Test Suite Prioritization by Interaction Coverage

Renée C. Bryce
Computer Science
University of Nevada at Las Vegas
Las Vegas, Nevada 89154
reneebryce@cs.unlv.edu

Atif M. Memon
Computer Science
University of Maryland
College Park, MD 20742
atif@cs.umd.edu

ABSTRACT

Event-driven software (EDS) is a widely used class of software that takes sequences of events as input, changes state, and outputs new event sequences. Managing the size of tests suites for EDS is difficult as the number of event combinations and sequences grow exponentially with the number of events. We propose a new testing technique that extends *software interaction testing*. Traditional software interaction testing systematically examines all t-way interactions of parameters for a program. This paper extends the notion to t-way interactions over sequences of events. The technique applies to many classes of software; we focus on that of EDS. As a proof-of-concept, we prioritize existing test suites for four GUI-based programs by t-way interaction coverage. We compare the rate of fault detection with that of several other prioritization criteria. Results show that prioritization by interaction coverage has the fastest rate of fault detection in half of our experiments, making the most impact when tests have high interaction coverage.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Experimentation

Keywords

combinatorial interaction testing, covering arrays, event driven software, t-way interaction coverage, test suite prioritization

1. INTRODUCTION

Event-driven software (**EDS**) is a class of software that is quickly becoming ubiquitous. All EDS share a common event-driven model – they take sequences of events (i.e., messages, mouse-clicks) as input, change their state, and (sometimes) output an event sequence. Examples include web applications, graphical user interfaces (GUIs), network protocols, device drivers, and embedded software. Quality assurance tasks such as testing have become important for EDS as they are now being used in critical applications.

Our earlier research on a particularly important class of EDS, namely Graphical User Interfaces (GUIs) has shown that existing testing techniques do not apply directly to GUIs primarily because the number of permutations of input events leads to a large number of states, and for adequate testing, an event may, in principle, need to be tested in many of these states, thus requiring a large number of test cases (each represented as an event sequence) [7, 16]. Reduction and prioritization of GUI test suites is an important and challenging area of research.

In this paper, we extend *software interaction testing*. Software interaction testing can systematically examine event interactions. Further, test suites can be generated with a logarithmic guarantee on size [1]. For the purpose of testing GUIs we need to extend "software interaction testing" to consider sequences of events (that contain event interactions). As preliminary work towards this ultimate goal of test-case generation of test suites that are *event-interaction adequate*, we prioritize existing test suites by event interaction coverage. This enables us to understand whether event interactions are indeed useful in testing. Results of our empirical studies demonstrate that prioritization by interaction coverage is useful for two programs under test and their existing test suites. The test suites that benefit the most from our prioritization have high 2-way and 3-way interaction coverage.

The specific contributions of this work include: *(1) the extension of interaction coverage for EDS, (2) application of interaction coverage for GUI test-suite prioritization, and (3) empirical demonstration that higher 2-way and 3-way interaction coverage test suites benefit the most from our prioritization technique.*

Structure of the paper: Section 2 provides background of interaction testing, GUI testing, and test prioritization. Section 3 discusses how to prioritize tests by t-way interaction coverage. Section 4 applies prioritization by 2-way and 3-way interaction coverage to four GUI-based programs and their existing test suites. Section 5 concludes with a discussion of future work.

2. BACKGROUND

This section gives an overview of interaction testing, GUI testing, and test prioritization. Additional details have been

DoSTA'07, September 4, 2007, Dubrovnik, Croatia.

presented in earlier reported research [14].

2.1 Software interaction testing

Software interaction test suites are a collection of tests that systematically cover all t-way combinations of interactions among system inputs. Consider an example of a subset of an on-line store. Table 1 shows four *parameters* and each of their equivalence classes that we refer to as "*options for the parameters*". Interactions include combinations of the options for different parameters. For instance, a pairwise (2-way) interaction for this input is: {Member Status=New Member, Member Type=basic}. Testing pairwise interactions may be useful when exhaustive testing of all parameter-option interactions is not possible. In this example, such exhaustive testing requires 3^4 tests. Pairwise interaction testing needs 9 tests.

