Covering Array Sampling of Input Event Sequences for Automated GUI Testing

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

Myra B. Cohen
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115, USA
myra@cse.unl.edu

Atif M. Memon
Dept. of Computer Science & Institute for Advanced Computer Studies
University of Maryland
College Park, MD, USA
atif@cs.umd.edu

ABSTRACT

This paper describes a new automated technique to generate test cases for GUIs by using *covering arrays* (CAs). The key motivation is to generate long GUI event sequences that are systematically sampled at a particular coverage strength. CAs, to date, have not been effectively used in sampling event driven systems such as GUIs which maintain state. We leverage a "stateless" abstraction of GUIs that allows us to use CAs. Once the CAs have been generated, we reuse the abstractions to reinsert ordering relationships between GUI events, thereby creating executable test cases. A feasibility study on a well-studied GUI-based application shows that the new technique is able to detect a large number of previously undetected faults.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

GUI Testing, Event Driven Software, Covering Arrays

1. INTRODUCTION

The ubiquitousness of graphical user interfaces (GUIs) makes it the most important (and in many cases the only) means used to interact with today's software [5]. As an end user interacts with GUI-based software by performing sequences of *events* (*e.g.*, *Click-on-Cancel-Button*, *Type-in-Text*) on GUI *widgets* (*e.g.*, Cancel-Button, Text-box), the software responds in one of several ways which may include a change of the state of some GUI widgets. Because of this strategic role in a software system, ensuring the quality of GUIs is paramount for ensuring the quality of the overall software. Over the last eight years, we have developed a number of techniques to test GUI-based applications for functional correctness [6, 8]. Most of these techniques employ software models to *generate* sequences of GUI events, which are replayed automatically as test cases.[1]

To date, our most successful automated model-based GUI testing solution uses a directed graph model of the GUI called an event-interaction graph (EIG) [4]. This solution requires generating test cases, each covering a single directed edge (a pair of events) in the EIG. These test cases are referred to as *2-way covering* because each test targets a unique pair of EIG events [4]; in fact, we generate each 2-way covering test case by picking the two events (e_x, e_y) connected by an edge and use graph traversal algorithms to generate an event sequence to "reach" event e_x from a member of the set of events that are available when the GUI application is first launched. This ensures that the resulting test case is executable in the GUI's *start state*. The 2-way covering sequences have a distinct advantage over testing each event in isolation – they expand the number of *states* or contexts in which a particular event is executed, improving the likelihood that a fault is detected. Our previous empirical studies have demonstrated that these test cases reveal a large number of GUI faults. However, there is a class of faults which, we term EIG_{2++} *faults* that are missed by 2-way test coverings. In fact, these faults may only be detected by executing 3-, 4-, or t-way test cases between certain events [4, 8]. A 3-way covering test case is an event sequence $< e_1; e_2; e_3 >$ such that (e_1, e_2) and (e_2, e_3) are directed edges in the EIG; similarly, a 4-way covering test case $< e_1; e_2; e_3; e_4 >$ covers edges (e_1, e_2), (e_2, e_3), and (e_3, e_4); and so on up to t. However, as we increase the number of events in a sequence, the size of the test suite grows prohibitively large, preventing us from using sequences longer than 3 or 4 in most cases.

In this paper we focus on this subset of EIG_{2++} faults and leverage *covering arrays* [2] (see Section 2) to develop a new automated technique for generating GUI test cases. The key motivation behind using covering arrays is to generate longer sequences that are systematically sampled at a particular coverage strength. By using covering arrays we maintain t-way coverage, but we can use any length sequence of at least t. Furthermore we gain additional coverage by testing all t-way sequences from a variety of start states, since covering

[1] http://guitar.cs.umd.edu

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

arrays ensure that a given 2-, 3-, 4-, or t-way relationship is maintained between GUI events in all possible combinations of t-locations in the sequence.

In order to use covering arrays effectively, we employ an EIG model that eliminates the need for ordering relationships between GUI events. This model is then used to obtain the covering arrays. During testing, the rows of the arrays are mapped back to the GUI's original input space in which ordering relationships are reinserted and used to generate test cases. We demonstrate and evaluate our new technique via a feasibility study on one application. The results of our study show that the covering array approach is able to detect a large number of EIG_{2++} faults.

The next section gives an overview of covering arrays. Section 3 presents the major steps of our techniques via a feasibility study. Finally, Section 4 concludes with a discussion of future work.

