

# Employing User Profiles for Regression Testing of GUIs

Penelope Brooks and Jaymie Strecker  
Department of Computer Science  
University of Maryland  
College Park, Maryland, USA  
{penelope,strecker}@cs.umd.edu

## Abstract

*Graphical user interfaces (GUIs) prevail in today's software. Like any other part of an application, a GUI can house faults; hence GUIs need to be tested during and after development and re-tested after modifications. Current regression testing techniques select a subset of a GUI's state space to re-test. A weakness of these techniques is that the regression test suite may not adequately test parts of the state space that users of an application frequently encounter; hence these techniques may miss failures that users may encounter. This paper presents a new technique to capture GUI user profiles with very little overhead, employ the profiles to populate a probabilistic state machine model of the GUI, and generate test cases from this model. User profiles may be collected from fielded (previous) versions of an application. An empirical study on one large application shows that test cases generated from the new model successfully reveal faults that an event coverage-adequate test suite does not.*

## 1 Introduction

Graphical user interfaces (GUIs) prevail in today's software. For many users, the GUI is the most important part of the software, as it is the only part with which they interact. Like any other part of an application, a GUI can house faults; hence GUIs need to be tested. Exhaustive testing is infeasible in general, and GUIs are no exception [27]. GUI testing presents new challenges that impede conventional approaches to testing. Previous work has addressed challenges of test case generation, test oracles, test coverage, and regression testing [19, 17, 18]. The work presented in this paper focuses on regression testing.

GUI regression testing presents new challenges. Previous work has addressed the problems of obsolete test cases [18] and model-based testing of rapidly evolving software [20]. Memon showed that modifications to a GUI render a

large portion of the previous version's test cases unusable, since they exercise GUI event interactions that no longer exist [18]. To salvage these obsolete test cases, an automated method of discovering and repairing some obsolete test cases was presented. In other work, it was shown that test suites that cover all feasible 2-way event interactions (that is, interactions among at most 2 GUI events) reveal a large number of faults [20]. A weakness of these techniques is that they are not representative of the way users typically interact with the GUI. These techniques may select all 2-way event interactions, including pairs such as *Save* and *Save As* that are unlikely to occur consecutively in actual usage, yet omit interactions users frequently exercise, such as the sequence of mouse clicks and text entries used to open a document. Since faults detected by in-house testing cost less to repair than faults exposed when users encounter failures, an effective test suite must remove as many faults from the latter category as possible. Increasing the degree of the event interactions (e.g., from 2-way to 3-way) is not a viable solution, since the number of selected test cases grows exponentially.

An alternative approach, and the one used in this paper, is to collect information about the way users interact with previous versions of an application and use this information to test versions under development. Usage data may be collected in the form of user profiles (sequences of events executed) and replayed by a capture-replay tool [22]. However, incorporating information from user profiles into GUI regression testing presents another set of challenges.

First, collecting real user data, in the form of user profiles, is challenging. It requires instrumenting the code or running a tool in the background to collect the user interactions with the GUI, which could adversely affect the performance of the application being monitored. Second, user profiles are collected either on fielded software in preparation for the next version or on a beta version before primary release of the application. User profiles are gathered with the goal of automatically generating test cases and replaying them with minimal human interaction. However, changes

in the GUI structure between applications could render collected user profiles obsolete in subsequent test case generation [18].

Previous researchers have used profiling techniques in a number of ways, but very few have combined profiling with GUI testing. One researcher captured usage data from customers to determine which failures in a GUI application were most critical Donovan. Other researchers have created tools to capture and replay users' interactions with the GUI [26]. Still others have used *a priori* knowledge of the application to populate a model upon which to base GUI testing [33]. Most of these techniques provide very little automated processing, requiring testers to spend a lot of time interacting with the GUI to develop appropriate test cases.

To address the challenges discussed above, this paper presents a new technique to capture GUI user profiles with very little overhead, employ the profiles to populate a probabilistic model of the GUI, and generate test cases from this model. Test cases are generated in descending order of their estimated likelihood of being executed by real users. With a parameter called *history*, testers can control the way this likelihood is estimated and, indirectly, the fault detection ability and cost of testing. The probabilistic GUI model can be used to identify event interactions in the user profiles that have become obsolete and salvage the fragments of the profiles that remain usable. In order to relate different application versions to each other, we have created a mapping method that can correct the test files and allow them to be replayed on the new version.