Member Status	**Member Type**	**Discount**	**Ship Method**
New Member	Basic	None	Standard (5-7 day)
Verified member	Silver	Employee	Express (3 - 5 day)
Unverified member	Gold	Holiday	Overnight (1 day)

Table 1: Four parameters to a user interface

Software interaction testing has been applied to generate test suites from scratch (see [3] and therein). Empirical studies of interaction testing show promise. For instance, Kuhn *et al.* examine medical devices recalled by the U.S. Food and Drug Administration and identify 97% of defects with pairwise interaction testing [6]. White generates interaction test suites to test component interactions on individual GUI screens [12]. Yilmaz *et al.* find interaction testing useful for fault localization [15]. Bryce *et al.* apply 2-way through 5-way interaction test suites to a Flight Guidance System and find that the tests do not significantly increase model coverage over simple requirements-based test suites [2]. However, they conclude that the interaction tests only consider combinations of input parameters and do not consider state information. This recent work identifies the issue that testing internal states is often important and that interaction testing should evolve to include temporal sequences of events. This issue is important in many types of EDS. In the remainder of this paper, we focus on a particular type of EDS, that of GUI-based programs.

2.2 Event-interaction coverage for GUIs

GUI-based programs are difficult to test. A tester needs to consider end-user behavior when testing GUI-based programs, yet such behavior is often unpredictable. Users can invoke many events from windows. The number of possible event sequences grows exponentially with length. GUI-based programs often have numerous windows with many invocable events.

As exhaustive testing of all possible sequences of events between windows is prohibitive, we consider that we can test combinations of events between windows in a smaller number of tests with interaction testing. However, interaction testing has not yet been applied to generate tests that consider temporal sequences of events. As a precursor to such work, we explore prioritization by event interactions for GUI programs and their test suites that already contain such valid sequences of events.

2.3 Test Suite Prioritization

The prioritization problem is formally defined in [9]. Given (T, Π, f), where T is a test suite, Π is the set of all test suites obtained by permuting the tests of T, and f is a function from Π to the real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi, f(\pi) \geq f(\pi')$. In this definition, the possible prioritizations of T are referred to as Π and f is a function to evaluate the orderings.

Prioritization can be based on any criteria. Examples include code coverage, cost estimates, areas of recent changes, and others [4, 5, 9, 10]. Prioritization by event interactions is a new criterion that we introduce next.

3. INTERACTION COVERAGE FOR PRIORITIZATION

Consider the function for the test prioritization problem by interaction coverage. Given a test suite T, Π is the set of all possible test suites obtained by permuting the ordering of tests. Each permutation is referred to as $\pi_i \in \Pi$ and an individual permutation contains n tests, $\pi_i = (\pi_{i1}, \pi_{i2}, \ldots, \pi_{in})$. A function $\text{tCov}(\pi_{ik})$ computes the set of covered t-tuples in a test π_{ik}. Then our prioritization function is $f(\pi_i) = \sum_{j=0}^{n} | \bigcup_{k=0}^{j} \text{tCov}(\pi_{ik})|$.

The $\text{tCov}(\pi_{ik})$ function that computes the set of covered t-tuples in a test π_{ik} can apply to numerous types of software, EDS being one type of such software. The definition of "t-tuples" may even vary within a particular type of software. In our current experiments, we apply the prioritization function to GUI-based programs and consider t-tuples as combinations of events such that each event is from a unique window. For instance, assume that we have unique windows: ω_1, ω_2, ω_3, and ω_4. Each window has two events as shown in Table 2. **An example test, t=$\{E0 \rightarrow E1 \rightarrow E4 \rightarrow E7\}$ includes four events that are each from three unique windows.** Events E0 and E1 are both invocable from window ω_0 and are not counted as an event interaction in our work since we consider event interactions between unique windows. (Indeed, this is different than work in [13] in which the covered t-tuples in a test include interactions on a single window; this is also different from the definition of event-interaction coverage used in [8], in which an interaction is restricted to events that are related via their `follows` relationship.) The 2-tuples of event interactions that we count for this test include: (E0,E4), (E0,E7), (E1,E4), (E1,E7), and (E4,E7). The 3-tuples include: (E0,E4,E7) and (E1,E4,E7). No valid 4-tuples exist in this test.

