Developing Cost-Effective Model-Based Techniques for GUI Testing

Qing Xie
Department of Computer Science
University of Maryland, College Park, MD 20742
qing@cs.umd.edu

ABSTRACT

Most of today's software users interact with the software through a graphical user interface (GUI). While GUIs have become ubiquitous, testing of GUIs has remained until recently, a neglected research area. Existing GUI testing techniques are extremely resource intensive primarily because GUIs have very large input spaces. This research proposes to advance the state-of-the-art in GUI testing by empirically studying GUI faults, interactions between GUI events, why certain event interactions lead to faults, and use the results of these studies to develop cost-effective model-based GUI testing techniques. The novel feature of this research will be a reduced model of the GUI's event-interaction space. The model will be derived automatically from the GUI; it will be used to automatically generate focused GUI test cases that are effective at detecting GUI faults. The model will be extended to develop new test oracles, new coverage criteria for GUIs, and new regression testing techniques. Moreover, this research will empirically evaluate the developed techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, verification, reliability, human factors.

1. INTRODUCTION

Graphical User Interfaces (GUIs) have become the most popular means used to interact with today's software. A software user performs events to interact with the GUI, manipulating GUI objects, such as clicking a button, selecting a menu item, and inputing text. The GUI uses the input events to interact with the underlying code via messages or method invocations. These events may cause changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects.

Recognizing the importance of GUIs, software developers are dedicating an increasingly large portion (as much as 45-60%) of software code to implementing GUIs [7]. Consequently, the correctness of the GUI is necessary to ensure the correctness of the overall software. One of the ways to ensure the correctness of the GUI is to do testing. Testing has received considerable attention in the last decade, however, testing of GUIs has remained until recently, a neglected research area [2], basically because GUIs have characteristics different from those of traditional software.

Testing the correctness of a GUI is difficult for a number of reasons. First of all, the space of possible interactions with a GUI is enormous, in that each sequence of GUI events can result in a different state, and each GUI event may need to be evaluated in all of these states [2]. The large number of possible states results in a large number of input permutations requiring extensive testing. A related problem is to determine the coverage of a set of test cases [2]. For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing, because what matters is not only how much of the code is tested, but in how many different possible states of the software each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution [2]. An incorrect GUI state can lead to an unexpected screen, making further execution of the test case useless since events in the test case may not match the corresponding GUI elements on the screen. Thus, the execution of the test case must be terminated as soon as an error is detected. Also, if verification checks are not inserted at each step, it may become difficult to identify the actual cause of the error. Finally, regression testing presents special challenges for GUIs, because the input-output mapping does not remain constant across successive versions of the software [2]. Regression testing is especially important for GUIs since GUI development typically uses a rapid prototyping model [6].

Several researchers have exploited the event-driven nature of GUIs to develop new GUI testing techniques [2, 9]. However, since the number of permutations of all possible GUI interactions (event sequences) with the user is enormous, and hence prohibitively expensive to test, these techniques test the GUI for a suitable sub-space of user interactions. For example, AI planning has been used to generate focused test cases for specific GUI tasks [2]. While these techniques have been shown to be useful for test case generation, they are resource intensive and the fault detection effectiveness of the generated test cases has never been studied. It remains unclear how GUI events interact and why certain interactions lead to GUI faults. This research proposes to study GUI faults, interactions between GUI events, and why certain interactions lead to GUI faults, and develop efficient model-based GUI testing techniques that provide the best combination of fault detection effectiveness and cost.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/2005.

The next section introduces the related work. Section 3 lists the goals of this research and describes the approaches proposed to achieve them. Section 4 presents the preliminary work done to establish proof of concept. Finally, Section 5 concludes and outlines the remaining work.

2. RELATED WORK

The most popular GUI testing approach is to use semi-automated tools to do limited testing [9]. Examples of some tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, and *Pounder* to create unit tests for GUIs. Other tools include capture/replay tools that "capture" a user session as a test case that can be later "replayed" automatically during regression testing [1]. These tools facilitate only the execution of test cases; creating test cases is very labor-intensive.

