GUI Path Oriented Test Generation Algorithms

Izzat Alsmadi
Department of computer science
North Dakota state university
izzat.alsmadi@ndsu.edu

ABSTRACT

Testing software manually is a labor intensive process. Efficient automated testing can significantly reduce the overall cost of software development and maintenance. Graphical User Interfaces (GUI's) code has some characteristics that distinguish it from the rest of the project code. Generating test cases from the GUI code requires different algorithms from those usually applied in test case generation. We developed several GUI test generation automated algorithms that do not need any user involvement and that ensure test adequacy in the generated test cases. The test cases are generated from an XML GUI model or tree that represents the GUI structure. This work contributed to the goal of developing fully GUI test automated framework.

General Terms
User interface, Automatic test case generation.
Keywords

Test Automation, GUI Testing, Test Case Generation.

1. INTRODUCTION

Testing tries to answer the following questions(3): Does the system do what it should do, or does its behavior comply with its functional specification (conformance testing), how fast can the system perform its tasks (performance testing), how does the system react if its environment does not behave as expected (robustness testing), and how long can we rely on the correct functioning of the system (reliability testing).

User interfaces have steadily grown more rich, more user interactive and more sophisticated over time. In many applications one of the major improvements that are suggested with the new releases is to improve the user interface.

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 will be 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 will automate the generation and executing process. We designed a tool in C# that uses reflection to serialize the GUI control components or widgets. Certain control properties are selected to be serialized. These properties are relevant to the user interface. The application then uses the XML file that is produced 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. Test cases are selected with the consideration of the effectiveness of the selected test suit. We will study the fault detection effectiveness of our test case selections.

The algorithms developed to generate test cases from the GUI are novels. The two factors that affect the suggested algorithms were first generating a valid test scenario in which each edge is a legal edge in the actual GUI model. The second factor is ensuring a certain level of effectiveness on the generated test scenarios.

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 introduces in summary the developed GUI Auto tool. Section 5 presents the conclusion and future work.

2. RELATED WORK

Software testing is about checking the correctness of the system and confirming that the implementation conforms to the specifications. Conformance testing checks whether a black box Implementation Under Test (IUT) behaves correctly with respect to its specification. The work in this paper is related to test case generation algorithms, automatic test case generation and GUI test case generation in software testing. Several approaches have been proposed for test case generation, mainly random, path-oriented, goal-oriented and intelligent approaches (5) and domain testing (which includes equivalence partitioning, boundary-value testing, and the category-partition method) (7). Path-oriented techniques generally use control flow information to identify a set of paths to be covered and generate the appropriate test cases for these paths. These techniques can further be classified as static and dynamic. Static techniques are often based on symbolic execution, whereas dynamic techniques obtain the necessary data by executing the program under test. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken. Intelligent techniques of automated test case generation rely on complex computations to identify test cases. The real challenge to test generation is in the generation of test cases that are capable of detecting faults in the IUT. We will list some of the works related to this paper. Goga(2) introduce an algorithm bases on probabilistic approach. It suggests combining the test generation and the test execution in one phase. Tretmans(3) studied test case generation algorithms for implementations that communicate via inputs and outputs, based on specifications using Labelled Transition Systems (LTS). In MulSaw project (4), the team use 2 complementary frameworks, TestEra and Korat for specification based test automation. To test a method, TestEra and Korat automatically generate all non-isomorphic test cases from the method's pre-condition and check its correctness using its post-condition as a test oracle. There are several papers related to this project. We have a similar approach that focus on GUI testing. As explained earlier, one of the goals of our automatic generation of test scenarios is to produce non-isomorphic test scenarios. We also check the results of the tests through comparing the output results with event tables generated from the specification. Those event tables are similar to the pre post condition event tables. Clay (6) presented an overview

for model based software testing using UML. Prior to test case generation, we develop an XML model tree that represents the actual GUI that is serialized from the implementation. Test cases are then generated from the XML model. Turner and Robson [8] have suggested a new technique for the validation of OOPS which emphasizes the interaction between the features and the object's state. Each feature is considered as a mapping from its starting or input states to its resultant or output states affected by any stimuli. Tse, Chan, and Chen (9) and (11) introduce normal forms for an axiom based test case selection strategy for Object oriented programs and equivalent sequences of operations as an integration approach for object oriented test case generation. Orso and Silva (10) introduce some of the challenges that Object Oriented technologies added to the process of software testing. Rajanna (12) studies the impact and usefulness of automated software testing tools during the maintenance phase of a software product by citing the pragmatic experiences gained from the maintenance of a critical, active, and very large commercial software product as a case study. It demonstrated that most of the error patterns reported during the maintenance are due to the inadequate test coverage, which is often the outcome of manual testing, by relating the error patterns and the capability of various test data generators at detecting them. Stanford paper (13) is an example of using formal methods in defining the specifications through object specification tool that check for some properties like correctness. It is hoped that the application produced by this project should form the groundwork for another tool that is capable of producing small adequate test-sets that can successfully verify that an implementation of the specification produced is correct.