ω_0	ω_1	ω_2	ω_3
E0	E2	E4	E6
E1	E3	E5	E7

Table 2: Four windows each have 2 events that may be invoked.

Previous algorithms generate interaction test suites from scratch, whereas our algorithm here prioritizes existing test suites. Further, unlike previous applications of interaction

Input			
ω_0	ω_1	ω_2	ω_3
0	2	4	6
1	3	5	7

Original tests				
Test No.	ω_0	ω_1	ω_2	ω_3
T1	0	2	4	6
T2	0	2	4	7
T3	0	3	5	7
T4	1	2	4	6
T5	1	3	5	6

	Prioritized test selection				
Original Test No.	No. of uncovered 2-tuples				
	Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 5**
T1	**6**	-	-	-	-
T2	6	3	3	**3**	-
T3	6	**6**	-	-	-
T4	6	3	3	2	**2**
T5	6	6	**5**	-	-
Best test	**T1**	**T3**	**T5**	**T2**	**T4**

Step 1. All tests initially cover 6 pairs. Random tie-breaking selects test T1.
T1 contains six new pairs: (0,2), (0,4), (0,6), (2,4), (2,6), (4,6).

Step 2. Now that T1 has been selected as the first test, tests T3 and T5 both cover the most remaining pairs. Random tie-breaking selects test T3.

Step 3. Test T5 covers the most new pairs.

Step 4. Test T2 covers the most new pairs.

Step 5. Test T4 is the last remaining test.

Figure 2: Example trace of prioritizing existing tests by t-way interaction coverage

```
testCount = number of tests to prioritize
bestTest = select a test that covers the most
   unique t-tuples
mark test_bestTest as used
selectedTestCount = 1
while(selectedTestCount < testCount)
    tCountMax = -1
    for j=1 to (testCount-selectedTestCount)
      if test_j is not used
        compute tCount as the number of
          newly covered t-tuples in test_j
        if(tCount > tCountMax)
          tCountMax = tCount
          bestTest = j
        else if(tCount == tCountMax)
            break the tie at random
      end for
      add test_bestTest to T_pi
      mark test_bestTest as used
      selectedTestCount++
end while
```

Figure 1: Pseudocode to prioritize test suites

testing, the tests that we use do not include exactly one option (event) for every parameter (window) in each test. More than one event for a window may occur in the same test, or a test may not include an event from every window. Our algorithm accounts for these issues next.

A simple greedy algorithm can instantiate the prioritization function for t-way interaction coverage. Figure 1 shows one such algorithm. We select one-test-at-a-time to incrementally cover the largest number of previously uncovered t-tuples (of event interactions between unique windows). Ties are broken at random. Once remaining tests do not cover any additional t-tuples, remaining tests are ordered at random.

Figure 2 shows a sample trace of the algorithm for 2-way interaction coverage. The program under test has 4 windows $(\omega_0, \omega_1, \omega_2, \omega_3)$ that each have 2 events. The existing test suite contains 5 tests (T1,...,T5). In the first iteration of the algorithm, all five tests cover 6 new pairs. The tie is broken at random to select test T1. In the second iteration, tests T3 and T5 tie as they both cover the most (6) new pairs. Random tie-breaking selects test T3. In the third iteration, test T5 covers the most new pairs and becomes the third test in the prioritized test suite. In the fourth iteration, there are two remaining tests. Test T2 covers 3 new pairs and test T4 covers only 2 new pairs. Therefore, test T2 becomes the fourth test. In the last iteration, only test T4 remains and becomes the last test in the prioritized test suite.