2. COVERING ARRAYS

A covering array, $CA(N; t, k, v)$, 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 [2]. In other words, any subset of t-columns of this array will contain all t-combinations of the symbols. We use this definition to define the GUI event sequences.[2] To see how this can be applied to GUI event sequences, suppose we want to test sequences of length four and each location in this sequence can contain exactly one of three events (*Clear Canvas, Draw Circle, Refresh*) as is shown in Figure 1. Testing all combinations of these sequences requires 81 test cases. We can instead sample this system, including all sequences of shorter size, perhaps two. We model this sequence as a $CA(N; 2, 4, 3)$ (lower portion of Figure 1). The *strength* of our sample is determined by t. For instance we set $t = 2$ in the example and include all pairs of events between all four locations. If we examine any two columns of the covering array, we will find all nine combinations of event sequences at least once. In this example there are 54 event sequences of length two which consider the sequence location. This can be compared with testing *only* the nine event sequences which would be used in our prior generation technique for a 2-cover (top portion of Figure 1)

The number of test cases required for the t-way property, is N. In our example, we can generate a $CA(9; 2, 4, 3)$. Since the primary cost of running the test case is the setup cost, we cover many more event sequences for almost the same cost as our 2-way cover. In general we cannot guarantee that the size of N will be the same as a shorter sequence, but it will grow logarithmically in k rather than exponentially as does number of all possible sequences of length k [1].

3. FEASIBILITY STUDY

We now demonstrate the main steps of our approach via a feasibility study. The study is designed to compare the fault detection effectiveness (FDE) of t-way covering array sampling with that of *same* and *stronger* t-way coverage on shorter sequences.

[2]A more general definition for a covering array, a *mixed level covering array*, can be defined that allows each location in the array to have a different number of symbols. This type of array is not necessary for this study, since we have the same number of events in each of the k positions.

Events: {Clear Canvas, Draw Circle, Refresh}

2-way covering
1. <Clear Canvas, Clear Canvas>
2. <Clear Canvas, Draw Circle>
3. <Clear Canvas, Refresh>
4. <Draw Circle, Draw Circle>
5. <Draw Circle, Refresh>
6. <Draw Circle, Clear Canvas>
7. <Refresh, Refresh>
8. <Refresh, Clear Canvas>
9. <Refresh, Draw Circle>

Covering Array: CA(9;2,4,3)

Clear Canvas	Clear Canvas	Clear Canvas	Clear Canvas
Clear Canvas	Refresh	Refresh	Draw Circle
Clear Canvas	Draw Circle	Draw Circle	Refresh
Draw Circle	Clear Canvas	Refresh	Refresh
Draw Circle	Draw Circle	Clear Canvas	Draw Circle
Draw Circle	Refresh	Draw Circle	Clear Canvas
Refresh	Clear Canvas	Draw Circle	Draw Circle
Refresh	Refresh	Clear Canvas	Refresh
Refresh	Draw Circle	Refresh	Clear Canvas

Figure 1: 2-way Covering and Covering Array

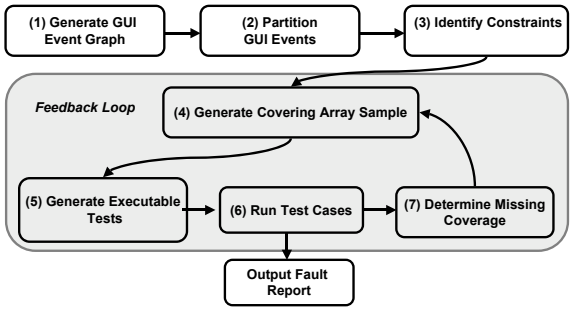

Figure 2: Test Generation Process Using Covering Array Sampling

We use the `TerpPaint` application for this study [6]. The application was hand seeded with 263 faults by several graduate students in the University of Maryland testing team. Since this study is concerned primarily with the fault detection effectiveness of event sequences *longer than two* in order to detect EIG_{2++} *faults*, only the seeded faults which were not detected by the existing 2-way covering EIG test cases were used in this study. There are a total 115 faults out of the original 263 seeded faults that fall into this category.

3.1 Process

Our test process consists of seven steps (see Figure 2). It begins by creating an EIG which is then partitioned into groups of interacting events. Next reachability constraints are identified and used to generate an unordered abstract model for testing. We continue with a feedback loop that generates covering array samples, runs the test cases and determines t-way coverage. If coverage is incomplete we may repeat these steps. We discuss each step in more detail below in relation to our feasibility study.

