AN OBJECT ORIENTED FRAMEWORK FOR USER INTERFACE TEST AUTOMATION

Izzat Alsmadi and Kenneth Magel
Department of Computer Science
North Dakota State University
Fargo, ND 58105
{ izzat.alsmadi, kenneth.magel}@ndsu.edu

ABSTRACT

Software testing is an important stage in the software projects lifecycles. It is one of the most expensive stages. Effective testing automation is expected to reduce the cost of testing. GUI is increasingly taking a larger portion of the overall program's size and its testing is taking a major rule in the whole project's validation. GUI test automation is a major challenge for test automation. Most of the current GUI test automation tools are partially automated and require the involvement of the users or testers in several stages.

This paper is about a proposed framework for user interface test automation that uses object oriented features to build the tested model. The GUI model is parsed from the application under test at run time. The GUI model is built as an XML tree that represents the GUI hierarchical structure. Test cases are then generated from the XML tree using different proposed algorithms. Some techniques for test case prioritization and critical path testing are suggested to minimize the number of required test cases to generate, that ensure an acceptable level of test adequacy or branch coverage.

The framework is concluded with test execution and verification part that execute the generated test cases and compare them with the original test suite. The advantages of the object oriented approach over the widely used capture/replay back one, is in the fact that the model is generated at run time which makes it represents the current state of the GUI model. In record/play back cases, we have to retest the application in case of any change in the functionalities or the GUI of the program. Once utilized, this object oriented approach is expected to be less expensive as it does not require users to manually test the application.

General Terms

GUI testing, test automation and GUI modeling.

Keywords

Test case generation, Test execution, GUI testing, GUI modeling, and verification.

1. Introduction

Graphical User Interfaces (GUIs) are taking larger portion of the overall project design, code and testing as newer programs are more interactive with the user inputs. Automated test refers to testing by a computer. We choose the option of automating the GUI testing when tests are repeated several times [13]. Software test automation tasks include selecting and generating the test cases, building the test oracle, executing, reporting and validating the results.

Graphical user interfaces manage controls. Controls are just reusable objects with which users can interact. We use "control" just as a generic term for any graphical object or widget that an application may produce. Controls have common elements that we need to consider before writing a program that interacts with a GUI [14].

- Each control belongs to a window class (making it possible to search them by class).
- Controls have an organizational hierarchy; every GUI has at least one root control, and every control may have child controls. Controls form a tree. This makes them searchable (by class or not) in depth: start from a root control and search among its siblings.
- Some controls have text attached to them. This text or caption can be used to locate the control from its class.
- Controls have different ways to be located or identified at run time. They can be identified by their parents or by the handle associated to them at run time.

The contribution of this work is in presenting a GUI testing framework that does not require the involvement of the tester throughout the different stages. The tests do not need to be revisited during any change in the GUI structure. Since the application uses reflection to get the actual model at run time, those changes are included in the current test. Generating test cases can happen from requirements, design or the actual GUI implementation. Although it is expected that those three should be consistent and related, yet they have different levels of abstraction. Requirements and design are usually of a high level of abstraction to generate from them the test cases. On the other hand the task of generating the test cases from the GUI implementation model is delayed until we implement the GUI, which is usually occurred in the late implementation. We should not have any problems in delaying GUI testing giving the fact that a tool automates the generation and executing process. We designed a tool in C# that uses reflection to serialize the GUI control components. Certain control properties are selected to be serialized. These properties are relevant to the user interface testing. The application then uses the XML file that is created to build the GUI tree or the event flow graph and generate the test cases. Generating the test cases takes into consideration the tree structure to select the test cases that cover unique branches. We will study the fault detection effectiveness of our test case selections.

The next section introduces the related work. Section 3 lists the goals of this research and describes the work done toward those goals. Section 4 presents the conclusion and future work. Section 5 presents a summary about the developed GUI auto tool.