4. EXPERIMENTS

In the following experiments, we set out to find whether prioritization by event interaction coverage improves the rate of fault detection in four GUI-based programs that have existing regression test suites. In the following sections, we describe the systems under test, characterize the test suites in regards to their coverage of 2-way and 3-way event interactions, and compare the rate of fault detection for tests prioritized by unique coverage of events, 2-way and 3-way interaction coverage, test length, and at random.

4.1 Systems under test

We prioritize test suites for four GUI-based programs shown in Table 3. The table shows each of the program names and the number of windows, widgets, and user-invocable events to test in each program. For instance, the TerpCalc program has 2 windows and $15^1 70^1$ events (read as one window has 15 invocable events and one window has 70 invocable

	TerpCalc	TerpPaint	TerpSpread-sheet	TerpWord
Windows	2	11	9	12
Widgets	82	200	145	112
Events	85	247	188	156
LOC	9916	18376	12791	4893
Classes	141	219	125	104
Methods	446	644	579	236
Branches	1306	1277	1521	452

Table 3: Composition of TerpOffice applications.

events). Since there are only 2 windows for TerpCalc, we can only prioritize by 2-way interactions among windows at most. We may prioritize TerpPaint, TerpSpreadsheet, and TerpWord by higher strength ($t > 2$) interaction coverage since there are more than 2 windows. Table 3 also includes the LOC, classes, methods, and branches for each application.

4.2 Existing test suites

The TerpCalc, TerpPaint, and TerpSpreadsheet test suites contain 300 tests; TerpWord contains 250 tests. The existing test suites that we use cover every window and unique user invocable event at least once. The length of tests vary, as does the composition of tests. The tests contain as many as 47 steps for TerpCalc, 51 for TerpPaint, 50 for TerpSpreadsheet, and 50 for TerpWord.

Additionally, each application has a pre-existing *fault matrix*, i.e., a representation of a set of faults known to be detected by each test case. These faults were similar to those described in earlier reported research [14]. Hence, for each test suite, we can compute the "set of faults detected" by simply taking a set union of the faults detected by all its constituent test cases.

4.3 Prioritization criteria

In these experiments, we prioritize by five criteria (with all ties broken at random): **(1) Unique event coverage** - order tests to cover as many unique events as possible, as early as possible; **(2) Event Interaction coverage (IC)** - order tests by event interaction coverage (we include 2-way and 3-way interaction coverage in the studies in this paper); **(3) Longest to shortest** - order tests by their lengths, from longest to shortest; **(4) Shortest to longest** - order tests by their lengths, from shortest to longest; **(5) Random test ordering** - randomly permute the ordering of tests.

4.4 Results

The results of the prioritization techniques vary among the four GUI-based programs and their test suites. In this section, we begin with a summary of the composition of the existing test suites in regards to the number of 2-way and 3-way interactions that they cover. We follow this summary with a discussion of the rate of fault detection for each of the four programs, reported as the Average Percentage of Faults Detected (APFD). (APFD measures how rapidly a prioritized test suite detects faults. See [9] for a discussion of how APFD is computed.)

Table 4 shows that the existing test suites do not cover all 2-way and 3-way interactions. The TerpCalc test suite includes the largest coverage of 2-way interactions with 99.34% covered. The TerpWord test suite contains the next best coverage with 64.58% of 2-way and 16.51% of 3-way interactions. The TerpSpreadsheet and TerpPaint test suites cover only about half of the possible 2-way interactions and less than 10% of the possible 3-way interactions. In the coming discussions, we observe that TerpCalc and TerpWord have the best event interaction coverage and benefit the most from prioritization by interaction coverage. Table 4 also shows that the available 2-way and 3-way interactions are covered in a subset of the test suite. For instance, the TerpCalc test suite contains 300 tests, but as few as 67 tests cover the available 2-way interactions.