Several researchers have developed techniques to automate some aspects of GUI testing. Memon et al. [2] have developed an automatic GUI testing framework called PATHS that uses AI planning to generate test cases. PATHS uses a description of the GUI (event-flow graphs) to automatically generate test cases and test oracles (preconditions and postconditions) from tasks (pairs of initial and goal states) by iteratively invoking the planner. In this approach, a GUI is modeled as a set of *widgets* $W = \{w_1, w_2, \ldots, w_m\}$, a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those widgets and values of the properties. At any specific point in time, the state of the GUI is represented in terms of all the widgets that it contains, and the values of all their properties. The event interactions in a GUI is represented as an event-flow graph (EFG). An EFG contains nodes that represent events and edges. An edge from node n_1 to n_2 means that the event represented by n_2 can be performed immediately after the event represented by node n_1. While this approach is successful in automating test case generation, it has several limitations: (1) there is no evidence showing that PATHS generates test cases that are effective at detecting faults; (2) the test case generator is largely driven by the choice of tasks given to the planner; a poorly chosen set of tasks will yield an inadequate test suite; (3) the test oracle compares the expected and actual output once after each event, making test execution very slow; (4) the planner uses an "operators" file, which is resource intensive to create; (5) the coverage criteria used by PATHS require a prohibitively large test suite; and (6) regression testing is performed by repairing test cases that have become unusable for the modified GUI; the fault detection effectiveness of the repaired test cases has not been demonstrated.

White et al. [9] model a GUI in terms of "responsibilities" (user tasks) and their corresponding "complete interaction sequences" (CIS). CIS is a sequences of GUI objects and selections that may be used to complete a responsibility. For each CIS, a reduced finite-state machine (FSM) model is constructed; this FSM is "traversed" to generate test cases [9]. The CIS concept is extended to develop a selective regression algorithm based on identifying changed and affected GUI objects and CISs. This technique requires a substantial amount of manual work on part of the test designer, who has to manually identify the responsibilities and the associated CISs. Moreover, there are no studies demonstrating the fault detection effectiveness of the generated test cases.

Other researchers have developed techniques to address isolated problems of GUI testing. For example, a variable finite state machine based approach has been proposed to generate test cases by Shehady et al. [8]. In summary, all of these techniques suffer from more or less similar problems. They are not systematic, they depend largely on how the test designers realize the models. They are all resource intensive, they address only one specific aspect of GUI testing, and no one has demonstrated the fault detection effectiveness of the test cases generated by these techniques.

3. GOALS AND APPROACHES

The primary goal of this research is to develop efficient model-based GUI testing techniques that provide the best combination of fault detection effectiveness and cost.

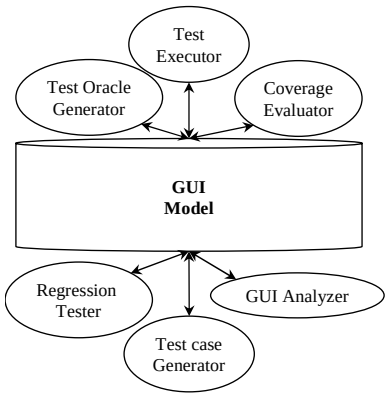

Figure 1: Overview of the Framework.

The output of this research will be a framework consisting of a model of the GUI and modules that employ this model to perform various GUI testing tasks. Figure 1 shows the primary modules of the framework. The central part of the framework will be the GUI model that will be used to represent GUI events, event interactions, and all the artifacts needed for test case generation, test oracle creation, coverage evaluation, and regression testing. The GUI model will be obtained using a *GUI analyzer* that will automatically reverse engineer the GUI and extract all the widgets, properties and their values. Given the GUI model as input, a *test case generator* will automatically generate focused test cases. The generated test cases will be provided to a *test oracle generator*, which will derive the expected output of the GUI for each test case. This expected output will be compared with the actual output obtained during execution of the test cases by the *test executor*. The *coverage evaluator* will evaluate the adequacy of the executed test cases. The *regression tester* will use the results from prior testing sessions to guide regression testing. New techniques that target specific error-prone parts of GUIs will be developed for each module.

The proposed framework will have the following characteristics:

- It will be *automated* so that the tester's work is simplified. The GUI model will be obtained automatically. Each module will use the model for automated testing.
- It will be *efficient* so that practitioners can use the framework even in the presence of tight deadlines.
- It will be *extensible* so that new techniques can be implemented and packaged as new modules of the framework.

- It will be *general* enough to be applicable to a wide range of GUIs.

Several approaches will be required to achieve the above goals. The details of these approaches are described next.

1. Observe real GUI faults: Many developers of open-source GUI-based software maintain detailed bug reports on the web (e.g., SourceForge). It is expected that several of these bugs will be relevant to the software's GUI. The bug reports for some popular GUI-based applications will be studied.