2. Related work

There are several research papers about GUI testing using the data model, [2] [3] [4] [5] [7] [8] [10]. The overall goals and approach for this work is very similar to their goals. The GUI testing framework described, as a GUI test automation structure, is generic and should be applied to any testing or GUI testing model. The GUI ripper described in the above papers does not select certain criteria to rip or serialize. Serializing all GUI objects and their properties make the task cumbersome and error prone. GUI testing does not need all those properties as there are only certain properties that are relevant to the purpose of testing. The above approach requires the user involvement and decision in several places which does not make it fully automatic as suggested. The above research also follows a complex procedure in test case generation and do not consider any state reductions. Assuming that changing any property in any GUI object changes the GUI state is an assumption that generated a very large number of possible states for even small applications. We consider some state reduction techniques that should improve the effectiveness of our approach. We intended to follow the same GUI testing framework for our future work and expect the overall results to be more practical and easier to apply on actual projects.

The other related works to this project are in two categories. The first category is the work related to run time analysis of GUI or code [9]. In general dynamic or run time metrics or tools study the code while running and collecting data dynamically from the Application Under Test (AUT). Our work analyzes the GUI while running and extracts some specific information that can be used for test case generation.

The second category[1][6] is related to semi test automation using some capture/reply tools like WinRunner, QuickTest pro, Segui silk, QARun, Rational Robot, JFCUnit, Abbot and Pounder to creates unit tests for the AUT. Capture/reply tools have been existed and used for years. This may make them currently more reliable and practical as they have been tested and improved through several generations and improvements. There are several problems and issues in using record/play back tools [11]. The need to reapply all test cases when the GUI changes, the complexity in editing the scripts code, and the lack of error handlings are some examples of those issues. The reuse of test oracles is not very useful in the case of using a capture/replay tool [12]. We expect future software projects to be more GUI complicated that may make the test automation data model more practical. Many researches and improvements need to be done for the suggested data model to be more practical and usable.

3. Goals and approaches

One important aspect of GUI's that helps in state reduction is the fact that the GUI is hierarchical by nature. For example, in MS Notepad, to reach the event "Print" we have to pass through the event or menu item "File" and so on.

Our tool analyses the GUI and extracts its hierarchical tree of controls or objects using reflection. The decision to use XML file as a saving location for the GUI model, is in the fact that XML supports hierarchy. This hierarchy can be extracted easily from the XML file. We used the information about the parent of each control in the assembly to build the GUI tree. Figure1 is a screen shot from the XML output file extracted from a simple Notepad Application developed to be the AUT.

```
<Root>GUI-Forms</Root>
<Root>Form1</Root>
<Parent-Form>Form1</Parent-Form>
<Name>Form1</Name>
<Control-Level>0</Control-Level>
<TextBox>
<Parent-Form>Form1</Parent-Form>
<Name>textBox1</Name>
<Control-Level>1</Control-Level>
<ControlUnit>0</ControlUnit>
<LocationX>0</LocationX>
<LocationY>24</LocationY>
<Forecolor>Color [DarkBlue]</Forecolor>
<BackColor>Color [Linen]</BackColor>
<Enabled>True</Enabled>
<Visible>False</Visible>
</TextBox>
<MenuItem>
<Name>System.Windows.Forms.MenuItem, Items.Count: 4, Text: File</Name>
<Control-Level>1</Control-Level>
<ControlUnit>0</ControlUnit>
<Parent>System.Windows.Forms.MainMenu, Items.Count: 3</Parent>
<Text>File</Text>
<Visible>True</Visible>
<Enabled>True</Enabled>
<ShortCut>None</ShortCut>
</MenuItem>
```

Figure1: a screen shot from the XML file that is generated using GUI Auto tool. The AUT is a simplified Notepad application developed in C#.

The above information about the controls and their properties are extracted directly from the assembly. We added two more properties; control level and control unit for each control. Those properties are used along with the parent property to locate each control during test generation, selection and execution.

In the generated control graph, for branch coverage, each path of the GUI tree should be tested or listed at least once in the test cases. We define a test case as a case that starts from an entry level control and the select two or more controls from the lower levels. For example, "NotepadMain File Exit" is a test case or test scenario as it represents three controls that form one column in the tree. During test generation, algorithms have to take the hierarchical structure into consideration and select for example a control that is a child for the current control.