	Calc	Paint	Spread-sheet	Word
	2-way	*2-way*	*2-way*	*2-way*
No. of 2-tuples	1,065	26,253	14,721	10,815
% of 2-tuples covered in test suite	99.34%	46.34%	50.75%	64.58%
No. of tests that cover the unique 2-tuples in the test suite	67	199	103	146
	3-way	*3-way*	*3-way*	*3-way*
No. of 3-tuples	n/a	1,577,160	626,012	439,734
% of 3-tuples covered in test suite	n/a	6.15%	9.76%	16.51%
No. of tests that cover the unique 3-tuples in the test suite	n/a	185	127	168

Table 4: Summary of 2 and 3-tuples covered in the TerpCalc, TerpPaint, TerpSpreadsheet, and TerpWord test suites

4.5 TerpCalc

The results of the prioritized test suites are shown in Figure 3. The figure includes graphs of the rate of fault detection in (a) logarithmic scale to better visualize the impact of prioritization in earliest tests and also (b) unscaled. Prioritization by 2-way interaction coverage is the most effective technique during the initial tests. The results of the initial 30% of the tests show that prioritization by 2-way event interaction has the best APFD. Prioritization by test length (longest to shortest) has the second best APFD, followed by prioritization by unique events. Randomly ordered tests are less effective and prioritization by shortest to longest test length is the least effective. The first 30% of the test suite is most interesting here because all 2-way event interactions are covered in the first 67 tests (we then prioritize the remaining tests at random).

The APFD for the first 10 tests is best with 2-way interaction coverage prioritization. Prioritization by unique events is the second most effective in these initial tests. While covering all unique events early is useful in this example, it does not apply for prioritizing the entire test suite here since all unique events are covered in the first 7 tests (we prioritize the remaining 293 tests at random).

After the initial 30% of the tests are run, prioritization

by test length (longest to shortest) is the most effective technique. Prioritization by 2-way interaction coverage and unique events follows in effectiveness. Random ordering and prioritization by length (shortest to longest) are least effective.

4.6 TerpPaint

Figure 4 shows that in TerpPaint, prioritization by the length of tests (longest to shortest) has the best overall APFD. Prioritization by 3-way interactions is the second best; by 2-way is the third best; unique event coverage is fourth best; random is fifth; and length (shortest to longest) is least effective.

4.7 TerpSpreadsheet

Figure 5 shows that prioritization by unique event coverage is initially most effective, followed by 2-way and then 3-way interaction coverage. After 50% of the tests are run, 2-way and then 3-way interaction coverage are most effective. Prioritization by length (longest to shortest) is generally less effective (especially in earliest tests), but works better than random ordering and prioritization by length (shortest to longest).

4.8 TerpWord

Figure 6 shows that the TerpWord test suite has the most effective APFD when prioritized by 2-way and 3-way interaction coverage (each slightly outperforming each other at different points in the test suite). Prioritization by test length (longest to shortest) and unique event coverage are the next most effective. Random test ordering and prioritization by length (shortest to longest) are the least effective.

4.9 Summary of Results

Our experiments show that test suites with the highest event interaction coverage benefit the most from our prioritization technique. Given that all of our test suites are in a comparable range of size (250 to 300 tests) and yet the number of t-tuples of event interactions are quite different for the 4 applications (see Table 4), perhaps it is not surprising that the test suites for the programs with the larger number of t-tuples of possible event interactions, do not have high interaction coverage in the limited number of tests. For instance, TerpCalc has the fewest number of windows, widgets, and events. The TerpCalc test suite has the best 2-way interaction coverage among those in our experiments; as few as 67 out of the 300 tests cover 99% of the 2-way event interactions. TerpWord is our smallest application in terms of LOC and has the second smallest number of event interactions to cover. TerpWord has the second highest 2-way interaction coverage and the best 3-way interaction coverage in our experiments. Similar to TerpCalc, it benefits from prioritization by 2-way, and also from 3-way interaction coverage. TerpSpreadsheet has more events than TerpCalc and TerpWord; the test suite has less interaction coverage and does not benefit as much from prioritization by event interaction coverage. Further, TerpPaint is the largest application and has a test suite with the least event interaction coverage. Prioritizing the TerpPaint test suite by event interaction coverage does not improve the rate of fault detection. Indeed, these results show that prioritization by interaction coverage is quite useful in some cases, particularly when test suites have higher interaction coverage.