The bugs will be classified as either GUI faults or non-GUI faults. GUI faults include those that (1) are manifested on the visible GUI at some point of time during the software's execution and can be observed visually and (2) lead to a software crash following an input given via the GUI.

The event interactions that lead to the GUI faults will also be studied and used later to develop new test-case generation techniques. The faults themselves will be used as a basis for fault seeding to conduct experiments.

2. Develop an experimentation platform: An experimentation platform will be developed with the goal of using it to conduct experiments on GUIs and studying existing GUI models and techniques. The idea is to use the strengths of existing models and approaches, and evolve them for improved performance and fault detection effectiveness.

As a starting point, the following models and techniques will be implemented to develop the platform:

- The GUI representation in terms of events, event interactions, state of GUI, and event-flow graphs, as described in Section 2 will be used as the base model.
- GUI Ripper [3], a reverse engineering tool that can extract all widgets in all windows of a GUI automatically, will be used as the GUI analyzer. An editing tool will be provided to help test designers to verify the completeness and correctness of the model.
- Event-flow graph based approach will be used to generate test cases to cover the space of GUI interactions.
- Existing test oracles that obtain the GUI state after the execution of each event and the last event of the test case will be implemented for the platform.

3. Conduct experiments using the experimentation platform: Once the platform is implemented, large comprehensive experiments will be conducted to analyze the impact of test case length, event composition, test suite size and test oracle on fault detection effectiveness.

Existing in-house GUI software will be used as subject applications for the experiments. They are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[1] and includes TerpWord (a small word-processor), TerpSpreadSheet (a spreadsheet application), TerpPaint (an image editing/manipulation program), and TerpCalc (a scientific calculator with graphing capability). They have been implemented using Java. Faulty versions of these applications will be created by using the GUI faults observed in the bug reports (discussed earlier) as "templates" and creating instances of these faults.

Test cases will be generated and executed on both the original and faulty versions of the subject applications. A fault is detected if the test case running on the faulty version leads to a crash, or there is a mismatch between the original software's GUI and the faulty version's GUI. The characteristics of the test cases that are successful at detecting faults will be studied. The effect of test suite size, test case length, events composition, and test oracle on fault detection effectiveness will be analyzed. The testing cost in terms of space, time, and manual effort will be measured.

4. Evolve the framework with new models and techniques: Lessons learned from the experiments will guide this research to enhance the base model. The model will represent important event interactions in the GUI. These interactions will target specific error-prone events of the GUI. Specialized test case generation techniques that make use of the enhanced model will be developed. The model will continue to evolve as new test oracles, coverage criteria and regression testing techniques are developed and evaluated.

5. Evaluate new techniques: The evolved model and new techniques will be evaluated on existing subject applications and some open-source applications. A model for comparing cost and fault detection effectiveness will be constructed. Specifically, the evaluation process will consist of the following steps:

- Choose software subjects with GUI front-ends. As discussed earlier, in-house subject applications will be used first followed by some open-source applications.
- Generate test cases using new test case generation techniques and new models.
- Create oracles using new oracle creation algorithms.
- Seed faults in the subject applications to create faulty versions.
- Execute all test cases on the faulty versions of the subject applications.
- Measure (1) the number of faults detected, (2) costs in terms of time and storage space, and (3) other dependent variables needed for certain techniques, e.g., code and event coverage for the new test case generation techniques.
- Compare fault detection effectiveness (number of faults detected) and cost of the new techniques to the existing techniques using the above model.

Note that some of the above approaches are iterative as results of experiments drive the evolution of the GUI model and techniques.

4. PRELIMINARY RESULTS

The preliminary work includes the development of a new experimentation platform (consisting of a test case generator, test oracle generator, test executor, coverage evaluator and regression tester), a reduced GUI model [5], new test case generation techniques for evolving GUI software [5, 10], and new test oracles for GUIs [4]. The fault detection effectiveness of the new test cases and test oracles has been empirically demonstrated. Although additional research needs to be performed, the results presented in this section serve to illustrate that the new GUI testing techniques are efficient in terms of cost and fault detection effectiveness.