The user has the ability to select certain controls from the tree and gives them more than the typical equally distributed weight. This will have an effect on test case selection algorithms. For example, a control that is selected manually by the user will be selected whenever it is possible or valid to do so.

Rather that defining the GUI state change as a change in any control or control property (2), which produces a large amount of possible test cases, we define the GUI state as the state that is represented by the generated XML file from that GUI. If any of the parent-child relations in the GUI is changed, or any property of those parsed controls is changed, then that is considered a GUI state change.

Using finite state machines in GUI modeling have some problems. First due to the huge number of states we can get. Second is in the fact that the GUI state model is not exactly a state machine. In typical state machines, at any time, the system should be in a specific state (with certain triggers). In GUI however, there are many states that are inaccessible without being in another specific state that is higher in the hierarchy. This rule sometimes is broken through the use of shortcuts, but it is true for most cases. A typical state machine does not take into consideration the tree or hierarchical structure of the GUI by assuming that the GUI can be in any state at any time.

We used some abstraction removing those properties that nearly irrelevant to the GUI state to reduce the large number of possible states. The process of selecting those properties is manual where those only relevant properties are parsed through the application.

In the process of developing test generation techniques, we developed several test generation algorithms. All algorithms check for a valid selection of a tree edge. For example, using Notepad AUT, if the current control is "File", then a test algorithm may select randomly a valid next control from the children of the File control (e.g. Save, SaveAs, Open, Exit, Close, Print). In another algorithm, we processed the selected test scenarios to ensure that no test scenario will be selected twice in a test suite. Figure2 is a sample output generated from one of the test generation algorithms. In the algorithm, each test case starts from the root or the main entry "Notepad Main", and then selects two or more controls randomly from the tree. The algorithm verifies that the current test case is not existed in the already generated test cases.

```
1,NOTEPADMAIN,SAVE,SAVEFILEBUTTON1,,,
2,NOTEPADMAIN,EDIT,FIND,TABCONTROL1,TABFIND,
3,NOTEPADMAIN,VIEW,STATUS BAR,,,
4,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABTXTREPLACE,
5,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABBTNREPLACE,
6,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABLABEL2,
7,NOTEPADMAIN,EDIT,CUT,,,
8,NOTEPADMAIN,EDIT,FIND,TABCONTROL1,TABREPLACE,
9,NOTEPADMAIN,OPEN,OPENFILECOMBOBOX4,,,
```

Figure2: A sample from a file generated from a test generation algorithm using GUI Auto.

To calculate test generation efficiency we calculate the total number of arcs visited in the generated test cases to the total number of arcs or edges in the AUT. File-Save, Edit-Copy, Format-Font are examples of arcs or edges. We developed an algorithm to count the total number of edges in the AUT by using the parent info for each control. (This is a simple approach of calculating test efficiency, we are planning to try more rigorous efficiency measuring techniques in future work). Figure3 is a chart result from four of the proposed test generation algorithms. It shows the number of test cases generated and its test effectiveness.

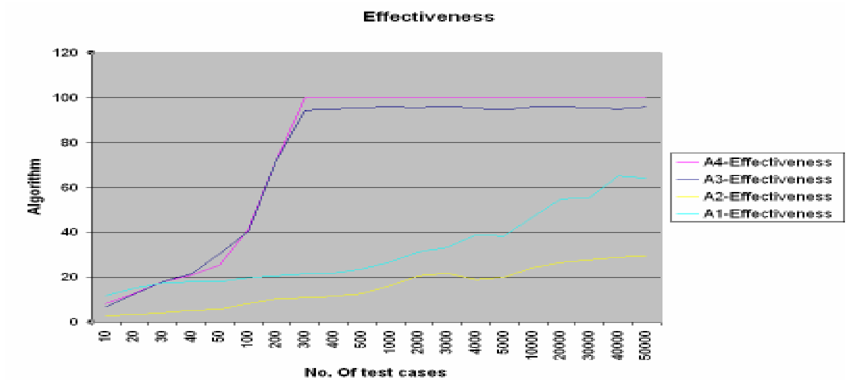

Figure3: Test case generation effectiveness for four algorithms used in GUI Auto.

