Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces

Xun Yuan
Dept. of Computer Science
Univ. of Maryland
College Park, MD
xyuan@cs.umd.edu

Myra B. Cohen
Dept. of Computer Science & Engineering
Univ. of Nebraska-Lincoln
Lincoln, NE
myra@cse.unl.edu

Atif M. Memon
Dept. of Computer Science & UMIACS
Univ. of Maryland
College Park, MD
atif@cs.umd.edu

Abstract

Graphical user interfaces (GUIs) present an enormous number of potential event sequences to users. During testing it is necessary to cover this space, however the complexity of modern GUIs has made this an increasingly difficult task. Our past work has demonstrated that it is important to incorporate "context" into GUI test cases, in terms of event combinations, event sequence length, and by considering all possible starting and ending positions for each event. Despite the use of our most refined modeling techniques, many of the generated test cases remain unexecutable. In this paper, we posit that due to the dynamic state-based nature of GUIs, it is important to incorporate feedback from the execution of tests into test case generation algorithms. We propose the use of an evolutionary algorithm to generate test suites with fewer unexecutable test cases and higher event interaction coverage.

1. Introduction

Graphical user interfaces (GUIs) comprise the main method of interacting with programs today [6]. GUIs respond to events (such as a mouse click or a menu selection) to drive the program flow of control. The sequences of events triggered, the order in which these are selected and the length of each sequence is typically unspecified; these programs run indefinitely until a closing event occurs. As they run, the underlying states of the program change based on *context*, i.e., the sequence of all preceding events determines the current program state. It is this ability—to interact in an almost infinite number of ways with the external environment—that makes the event driven programming paradigm powerful. Yet, it is exactly the same ability that makes GUIs notoriously difficult to validate or test. Embedded in the program functionality are a combinatorially large number of potential permutations of sequences of events, or system states. Even when bounded by sequence length, there are too many permutations to effectively validate, but combinations of *event interactions* are likely to trigger faults [10]. Given the ubiquity of GUIs, the ever increasing complexity of these programs and event interactions, and the cost of field-based failures, a new generation of testing techniques is needed to provide effective and efficient validation. In this paper we focus on one important aspect of validation for GUIs – **automated test generation**.

Current test generation techniques for GUI systems include those based on state machine models and event graphs. However, these techniques will not continue to scale [10]. Some of the limitations of the current methods include: (1) an inability to decouple sampling criteria from event sequence length which in turn dictates the test suite sizes – sizes that grow exponentially; (2) event interaction models that cannot predict *infeasible sequences of events* (test cases which cannot be run) a priori; many sequences may fail to execute when run causing inadequate testing; and (3) test generation that is based on a *static* view of the requirements or system under test. This does not account for learned information gathered during testing.

In this paper, we posit that dynamic sampling algorithms are needed to generate test cases. Test cases need to be generated in batches. The current batch, once executed, will inform and refine the next batch, helping to avoid unexecutable test cases. We believe that we can address each of the above limitations through a new test generation framework for GUIs. The framework will sample event interactions and *dynamically adapt and evolve to automatically generate*

tests based on feedback obtained from *prior* test execution results[1]. Our proposed framework (1) uses sampling techniques derived from combinatorial interaction testing to generate tests that are not tightly coupled with sequence length, and that adapt over time; (2) uses feedback to uncover constraints between events resulting in fewer infeasible test cases, and higher test adequacy; and (3) incorporates these elements into an evolutionary algorithm to dynamically generate test cases that can potentially uncover more faults and that have longer feasible sequences.

The next section gives a brief discussion of GUI testing, and an overview of combinatorial interaction testing. Section 3 provides an overview of our proposed iterative feedback-based refinement of the combinatorial test model. Section 4 concludes with a discussion of future work.

2. Background and Related Work

Many **GUI testing techniques** have been proposed; some have been implemented as tools and adopted by practitioners (see [6] for more details). All of these techniques automate some aspect(s) of GUI testing including model creation (for model-based testing), test-case generation, test oracle creation, test execution, and regression testing. All of these explore the GUI's state space via sequences of GUI events.

Semi-automated unit testing tools such as *JFCUnit*, *Abbot*, *Pounder* and *Jemmy Module* are used to manually create unit GUI test cases, which are then automatically executed. More advanced tools called capture/replay tools "capture" a user session as a test case, which can later be "replayed" automatically on the GUI. Model-based techniques have been used to automate certain aspects of GUI testing. For example, manually created *state machine* models have been used to generate test cases. Our own work on GUI testing has focused on *graph* models to minimize manual work. Our most successful graph models that have been used for GUI test-case generation include Event Flow Graphs, (EFGs), and Event Interaction Graphs, (EIGs). The nodes in these graphs represent GUI events; the edges represent different types of relationships between pairs of events. An important property of a GUI's EIG is that it can be constructed semi-automatically using a reverse engineering technique called *GUI Ripping*.