In the specific area of GUI test case generation, Memon (14) has several papers about automatically generating test cases from the GUI using an AI planner, the process is not totally automatic and requires the user decision to set current and goal states. The AI planner will find the best way to reach the goal states giving the current state. Another issue with respect to this research is that it does not address the problem of the huge number of states that a GUI in even small application can have and hence may generate too many test cases. The idea of defining the GUI state as the collection state of each control and that the change of a single property in one control will lead to a new state is valid but is the reason for producing the huge amount of possible GUI states. We considered in our research another alternative definition of a GUI state. By generate an XML tree that represent the GUI structure, we can define the GUI state as embedded in this tree. This means that if the tree structure is changed, which is something that can be automatically checked, then we consider this as a GUI state change. Although we generate this tree dynamically at run time and then any change in the GUI will be reflected in the current tree, yet this definition can be helpful in certain cases where we want to trigger some events (like regression testing) if the GUI state is changed. Mikkolainen (15) discusses some issues related to GUI test automation challenges. Alexander (16) and Haward present the concept of critical path

testing for GUI test case generation. They define the critical paths as those paths that have “repeated” edges or event in many test cases. The approach utilizes earlier manually created test cases through a capture\play back tool. Although this is expected to be an effective way of defining critical paths, yet it is not automatically calculated. As an alternative to the need of defining critical paths from run time, we define in one algorithm static critical paths through the use of metric weights. The metric weight is calculated by counting all the children- or grand children for a control. Other ways of defining critical paths is by measuring delay time during execution, or by manually locating critical paths from specification. From the specification a critical path can be a path that is calling an external API, saving to or calling an external file.

3. GOALS AND APPROACHES

4.CONCLUSION AND FUTURE WORK

5. REFERENCES

1. Pettichord, Bret. Homebrew test automation. ThoughtWorks. Sep. 2004. www.io.com/~wazmo/ papers/ homebrew_test_automation_200409.pdf.

2. Goga, N. A probabilistic coverage for on-the-y test generation algorithms. Jan. 2003. fmt.cs.utwente.nl/publications/files/ 398_covprob.ps.gz.

3. Jan Tretmans: Test Generation with Inputs, Outputs, and Quiescence. TACAS 1996: 127-146.

4. Software Design Group. MIT. Computer Science and Artificial Intelligence Laboratory. 2006. http://sdg.csail.mit.edu/index.html.

5. Prasanna, M, S.N. Sivanandam R.Venkatesan and R.Sundarrajan. A survey on automatic test case generation. Academic Open Internet Journal. Vol. 15. 2005.

6. Williams, Clay. Software testing and the UML. ISSRE99. 99. http://www.chillarege.com /fastabstracts/issre99/.

7. Beizer, Boris. Software Testing Techniques. Second Edition. New York, Van Nostrand Reinhold, 1990.

8. Turner, C.D. and D.J. Robson. The State-based Testing of Object-Oriented Programs. Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM- 93), Montreal, Quebec, Canada, Sep. 1993.

9. T.H. Tse, F.T. Chan, H.Y. Chen. An Axiom-Based Test Case Selection Strategy for Object-Oriented Programs. University of Hong Kong, Hong Kong. 94.

10. Orso, Alessandro, and Sergio Silva. Open issues and research directions in Object Oriented testing. Italy. AQUIS98.

11. T.H. Tse, F.T. Chan, H.Y. Chen. In Black and White: An Integrated Approach to Object-Oriented Program Testing. University of Hong Kong, Hong Kong. 96.

12. Rajanna V. Automated Software Testing Tools and Their Impact on Software Maintenance- An Experience. Tata Consultancy Services.

13. Stanford, Matthew. Object specification tool using VTL. Master dissertation. University of Sheffield. 2002.

14. Memon, Atef. Hierarchical GUI Test Case Generation Using Automated Planning. IEEE transactions on software engineering. 2001. vol 27.

15. Mikkolainen, Markus. Automated Graphical User Interface Testing. 2006. www.cs.helsinki.fi/u/paakki/mikkolainen.pdf.

16. Alexander K, Ames and Haward Jie. Critical Paths for GUI Regression Testing. www.cse.ucsc.edu/~sasha/proj/**gui**_testing.pdf