In figure3, two of the selected algorithms reach 100 % efficiency or test adequacy using about 200 test cases.

One way to build a GUI test oracle is to study the effect of the events. The result of the test case is the combining results of all its individual events' effects. For example If we have a test case as File-Save Edit- copy – select test – paste, then the result of this test case has effects on two objects. File save has effect on a file object. We should study the file state change, and not the whole GUI state change. Then Edit-copy has an effect on the clipboard object, paste will have the effect on the main editor object or state, adding to it the copied text. Verifying the results of this test case is achieved by verifying the state change of the three objects affected; the file, the clipboard and the object editor. In some cases, we may ignore some of the intermediate events.

Each application should have a table or tables like Table 1 to be checked for test oracles.

Objects of the event	**The event**	**The object(s) affected**	**Description**
File,save	File-save	A file	The text from the object editor will be saved to the specified file
Edit,cut	Edit-cut	Clipboard, object editor	
Edit,copy	Edit-copy	Clipboard	
Edit, paste	Edit-paste	Object editor	

Table 1: GUI events objects interaction.

As an alternative that is easier to achieve, we developed a log file to track the events that are executed in the tool during the execution process. In a simple example, figure 4 shown below, we generated two test cases that write a text in Notepad and save it to a file. Those test cases are generated using the tool.

```
1,NOTEPADMAIN,FILE,NEW,TXTBODY,,
2,NOTEPADMAIN,FILE,SAVE AS,SAVEFILEBUTTON1,,,
```

Figure 4: a sample of generated test cases.

The first test case opens a new document and writes to it. As part of the default input values, we set for each control a default value to be inserted by the tool through execution. A textbox writes the word “test” or the number “0” whenever it is successfully called. A menu item is clicked, using its parent, whenever it is successfully called. For example, if Save is called as a control, File-Save as an event is triggered. We should have tables for valid and invalid inputs for each GUI control. The second test case opens the save file dialogue and clicks the OK or accept button (Savefilebutton1), to save the document. Here is the corresponding log file output for the above test cases.

```
Control  Event     Date Time
File  new  Menu Click  2/3/2007 11:51:23 AM
File  new  Mouse Down  2/3/2007 11:51:23 AM
File  new  Mouse Up  2/3/2007 11:51:23 AM
New  txtbody Menu Click  2/3/2007 11:51:23 AM
New  txtbody Mouse Down  2/3/2007 11:51:23 AM
New  txtbody Mouse Up  2/3/2007 11:51:23 AM
TxtBody  Mouse Move  2/3/2007 11:51:23 AM
TxtBody  Key Down  2/3/2007 11:51:23 AM
TxtBody  Key Up  2/3/2007 11:51:23 AM
(Test) is written in the document  2/3/2007 11:51:23 AM
(Test) is written in the document  2/3/2007 11:51:24 AM
SaveFilebutton1  Mouse Move  2/3/2007 11:51:24 AM
SaveFilebutton1  Mouse Button Down  2/3/2007 11:51:24 AM
```

```
SaveFilebutton1  Mouse Button Up  2/3/2007 11:51:24 AM
File  SAVE AS Menu Click  2/3/2007 11:51:24 AM
File  SAVE AS Mouse Down  2/3/2007 11:51:24 AM
File  SAVE AS Mouse Up  2/3/2007 11:51:24 AM
SaveFilebutton1  Mouse Move  2/3/2007 11:51:24 AM
SaveFilebutton1  Mouse Button Down  2/3/2007 11:51:24 AM
SaveFilebutton1  Mouse Button Up  2/3/2007 11:51:24 AM
```

Figure 5 Log file output of a sample test suite.

Since the test execution process is complicated and subjected to several environment factors, the verification process is divided into three levels.

