate a collection of test scenarios. After that, test scripts are created corresponding to each test scenario.

4.1. Test scenarios specification

The method for specifying test scenarios is very important, it is the basis for testers design desired scenarios. This specification method should ensure that it is cost-effective to create new test scenarios. This paper proposes an approach to generate new test scenarios by combining the interactions of each widget. By using this method, testers only need to define a set of desired actions, validations for each widget. Then, these interactions are combined to generate test scenarios. Hence, testers spend fewer efforts than manual recording in Record and Replay technique.

Figure 3 shows several test scenarios generated from combination. The first row contains a list of widgets, widgets' attributes or values in column *Object* of Table 2 (hereinafter referred to as "Object"). Each remaining row is corresponding to a test scenario which consists of a collection of interactions, each of them is expressed by a cell.

	A	B	C	D	E	F
1	Mainscreen	TbUsername	TbPassword	ButtonOK	TaskScreen	MessageErrorBox
2	Display	'admin'	'123456'	N/A	Display	N/A
3	Display	'admin'	'123456'	N/A	N/A	Incorrect username
4	Display	'admin'	N/A	N/A	N/A	Incorrect password
5	Display	'admin'	'123456'	N/A	N/A	Incorrect password
6	Display	'admin'	'123456'	Click	Display	N/A
7	Display	'admin'	'123456'	Click	Display	Incorrect password
8	Display	'admin'	'123456'	Click	Display	Incorrect username

Figure 3. Several generated test scenarios for login action

Table 2. Other Objects in addition to widgets

Object	Describe	Example
Keyboard	Contains some interactions with keyboard	Press Ctrl+C
Wait	Action waiting until some conditions are satisfied	Wait until MainWindow displays
Delay	Action delay for an amount of time	Wait for 3 seconds
Capture	Action capture screen	Capture the current screen

Table 3 shows several rules of the specification method. There are three parameters in order to determine the correct interactions (corresponding to a cell in Excel file): *Object* - a cell in the first row, *Expression* - the value of the current cell, *Color* - the color of this cell (i.e., Environment, Pre-condition, Procedure, Validation).

4.2. Test scripts generation

The Excel specification file of test scenarios is analyzed to generate test scripts. These scripts and files store widgets' information are packaged into a test project. With this test project, users can run many times later independently. Test scripts are essentially a collection of statements interact with widgets of each test scenario. The statement *elements.LoginWindow.LoginButton.Click*(); is an example of expression of click action to *LoginButton* element in *LoginWindow*. Declaring the parent element *LoginWindow* in this statement is necessary because of the existence of two or more elements with the same name but they locate in different windows. For example, another window named *SecondWindow* also contains a *LoginButton*, so if only using *LoginButton.Click*(), this action will be not clear when not know the target element (in *LoginWindow* or in *SecondWindow*). If the action belongs to the *Pre-condition* type, a validation script must be added at the end of the scripts in order to check whether actions are performed successfully.

5. Test execution

Executing test project is essentially the execution of a sequence of actions, validations with widgets. There are multiple types of widgets, each one has various interactions type. Basically, they can be divided into three groups as follows.

- Direct actions (hereinafter referred to as "Action"): e.g., click a button, input text into a text-box, set value for widget's width, etc.

Table 3. Several specification rules to define interactions

Object	Expression	Color	Meaning
MainWindow	Open	Environment	Open MainWindow
MainWindow	Exist	Environment/ Validation	Check if MainWindow exists
MainWindow	Not Exist	Environment/ Validation	Check if MainWindow does not exists
Button1	Click	Procedure	Click Button1
Button1	DoubleClick	Procedure	Double click Button1
Button1	RightClick	Procedure	Right click Button1
Button1	Click (x,y)	Procedure	Click Button1 at position (x,y)
Textbox1	'abc'	Procedure	Input 'abc' into Textbox1
Textbox1	'abc'	Validation	Validate value of Textbox1's text attribute with 'abc'
Textbox1	'abc'	Pre-condition	Input 'abc' into Textbox1, then validate value of Textbox1's text attribute with 'abc'
Textbox1. Text	Contain 'abc'	Validation	Validate value of Textbox1's text attribute contains 'abc'
Textbox1. Text	Not contain 'abc'	Validation	Validate value of Textbox1's text attribute does not contain 'abc'
MainWindow. Width	500	Procedure	Set MainWindow width is 500
MainWindow. Width	500	Validation	Validate value of Textbox1's width attribute with 500
Keyboard	{K_Control; C}	Procedure	Press Ctrl and C
MainWindow	Capture	Procedure	Capture MainWindow screenshot
Delay	2	Procedure	Delay 2 times default duration time

- Validation interactions (hereinafter referred to as "Validation"): e.g., check the existence of a widget, validate the value of text attribute of a widget with some texts, etc.