4.10 Threats to Validity

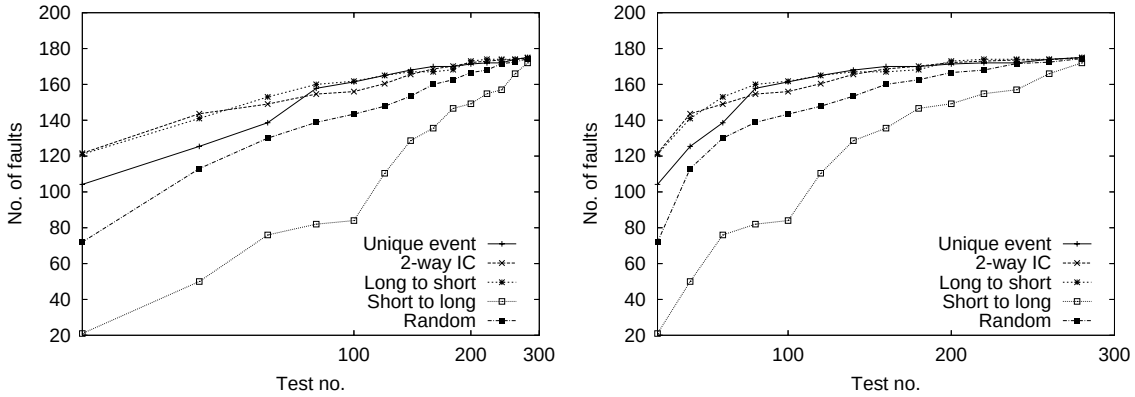
Threats to external validity are factors that may impact our ability to generalize our results to other situations. Our main threat to external validity in this experiment is the small number of subject applications. In this study, we only run our data collection and test suite prioritization process on four programs, which we chose for their availability. These programs were constructed in more or less the same manner and may not be representative of the broader population of programs. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. Moreover, the characteristics of original test suites (such as their fault detecting ability and how they were constructed) play a role in the size and fault detection reduction results. This threat can be addressed in future work by choosing original test suites adequate for a variety of coverage criteria.

Threats to construct validity are factors in the experiment design that may cause us to inadequately measure concepts of interest. In our experiment, we made some simplifying assumptions in the area of costs. In test suite prioritization, we are primarily interested in two different effects on costs. First, there is potential savings obtained by running "more effective" test cases sooner. In this study, we assume that each test case has a uniform cost of running (processor time) and monitoring (human time); these assumptions may not hold in practice. Second, we assume that each fault contributes uniformly to the overall cost, which again may not hold in practice.

5. CONCLUSIONS AND FUTURE WORK

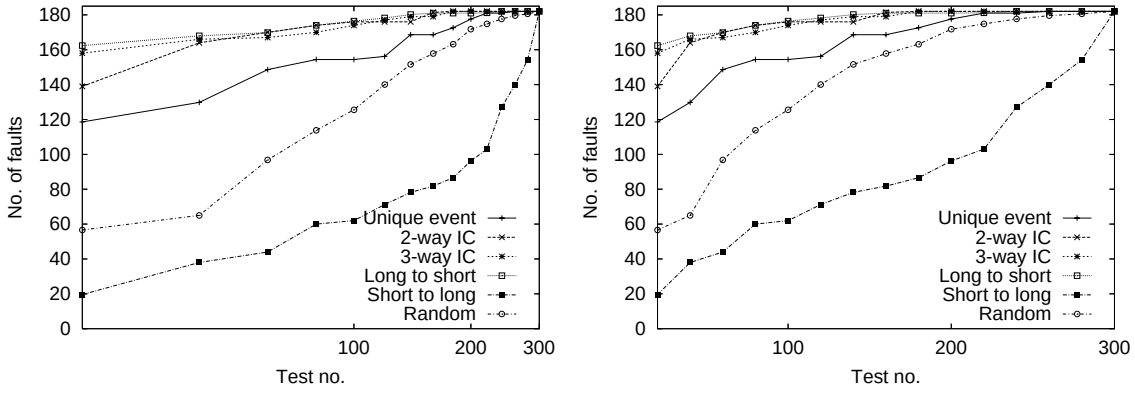
Event driven software (EDS) is a widely used class of software that requires novel testing techniques that can adequately test software with a manageable number of tests. We propose one technique for this purpose that extends previous applications of software interaction testing. Previous work on interaction testing generates test suites that cover t-way combinations of input parameters. This paper suggests that interaction testing evolve to consider temporal sequences of events. We begin to look at this issue by using interaction coverage to prioritize existing test suites for four GUI-based programs. Prioritization by interaction coverage of events improves the rate of fault detection in half of our experiments. The test suites that include the largest percentage of 2-way and 3-way interactions have the fastest rate of fault detection when prioritized by interaction coverage. The test suites with the highest event interaction coverage are our smallest programs that have fewer t-way event interactions to cover. These results raise the need for future work that identifies criteria for the "event-interaction adequacy" of test suites. In addition, techniques to generate test-cases that meet such event-interaction adequacy are needed.