1. In the first level the tool checks that every control in the test suite is successfully executed. This step is also divided into two parts. The first part is checking that all controls executed are existed in the test suite. This is to verify that the execution process itself does not cause any extra errors. The second part that ensures all controls in the test suites are executed tests the execution and its results. In our implementation of this level, some controls from the test scenarios were not executed. This is maybe the case of some dynamic execution issues where some controls are not available in certain cases.
2. In the second level the tool checks that the number of controls matches between the two suites.
3. In the third level the tool checks that the events are in the same sequence in both suites.

Table 2 summarizes the implementation of the verification process using the event log. All generated test suites in this example use the legal sequence algorithm described earlier. The generated test suite is referred to as one while the executed suite is referred to as two.

Test suite is 1 and execution suite is 2. The algorithm used is the legal sequence algorithm.						
No. of test cases	Test List controls count	Exec List controls count	All in 2 are in 1?	All in 1 are in 2?	Equal number?	Same sequence?
5	19	20	Pass	Pass	Fail	Fail
10	35	28	Fail. (Save is executed instead of SaveAs in test suite).	Fail. (About notepad link, could not be executed)	Fail	Fail
15	51	42	Pass	Fail. (About notepad link, could not be executed)	Fail	Fail
20	66	57	Pass	Fail (Status bar, could not be executed)	Fail	Fail
25	82	71	Fail. (SaveAs is executed instead of Save in test suite).	Fail (Printlabel1, could not be executed).	Fail	Fail
30	96	79	Pass	Fail (fontlabel4,	Fail	Fail

				could not be executed).		
35	112	93	Pass	Fail.(openfilelabel3 , could not be executed).	Fail	Fail
40	123	115	Pass	Fail (fontlistbox1, could not be executed).	Fail	Fail
45	145	140	Pass	Fail (fontlistbox1, could not be executed).	Fail	Fail
50	161	151	Pass	Fail (Savefilelabel5, could not be executed).	Fail	Fail

Table2: test execution and verification results.

Figure 6 shows the total percent of controls in the test suite that is executed. The percentage average varies between 80 to 100 %. In most of the results, there are some controls that could not be executed. Some of these controls are not available at run time. The dynamic issue in execution is an open issue that requires further research. Example of issues to deal with dynamically noticed here is how to execute disabled or invisible controls that are discovered by test generation algorithms but not the execution process. A suggested solution is to include preconditions for executing events on such controls.

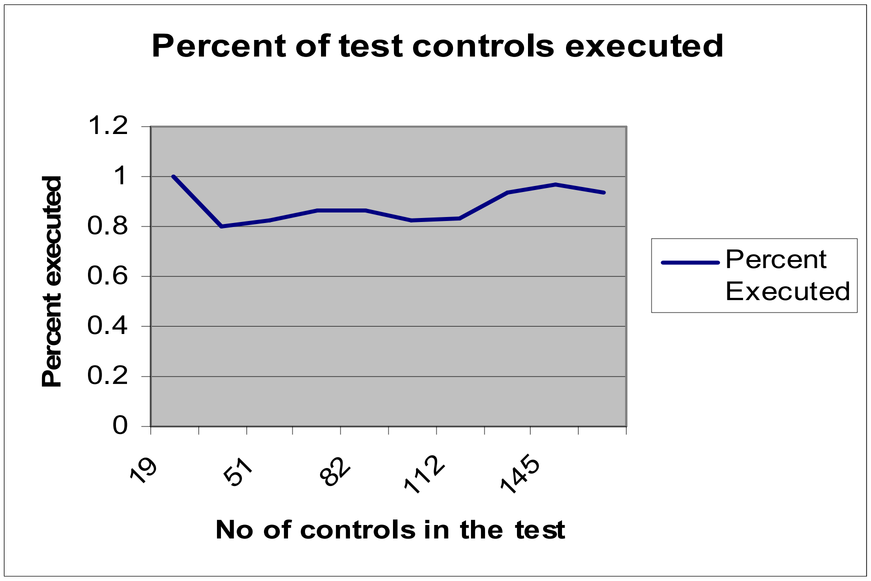

Figure 6: the percent of controls in the test suite that is executed.