- Other interactions: e.g., capture the current screen, delay an amount of time, wait until a window is displayed, etc.

If an interaction belongs to the *Action* or *Validation* type, there will be two phases to execute it including target widget detection, interaction execution. The target widget is found from the root-widgets using its attributes' value achieved in the previous phase. After test project was finished running, a test report is exported. This test report needs to show the state of the execution (success or failure) of each test scenario. There are three states of the result after performing an interaction as follows.

- Success: this interaction was executed successfully.

- Failure: this interaction was executed completely, but it returned false. For example, validating the value of text attribute of a widget with some texts returns false.

- Error: this interaction was not executed

completely. It means that in the execution process, there was something interrupted this interaction, and it could not continue. For example, the target widget cannot be found.

Whenever the result of executing an interaction is different from *success* state, the hold current screen will be captured and written into the test report as an additional information.

6. Tool and Experiments

6.1. GTA tool and a case study

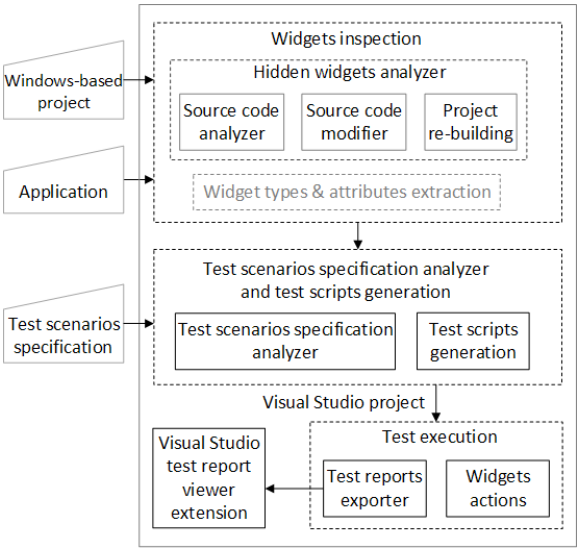

Figure 4. The architecture of GTA

Based on the proposed approach, a Windows Desktop tool named GTA has been developed. It contains four main modules as in Fig. 4. In widgets inspection module, the widgets and their properties are extracted by using Microsoft UI Automation library. The proposed combination method to generate test scenarios uses a free and lightweight tool named PictMaster. It is essentially an MS Excel file, contains some Macros to combine the values of each input parameter. Each *Object* (e.g., Button1, Delay, etc.) is considered as a parameter in PictMaster.

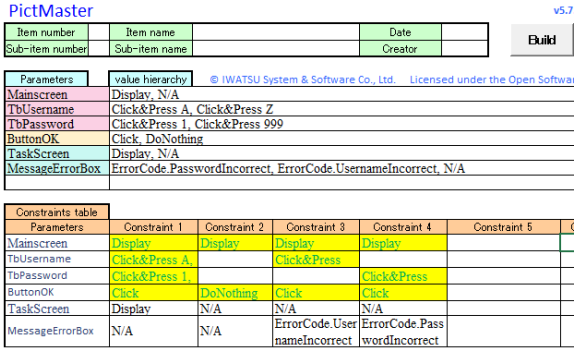

Figure 5. An example of specification using PictMaster

In order to show the effectiveness of GTA, it was tested with UI of several applications. In this section, a case study with ToDo application is illustrated. ToDo is a simple task management desktop application developed by Toshiba Software Development Vietnam (TSDV). From a set of widgets, GTA exports a template PictMaster file containing several pre-defined parameters (e.g., button, text box, etc.). Next, PictMaster generates a set of scenarios by combining the values of each parameter which were defined by testers before. Finally, GTA analyzes these scenarios to export a test project which can be opened, built, and run within Visual Studio. Figure 6 shows an example of test report which was generated whenever running test project completely. It can be seen that the scenario *Feature_1* is failed because the action checking the existence of *TaskManagetView* returns failure.

Table 4. Comparison in terms of the number of widgets extracted

Application	Widget type	GTA	Ranorex Spy	UI Spy
Calculator	Combo box	Fully	Miss not selected options	Miss not selected options
Testing Application	Tree view	Fully	Fully	Miss children nodes which parent are not expanded
	Tab	Fully	Miss widgets in not selected tab-pages	Miss widgets in not selected tab-pages
	Combo box	Fully	Miss all options	Fully
ToDo	Tab	Fully	Miss widgets' screenshots in not selected tab-pages	Miss widgets in not selected tab-pages
	Combo box	Fully	Miss all options	Fully