Step One - *Create EIG model*: The *GUIRipper* tool [3] is used to create the EIG model for `TerpPaint`. The EIG contains *System-Interaction events* (those that interact with the underlying system rather than manipulate the GUI's structure) and *Terminal events* (those that close windows) [4]. Other events such as *Open Window* and *Open Menu* events are omitted from the EIG to improve efficiency. Some of the other events may be inserted back into the final event

Groups	1	2	3	4	5
Description	Tool Mgt.	Image Settings	Clipboard Ops.	Layer Manip.	File Ops.
#Events	27	35	11	11	6
Event Space	27^{10}	35^{10}	11^{10}	11^{10}	6^{10}
CA	$CA(N; 2, 10, 27)$	$CA(N; 2, 10, 35)$	$CA(N; 3, 10, 11)$	$CA(N; 3, 10, 11)$	$CA(N; 4, 10, 6)$
#Test Cases (N)	1055	1783	2870	2870	3428

Table 1: Test Cases Generated by Covering Array Algorithm

sequences in Step 5 if they are needed to reach particular system-interaction events.

Step Two - *Partition the GUI events*: This step, partitions the events by functionality and is done manually. The numbers of events for five groups are shown in Table 1.

Step Three - *Identify constraints*: This step creates our abstract event model. Once the event graphs and groups have been identified, it is necessary to specify constraints on events such that the generated event sequences are executable. This is necessary because some events may not be executable without a set of prior set-up events or must occur only after another event has been fired. The abstract constraint model creates aggregate events for these. For instance, in the events of Figure 1 the *Draw Circle* event may require that an event *Set Ink Color* occurs first. Although this event may not have been of interest in the original EIG graph, it will be needed to reach the event *Draw Circle* and the new aggregate event will have these two events concatenated together. This will be retained as a single abstract event *Draw Circle* in our model. The output of this phase is the full set of aggregate events which can be expanded in later steps into test sequences.

Step Four - *Generate long test cases using covering array sampling*: We construct covering array samples for each of the groups independently. We have chosen coverage criteria for each group based on the number of events. Through previous experimentation, we have determined that abstract sequences of length 10, are about the maximum length sequences that run without failure. We have fixed our sequence length at 10 for this study. Each of the 10 locations in our event sequence can contain any one of the events from this group. This follows the model shown in Figure 1. We determine the strength (or t) for each covering array using a heuristic that generates the highest covering array strength, for each group, that will complete within two days given our current computational resources. This number is approximately 20,000.

Table 1 provides data on the covering array sampling for each group. The #Events row shows the total number of events in each group, the Event Space row, shows the number of sequences that would be required to cover all 10-way combinations of the group. The CA row gives the covering array definition that was used for each group and the #Test Cases row provides the number of test cases in the final covering array sample generated. The covering arrays were created using a simulated annealing program [2].

Step Five - *Generate executable event sequences*: Each test case will be mapped back to executable event sequences through the abstract event model identified in step three.

Step Six - *Execute test sequences*: The test cases are executed on `TerpPaint`. The validity of test cases are identified and the execution time is recorded. To gather oracles (used to determine whether a test case failed or not) for our experiments, all tests are run on the original, non-faulty version of `TerpPaint`. This collects the expected GUI run-time states as each test executes.

Group	3	4	Total
#Regenerated Test Cases	881	1536	2417
#Success	298	222	520
Percent Accum. t-way Cov.	96.1	77.7	NA

Table 2: Regenerated Test Cases

Step Seven - *Regenerate test cases to cover missing event interactions*: In the first two groups all tests executed successfully to completion, *i.e.*, we had 100 percent coverage modeled by the covering array. In the last three groups, however, we had failed test sequences that required a return to step four in the feedback loop For groups three and four, we re-generated covering arrays by identifying which of the 3-sets were already tested These were passed back into our covering array algorithm as an optional argument. We did not do this for the second group during our feasibility study because only 9 tests failed providing almost 100 percent of interaction coverage. In the last group, Group 5, we found that there was a very strict ordering requirement between GUI events, not captured in our abstract model. This may be related to the large number of additional events needed for reachability. Only a small set of the test cases ran to completion (1.8% of the 4-way coverage was achieved). Since the probability of generating new valid test cases was very low, we decided not to re-generate more sequences for this partition. Future work will involve further constraints and better abstract models to avoid these types of problems.