Our ultimate goals are to (1) identify criteria for "event interaction adequacy" of tests suites and (2) to develop tools that automatically generate tests that meet such criteria. This is particularly useful in GUI-based testing where it is difficult to predict end user behavior (in which users may execute numerous event sequences). The application of interaction testing to systematically examine possible event sequences in EDS is a new contribution that we have just begun to examine.



(a) log scale (b) unscaled

Figure 3: TerpCalc: rate of fault detection using prioritized test orderings.



(a) log scale (b) unscaled

Figure 4: TerpPaint: rate of fault detection using prioritized test orderings.

In this work, we show that prioritization by interaction coverage can be useful. Future work may extend this prioritization work to also consider: *prioritization with respect to test lengths and incremental t-way interaction coverage.* We discuss each of these issues.

Prioritization by length: One major issue that arises when prioritizing existing tests for GUIs is that tests may be of varying lengths; those with more steps may likely cover more interactions than tests of shorter length. Should one give preference to a test that takes 10 seconds to run and covers 6 interactions; or should one give preference to two tests that take 5 seconds each, but cover a total of 8 interactions? (This problem is similar to the "time-based test prioritization problem" in [11].)

Work in [14] shows that it is useful to give preference to running many shorter tests rather than fewer tests of longer length. They rapidly identify many "shallow" faults with shorter tests and then recommend using remaining time to run longer tests that find "deeper" faults. Our results show that the test prioritization technique of "shortest to longest length" has the slowest rate of fault detection on a test-by-test basis, but do not take the time to run the tests into account. Future work will examine this issue.

Prioritization by incremental *t*-way coverage: In the tests that we examine, the available 2-way and 3-way unique event interactions are covered in only a subset of the test suite. While we prioritized remaining tests at random in our experiments, remaining tests may instead be prioritized by higher strength ($t > 2$) interactions. For instance, prioritize the first ℓ tests to cover all unique events. Prioritize the next batch of tests by t=2 interaction coverage. Once remaining tests do not cover any additional t=2 (pairwise) interactions, order remaining tests to cover the most t+1 interactions. This process continues until all tests are prioritized by ascending order of t-way interaction coverage.

6. ACKNOWLEDGEMENTS

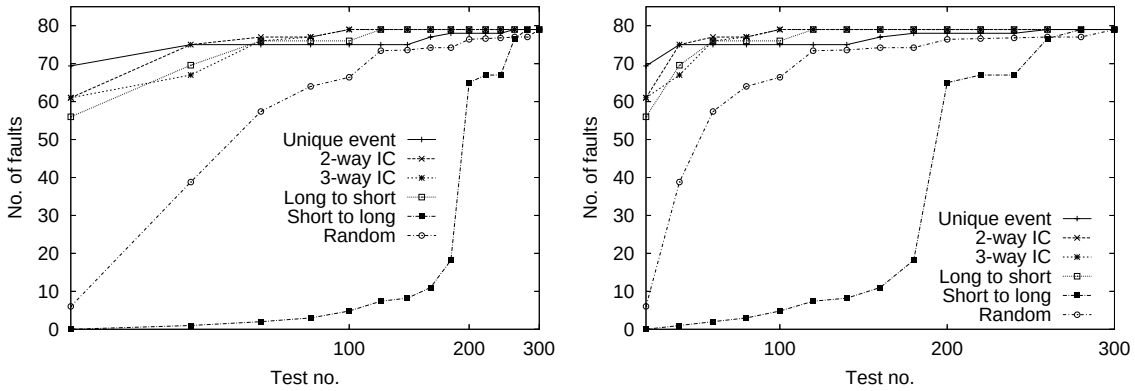
This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