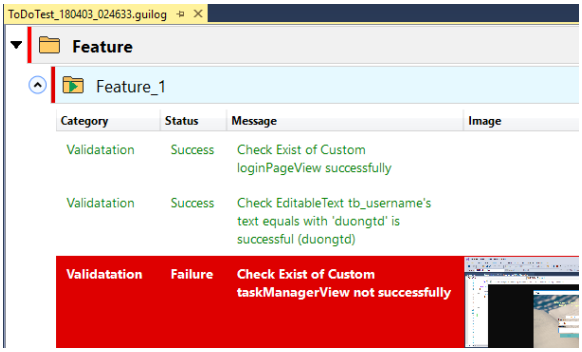

Figure 6. A test report of GTA

6.2. *Experiments*

This section makes a comparison in term of the number of widgets extracted between GTA, Ranorex Spy and Ui Spy. Ranorex Spy is a tool in commercial Ranorex tools and UI Spy is another provided by Microsoft. Three applications are used as the inputs including ToDo, TestingApplication (both were provided by TSDV) and Calculator (default on Windows 10). The details of comparison are shown in Table 4.

It can be seen that both Ranorex Spy and UI Spy ignore some widgets in the extraction process. Ranorex Spy always ignores some selections in a combo box widget with both three applications. Similar to the combo box, with a tab widget, Ranorex Spy also omits widgets in the unselected tab-pages. UI Spy detects fully selections of a combo box widget in two TSDV's application, but with Calculator application, it ignores un-selected selections. Besides, UI Spy cannot detect some nodes in a tree view whose parents have not expanded yet. In contrast, GTA overcomes these limitations because it applied several techniques to deal with hidden widgets. For example, whenever the extraction process encounters a combo-box widget, GTA will automatically execute clicking on the drop-down button on the right of it to visible all selections.

7. Conclusion

This paper presented a method of automated UI testing for Windows-based applications. The proposed method includes three phases including widgets inspection; test scenarios specification analyzer and test scripts generation; test execution. The key idea of the proposed method is to generate new test scenarios from the specification and widgets extracted from the application.

The proposed method can generate a large

number of test scenarios by combining a collection of interactions of each widget. Testers only need to define interactions of each widget, hence this method saves testers' efforts in creating UI interaction scenarios. The exported test project can be run multiple times automatically so it reduces the cost of test execution. Furthermore, some techniques are applied in order to extract hidden widgets which are ignored by other tools. A tool named GTA was implemented to demonstrate the effectiveness of the proposed method.

Currently, GTA has been used in TSDV and received positive feedback. Based on their assessments, GTA would be continually improving so as to apply more effectively on various kind of projects written different platforms in practice. Specifically, the next research would focus on solving the problem of hidden widgets detection in dynamic applications (e.g., widgets are loaded from the database). In addition, GTA will be extended the current idea of the proposal on other platforms such as Linux, and MacOS.

Acknowledgments

We thank Dr. Vo Dinh Hieu, VNU University of Engineering and Technologies for his reviews.

References

[1] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, J. Noujaim, Guicop: Specification-based gui testing, in: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, 2012, pp. 747–751.

[2] L. J. White, Regression testing of gui event interactions., in: icsm, Vol. 96, 1996, pp. 350–358.

[3] R. Narkhede, S. Korde, A. Darda, S. Sharma, An industrial research on gui testing techniques for windows based application using uft, in: Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM), 2015 International Conference on, IEEE, 2015, pp. 466–471.

[4] M. Cunha, A. C. Paiva, H. S. Ferreira, R. Abreu, Pettool: a pattern-based gui testing tool, in: Software Technology and Engineering (ICSTE), 2010 2nd International Conference on, Vol. 1, IEEE, 2010, pp. V1–202.

[5] S. Al-Zain, D. Eleyan, J. Garfield, Automated user interface testing for web applications and testcomplete, in: Proceedings of the CUBE International Information Technology Conference, ACM, 2012, pp. 350–354.

[6] A. M. Memon, M. E. Pollack, M. L. Soffa, Hierarchical gui test case generation using automated planning, IEEE transactions on software engineering 27 (2) (2001) 144–155.

[7] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, Reran: Timing-and touch-sensitive record and replay for android, in: Software Engineering (ICSE), 2013 35th International Conference on, IEEE, 2013, pp. 72–81.

[8] Z. Qin, Y. Tang, E. Novak, Q. Li, Mobiplay: A remote execution based record-and-replay tool for mobile applications, in: Proceedings of the 38th International Conference on Software Engineering, ACM, 2016, pp. 571–582.

[9] Y. Hu, I. Neamtiu, Valera: an effective and efficient record-and-replay tool for android, in: Proceedings of the International Conference on Mobile Software Engineering and Systems, ACM, 2016, pp. 285–286.

[10] S. Andrica, G. Candea, Warr: A tool for high-fidelity web application record and replay, in: Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on, IEEE, 2011, pp. 403–410.