4. Conclusion and future work

This paper is about an ongoing GUI test automation research project. There are some techniques that has been tested to make the process of GUI test automation more effective and practical. We believe that this project is in an early stage and there are many tasks yet to be done. We will continue refining our approach and extending test case generation techniques to include more effective ones. Test oracle techniques are explained in principles in this research. More elaborations and verifications needs to be accomplished to prove the effectiveness of the suggested track. We will also study the fault detection effectiveness of our test case selection techniques.

One proposed extension for this work is to use a formal model checker like LTSA to verify the GUI model against certain properties like safety and progress. Since our test cases are generated from an implementation model, we thought of using LTSA to verify the implementation model against the design model. Using LTSA, we will define some requirement properties to be checked for the correctness and safety of the model. The verification of the implementation model rather than the design model is expected to expose different issues. While the design model is closer to the requirement, it is more abstract and will generally cause some difficulties for testing. However, implementation model is closer to testing and is expected to be easier to test and expose more relevant errors. Those errors could be a reflection of a requirement, design, or implementation problems.

5. GUI Auto; The developed GUI test automation tool.

We are in the progress of developing GUI Auto as an implementation for the suggested framework. GUI Auto tool generates in the first stage an XML file from the assembly of the AUT. It captures the GUI controls and their relations with each other. It also captures selected properties for those controls that are relevant to the GUI. The generated XML file is then used to generate a tree model. Several test case generation algorithms are implemented to generate test cases automatically from the XML model. Test case selection and prioritization techniques are developed to improve code coverage. Test execution is triggered automatically to execute the output of any test case generation algorithm. An event logging module is developed to compare the output of the execution process with the test cases used. The generated files are in an XML or comma delimited formats that can be reused on different applications. A recent version of the tool can be found at http://www.cs.ndsu.nodak.edu/ ~alsmadi/GUI_Testing_Tool

6. References

1. L. White and H. Almezen. Generating test cases from GUI responsibilities using complete interaction sequences. In Proceedings of the International Symposium on Software Reliability Engineering, pages 110-121, Oct 2000.
2. A. M Memon. A Comprehensive Framework For Testing Graphical User Interfaces. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

3. Q. Xie. Developing Cost-Effective Model-Based Techniques for GUI Testing. In Proceedings of The International Conference of Software Engineering 2006 (ICSE'06). 2006.
4. A. M. Memon and Q. Xie . Studying the fault detection effectiveness of GUI test cases for rapidly evolving software. IEEE Transactions on Software Engineering, 31(10):884-896, 2005.
5. A. M. Memom, I Banerejee, and A. Nagarajan. GUI Ripping: Reverse Engineering Of Graphical User Interfaces For Testing. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), 1095-1350/03. 2003.
6. A. K. Ames and H Jie. Critical Paths for GUI Regression Testing. University of California, Santa Cruz. http://www.cse.ucsc.edu/~sasha/ proj/ gui_testing.pdf. 2004.
7. A. M. Memon. Developing Testing Techniques for Event-driven Pervasive Computing Applications. Department of Computer Science. University of Maryland.
8. A. M. Memon. GUI testing: Pitfall and Process. Software Technologies. August 2002. Pages 87-88.
9. A. Mitchell and J. Power. An approach to quantifying the run-time behavior of Java GUI applications.
10. A. M. Memon, and M. Soffa. Regression Testing of GUIs. In Proceedings of ESEC/FSE'03. Sep. 2003.
Saket Godase. An introduction to software automation. http://www.qthreads.com/articles/testing/an_introduction_to_software_test_automation.html. 2005.
12. Brian Marick. When should a test be automated. http://www.testing.com/writings/automate.pdf. (Presented at Quality Week *'98*.).
13. Yury Makedonov. Managers guide to GUI test automation. Software test and performance conference. 2005. http://www.softwaretestconsulting.com/ Presentations_slides/Manager_sGuide_GUI_TestAutomation_11wh.pdf.
14. George Nistorica. Automated GUI testing. http://www.perl.com/pub/a/2005/08/11/win32guitest.html. 2005.