The basis for **combinatorial interaction testing** is a *covering array*. A covering array (written as

[1]By "*prior*" we do not mean the previous version's test cases; rather, test cases will be generated *in batches* for the current version; execution results from batch 1 through batch $i-1$ will be used to generate additional (improved) test cases for batch i.

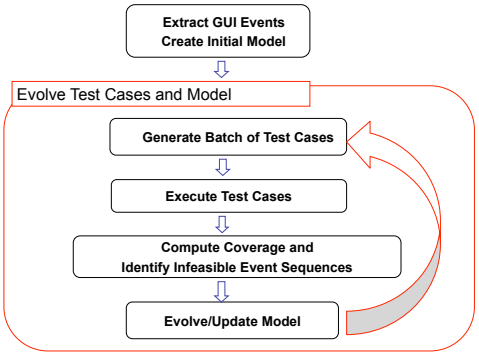

Figure 1. Automated Test Framework

$CA(N; t, k, v)$) is an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols *at least* once. In other words, any subset of t-columns of this array will contain all t-combinations of the symbols. We use this definition of a covering array to define GUI event sequence samples.

The *strength* of our sample is determined by t. For instance we set $t = 2$ to include all pairs of events between all event locations in the sequence. In this way the covering array provides not only 2-way coverage but it does this for each pair of locations in the sequence, providing more context for testing.

The number of test cases required for the t-way property, is N. Since the primary cost of running a test case is setup cost, we cover many more event sequences than the shorter 2-way sequences. In general we cannot guarantee that the size of N will the same as a shorter sequence, but it will grow logarithmically in k rather than exponentially as does the number of all possible sequences of length k [1].

Covering arrays have been used extensively to test input parameters of programs [1] and to test system configurations [8]. A special type of a covering array, (an orthogonal array) developed from Latin squares, has been previously used to define GUI tests by White [7], however this work used covering arrays in a stateless manner, to define subsets of the input parameter combinations. In [9] we used covering arrays to sample long event sequences, where events must consider state (determined by their location in the sequence).

3. Iterative Refinement Through Evolutionary Test Generation

There have been some recent approaches for developing automated test generation techniques that incorporate feedback during the test process [4, 5]. Most of these methods use some sort of structural measure such as code coverage for feedback, although a few have

used programmer supplied predicates or program behavior, both of which use the program state to provide feedback. One common way to automate test generation/feedback is through the use of evolutionary algorithms such as genetic algorithms [3, 4, 5]. An emerging field of software engineering, called search based software engineering utilizes meta-heuristic search to solve common software engineering problems [2].

Meta-heuristic search techniques can be applied to problems which can be formulated as an optimization problem that cannot be solved through exhaustive methods. The problem can be specified as a set Σ of feasible solutions (or states) together with a cost $c(S)$ associated with each $S \in \Sigma$. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum (or maximum) cost.

Genetic algorithms are from the class of meta-heuristic algorithms called evolutionary algorithms. These model the biological evolutionary process. A population is composed of many individuals from the set of feasible solutions. Pairs of solutions (*parents*) are selected based on their fitness. A crossover and recombination stage take place during which parents exchange and combine information to generate a set of children. A mutation is applied to the new population to diversify individuals and the fittest solutions from this new population are then selected and the process is repeated. The common driving elements for any type of meta-heuristic search is to formulate a fitness function that can be evaluated for each feasible solution, and to have a way to transform (or crossover) the individual or population.

3.1. The Proposed Algorithm

In [10] we proposed a feedback mechanism for generating new event sequences for testing using a state-based approach, but this was not automated. In our existing event driven test generation algorithm on GUIs we have found that it is feasible to exhaustively test all 2-way (length 2) sequences with limited resources. We call these *smoke tests* [10]. This is used as a seed to generate Event Semantic Interaction (ESI) relationships. To encode these relationships each possible event in the system is encoded with a unique event ID. From the ESI relationships we develop an ESI graph (ESIG). The ESIG contains nodes representing events and directed edges to show a relationship from one event to another. Given nodes n_1 and n_2, an edge from n_1 to n_2 means that there is an ESI relationship from the event represented by n_1 to the event represented by n_2. Figure 2 is an example of an ESIG graph for a small example program. Six of the 10 events are found to interact.

We used this method on four GUI-based open source programs and found that 25% of the events dominate the ESI relationships, indicating this is where we should focus our testing. When we used the ESIG to generate longer sequences (3,4,5-way interactions) based on combinatorial testing, new faults were uncovered, showing that the ESI approach has promise [9]. But we also ran into many infeasible test sequences.