Table 2 shows the lengths of the sequences and achieved accumulated coverage after re-generating and running more sequences. In both of these cases we were unable to achieve 100 percent coverage after 2 iterations of testing. Since the second set of test sequences in GROUP 3 and 4 did not uncover any new faults, we chose not to continue with our feedback loop and re-generate another set of test sequences.

Control Group: As a control group, we ran a test suite $T(orig)$ that uses the EIG algorithm to generate longer test cases of complete t-way coverage. (In previous work this has been the most effective test generation algorithm). It includes two separate test suites $T(same)$ and $T(stronger)$. $T(same)$ includes test cases with the corresponding t-way coverage for each group as the CA, $T(cov)$. For $T(stronger)$, due to the enormous number of test cases that will be generated for each group in full t-way coverage, we generated test cases which were only one-way longer than than the coverage criteria used in the covering array sampling generation. For instance, in GROUP 2, $T(stronger)$ is a 3-cover (all 3-sequences) and for GROUP 3 $T(stronger)$ is a 4-cover.

Table 3 shows the length of the $T(stronger)$ test cases for each group as well as the number of test cases.

Groups	1	2	3	4	5
Length (t-way)	3	3	4	4	5
#Test Cases	19683	42875	14641	14641	7776
#Success	19682	42806	13157	11321	281
Percent t-way Cov.	99.9	99.8	89.9	77.3	3.6

Table 3: $T(stronger)$ Test Case Sizes

3.2 Results

Table 4 shows the fault detection effectiveness or FDE for the covering arrays, sequences, $T(cov)$, using both the original arrays ($T1$) and regenerated arrays ($T2$), vs. two sets of $T(orig)$ test sequences, $T(same)$ and $T(stronger)$. FDE is measured as the number of unique faults detected divided by the total number (115). We can see that the number of faults detected by $T(cov)$ is much higher than that detected by the $T(stronger)$. The FDE increased by 17%. For all groups, the covering array-based test cases detected the same or more faults using a much smaller number of test cases.

Test Suite	$T(orig)$		$T(cov)$	
	$T(same)$	$T(stronger)$	$T1$	$T2$
Group 1	0	49	69	NA
Group 2	0	4	4	NA
Group 3	6	8	8	8
Group 4	0	0	0	0
Group 5	0	5	5	NA
Total(unique)	6	62	82	
FDE	5.22%	53.91%	71.3%	

Table 4: Fault Detection Effectiveness

4. CONCLUSIONS AND FUTURE WORK

This paper presented a new technique to improve GUI test-case generation for EIG_{2++} *faults*. The technique uses covering arrays to generate long event sequences that are systematically sampled at a particular coverage strength. This work is novel in its use of covering arrays to sample sequences for state-based event-driven systems, an abstraction of the GUI that enables efficient use of covering arrays, and empirical demonstration that certain faults can only be detected by using long event sequences. A feasibility study on a well-studied GUI-based application showed that the new technique is able to detect faults that were previously undetected.

Although the results of this first attempt at running a sample of longer sequences that also constitute a shorter t-way covering have been promising, it has also created a number of interesting directions for future research. We are preparing additional subject applications to further evaluate the technique. We have started to examine the automation of the identification of event partitions (**Step Two** of our process). Currently we perform this task manually; the partitions are based on event functionality. A possible automated solution might be based on grouping events using the event-semantic interaction relation that we have defined in earlier work [8], which is based on how events modify GUI widgets. We are also exploring the automation of **Step Three**, *i.e.*, specification of constraints. We feel that these constraints may be automatically inferred from user profiles, *i.e.*, sequences of actual usage of the GUI. We already have access to this data for TerpOffice. This data may also be able to help us weed out unexecutable test cases.

We also see many opportunities to improve the covering array sequences used in this study. We are working on adding temporal constraints that will allow the ordering of specific abstract events to play a role in sequence generation, and are looking at using covering arrays of varying strength [7] to generate test cases. We believe this may further improve overall fault detection. Finally, we feel there may be an opportunity to improve early fault detection through ordering of test cases within the covering array samples.

Acknowledgments

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421 and an EPSCoR FIRST award.

5. 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 Transactions on Software Engineering*, 23(7):437–444, 1997.

[2] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48, May 2003.

[3] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

[4] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

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

[6] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

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

[8] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Minneapolis, MN, USA, May 23–25, 2007.