7. REFERENCES

[1] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Journal of Software Testing, Verification, and Reliability*, to appear.

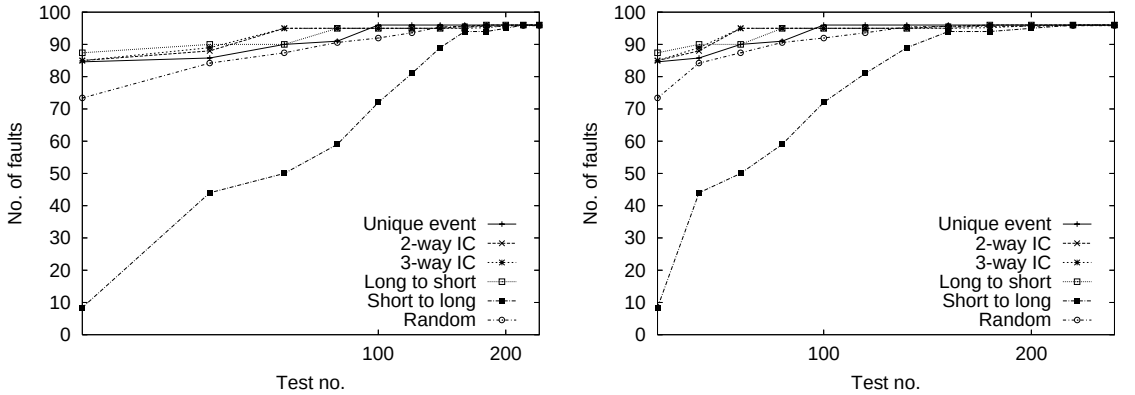
[2] R. C. Bryce, A. Rajan, and M. P.E. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. *Proc. of the* 13^{th} *Asia-Pacific Software Engineering Conf.*, pages 258–269, Dec. 2006.

[3] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58:121–167, 2004.



(a) log scale (b) unscaled

Figure 5: TerpSpreadsheet: rate of fault detection using prioritized test orderings.



(a) log scale (b) unscaled

Figure 6: TerpWord: rate of fault detection using prioritized test orderings.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Software Engineering*, 18(2):159–182, 2002.

[5] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[6] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. on Software Engineering*, 30(6):418–421, Oct. 2004.

[7] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 2007.

[8] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proc. of the 8th European software engineering conf. held jointly with 9th ACM SIGSOFT Int. symposium on Foundations of software engineering*, pages 256–267, 2001.

[9] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *ACM Trans. on Software Engineering and Methodology*, 27(10):929–948, 2001.

[10] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of the Int. Symposium on Software Testing and Analysis*, pages 97–106, Jul. 2002.

[11] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *Proc. of the Int. Symposium on Software Testing and Analysis*, pages 1–12, Jul. 2006.

[12] L. White. Regression testing of gui event interactions. In *Proc. of the Int. Conf. on Software Maintenance*, pages 350–358, Nov. 1996.

[13] L. White and H. Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Proc. of the Interactional Symposium on Software Reliability Engineering*, pages 110–121, 2000.

[14] Qing Xie and Atif M. Memon. Studying the characteristics of a 'good' GUI test suite. In *Proc. of the 17th IEEE Int. Symposium on Software Reliability Engineering*. IEEE Computer Society Press, 2006.

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

[16] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proc. of the 29th Int. Conf. on Software Engineering*, May 2007.