This makes us believe that sampling by the use of combinatorial testing as well as modeling using Event Semantic Interaction information will improve context and provide stronger fault detection. However, we need to address the issue of infeasible test sequences. During our experimentation we have uncovered patterns of infeasible sequences that we were unable to detect statically. This makes us believe that an evolutionary algorithm which generates ESI based test sequences, and then automatically adapts after a batch of test sequences is run will increase the interaction coverage and avoid infeasible test sequences.

Our genetic algorithm for GUI test generation, works as follows (see Figure 1). We begin by (1) generating an initial model of the GUI event interactions and an initial set of test sequences based on this model. Next we (2) generate and (3) run a batch of test cases and (4) determine code coverage and identify infeasible sequences. This will be done by examining patterns of failures and relative locations where events fail. Once we have this information, we will remove infeasible test sequence combinations from the model and (5) update our model to reflect the new ESI. Finally we will calculate our ESI coverage and evolve the population with the aim of covering as many uncovered ESI relationships previously uncovered as possible. We will use the ESIs to calculate fitness and select the next generation of tests.

The fitness function may be calculated several ways. The first method is based on the paths in the ESIG model. Longer paths will mean a higher (fitter) individual since these represent longer feasible sequences. Alternative fitness may be based on covering new event sequences (not seen before) or on a priority based basis. If we have information after step (4) indicating that particular ESI sequences are more fault prone we may use this to increase our interaction coverage (and model) in this part of the graph. In essence we can use this information to prioritize our test sequences or to increase our strength of ESI coverage.

Figure 2 shows an example of our initial fitness based on test sequence length of the ESIG. Shaded cells indicate they are part of an ESIG graph and will impact our coverage. We measure fitness as the length of each of the longest sequences (sub-sequences are subsumed).

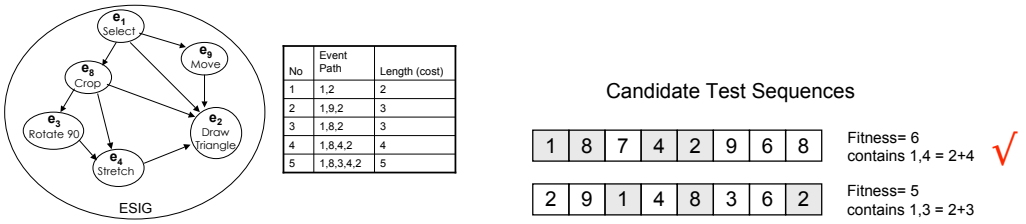

Figure 2. ESIG and Potential Test Sequences After Executing Some Test Cases

In this example the first test sequence is selected because it is fitter. We have been working to implement this algorithm to test empirically as a first step towards realizing our adaptive framework.

4. Conclusions and Future Work

Our earlier work based on a static model of the GUI used event semantic information and a combinatorial based test case generation approach for generating GUI test sequences. However, the percentage of unexecutable parts of test cases suggest that this approach needs to be augmented. In this position paper, we motivated the need for a dynamic approach to covering-array-based test-case generation.

In order to realize this framework we believe that several avenues of research in GUI testing are important to pursue. First sampling techniques that aim to incorporate context, such as those based on combinatorial interaction testing must be explored and understood further. We need coverage and adequacy criteria to understand how these impact fault detection. Second we need to further refine our ESI models to capture and extract those events that are likely to interact. Third, automated methods to extract failing patterns are needed, and finally, heuristics to define effective fitness functions for evolutionary algorithms should be developed. We believe that advances in each of these areas will bring us closer to our vision of an automated adaptive test generation framework for GUI testing.

Acknowledgments

This work is supported in part by the National Science Foundation through awards CCF-0747009 and CCF-0447864, the Office of Naval Research grant N00014-05-1-0421 and the Air Force Office of Scientific Research through award FA9550-09-1-0129. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF, the Navy or AFOSR.

References

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Trans. on Soft. Eng.*, 23(7):437–444, 1997.

[2] M. Harman. The current state and future of search based software engineering. In *Future of Soft. Eng.*, pages 342–357, 2007.

[3] B. Korel. Automated software test data generation. *IEEE Trans. on Soft. Eng.*, 16(8):870–879, 1990.

[4] P. McMinn. Search-based software test data generation: A survey. *Soft. Test., Verif. and Rel.*, 14(2):105–156, 2004.

[5] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Soft. Test., Verif. and Rel.*, 9(3):263–282, 1999.

[6] J. Strecker and A. M. Memon. Testing graphical user interfaces. In *Enc. of Info. Sci. and Tech., Second ed.* IGI Global, 2009.

[7] L. J. White. Regression testing of GUI event interactions. In *Intl. Conf. on Soft. Maint.*, pages 350 – 358, 1996.

[8] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. on Soft. Eng.*, 31(1):20–34, 2006.

[9] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *Intl. Conf. on Auto. Soft. Eng.*, pages 405–408, 2007.

[10] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Intl. Conf. on Soft. Eng.*, pages 396–405, may 2007.