4.1 Event-Interaction Graph

Since an EFG (discussed in Section 2) models all possible (an extremely large number of) event interactions, it cannot be used directly for rapid testing; abstractions are used

[1] www.cs.umd.edu/users/atif/TerpOffice

to model only specialized (and hence a smaller number of) event sequences. In a typical GUI, 20-25% of the events are used to manipulate the structure of the GUI, such as menu-open and window-open events. The code for such events is usually generated automatically by visual GUI-building tools. This code is very unlikely to interact with code for other events; hence very few errors are revealed by executing interactions between these events. The remaining events in the GUI are *non-structural events* that do not cause structural changes to the GUI; rather they are used to perform some action; common examples include the `Copy` event used for copying objects to the clipboard.

Events that interact with the underlying software include non-structural events and those that close windows. These events are called system-interaction events. Intuitively, the GUI test cases are composed only of these events (and any other events necessary to "reach" the system-interaction events). Once these events have been identified in a GUI, the EFGs are transformed to event-interaction graphs. For lack of space, I will not provide the detailed mapping algorithm, the interested readers are referred to [5].

4.2 Test Case Generation for Frequent Regression Testing

Since GUIs are typically designed using agile processes, frequent re-testing of the changed GUIs are necessary, which involves analyzing the changes to the layout of GUI objects, selecting test cases that should be rerun, as well as generating new test cases.

The key idea of this technique [5, 10] is that for frequent regression testing, all nodes and edges in the EIG be executed on the GUI to test the basic functionality of the GUI. The regression testing process is designed to have as less manual intervention as possible.

The results showed that (1) the entire smoke testing process is feasible in terms of execution time, storage space, and manual effort, (2) the short test cases that cover all nodes and edges in the EIG are effective at detecting a large number of faults, (3) short test cases execute a large percentage of code, though certain parts of the application code can not be covered, and (4) test suite that covers all nodes and edges in EIG detect more faults than that in EFG.

4.3 New Test Oracle

It has been showed that test oracle contributes significantly to test effectiveness and cost of test cases by Memon et al. [3]. However, the frequency of comparison is ignored except at two extreme points during test case execution: (1) O_{all} – "check for `equality` of the expected output and actual output after each event" and (2) O_{last} – "check for `equality` of expected output and actual output after the last event" of the test case.

During the empirical studies, it is found that there are a large number of transient errors that "disappear" during test case execution and a large number of persistent errors that "persist" until after the last event of the test case. Since we can always detect persistent errors by using O_{last} transient errors need more study. In order to detect transient errors, O_{all} is required. Not only is the comparison process expensive, it also requires that the oracle information after each event needs to be maintained; for long test cases, this is significantly large. O_{last}, although cheaper, would miss such errors. What is needed is a test oracle that is comparable to O_{all} in terms of fault detection effectiveness and O_{last} in terms of cost.

By studying after which event the GUI error occurs (a GUI error is a mismatch between the expected output and actual output), it is found that most of the mismatches occur after the events that open a window or terminate a window. So the new oracle is designed to compare the expected and actual output after those events and after the last event in the test case.

The results showed that the new oracle is effective at detecting errors and efficient in resource utilization [4].

5. CONCLUSIONS AND REMAINING WORK

This research proposes to overcome the limitations of existing techniques by developing new GUI testing techniques. Proof of concept studies done to date on in-house software subjects have demonstrated that the proposed research is feasible and has the potential to advance the state-of-the-art in GUI testing.

The remainder of this research will focus on formalizing the preliminary work and extending the subject application pool. The formalization will include (1) the state in which an event executes will be studied to check the impact on its fault detection behavior; (2) new experiments will be conducted on open-source GUI programs to evaluate the fault detection effectiveness of the new testing techniques; and (3) the trade-offs between test suite size, test case length, events composition, test oracle on the fault detection effectiveness of GUI test cases will be studied.

6. REFERENCES

[1] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of *SPECIAL SESSIONS: Demonstrations*, page 1042, 1993.

[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] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 164–173, 2003.

[4] A. M. Memon and Q. Xie. Using transient/persistent errors to develop automated test oracles for event-driven software. In *Proceedings of The International Conference on Automated Software Engineering 2004 (ASE'04)*, pages 186–195, 2004.

[5] 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.

[6] R. M. Mulligan, M. W. Altom, and D. K. Simkin. User interface design in the trenches: Some tips on shooting from the hip. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, Practical Design Methods, pages 232–236, 1991.

[7] B. A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995.

[8] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, June 1997.

[9] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 110–121, Oct. 2000.

[10] Q. Xie and A. M. Memon. Rapid crash testing for continuously evolving GUI-based software applications. In *Proceedings of The International Conference on Software Maintenance 2005 (ICSM'05)*, pages 473–482, 2005.