An empirical study was conducted to measure the benefit usage data adds to the regression test selection process. We compared three test suites: one composed of collected user profiles, one composed of test cases generated from our model (populated with the same set of user profiles), and one composed of all 2-way event interactions. For a large word-processing application, test cases generated from our model detected 11 of 295 seeded faults, including 3 that a 2-way interaction-adequate test suite did not detect.

The work presented in this paper makes several contributions, including:

1. a probabilistic finite state model representing the event-flow graph of the GUI, which can be populated with usage patterns gathered from users of the fielded application,
2. a method to populate the model with user information gathered from real application usage,
3. an algorithm to generate test cases based on the model with a tuneable history parameter, and
4. an empirical investigation comparing the size and fault detection ability of a test suite generated by our technique to a 2-way interaction-adequate test suite for var-

ious values of the history parameter on a large, fielded application.

The next section presents the background and related work. In Section 3, the probabilistic GUI model and test case generation algorithm are detailed. The results of our empirical study are given in Section 4, followed by conclusions and a discussion of future work in Section 5.

## 2 Related work

Our work relates to several overlapping areas: model-based testing, stochastic testing, user profile-based testing, and GUI testing.

### 2.1 Model-based and stochastic testing

State machines are a well-known way to model complex software. Behavior models are one type of state-machine model. In a behavioral model, nodes represent program states or sets of related states, and edges represent transitions between states. The model encodes the behavior of the program. Tools exist to generate test cases from a behavioral model by traversing paths through the model and outputting a test case specification for each path [3].

In another type of state-machine model called an operational model, nodes represent operations a user can perform, and nodes, edges, or paths can be weighted and labeled with the probability that a user traverses them. Whittaker and Thomason [31] have generated test cases stochastically using a Markov model version of the operational model, in which the probability of visiting a node depends only on the previously visited node. Özekici and Soyer [24] have extended this idea to a Bayesian framework, in which model parameters such as probability labels can be learned and updated during testing.

Woit [33, 32] extended the Markov framework in a different direction, toward a more general conditional probability framework. This framework more accurately models software for which event probabilities depend on a longer history of events. To define model parameters, Woit's method requires the space of event histories to be partitioned into a manageable number of subsets and probabilities to be found for each subset. A test case is generated by choosing a path through the model stochastically using the probability distribution observed or estimated for actual usage.

Some model-based approaches to testing have focused on GUIs. In an approach used by Dalal et al [4], testers use a tool to capture the functional model of the data, guided by the structure of the GUI. The tool generates test cases based on valid inputs to the model. Ostrand et al [23] have developed a test development environment (TDE) for GUI-based

applications. The TDE models the GUI with a *top-level* graph, in which each node represents a window and each edge represents a user action. A *component representation* models each window in more detail. To generate test cases, a tester can interact with the GUI model, or the TDE can capture users' interactions with the GUI and replay them.

## 2.2 User profile-based testing

User profiles, also called operational profiles or user session data, have been used to add realism (user-like behavior) to test suites for user interfaces. User profiles have been recorded using capture-replay tools [5, 26, 6, 22] and a combination of written logs and videotaping [9]. The collected user profiles may be replayed unchanged on the version under test, or they may be employed to generate additional test cases. Elbaum et al [7] have generated test cases for a web application by mixing subsequences of different users' session data. Collections of user profiles can also be used to estimate model parameters such as event interaction probabilities in the operational models discussed in section 2.1.

Weyuker [29] has proposed the use of operational profile information to measure test suite coverage. The percentage of probability mass of the operational profile distributions covered during testing can indicate an application's readiness for release.

## 2.3 GUI testing and verification

Formal methods have been used to specify and verify GUIs. Berstel et al [2] have used *visual event grammars* to write formal specifications that can be verified by a model checker. However, formal verification has not supplanted testing.

Typically, GUI testing follows one of three approaches. Unfortunately, the most popular approach is to skip GUI testing altogether, leading to poor software quality and software that does not perform as expected [13]. The second approach involves using test harnesses that invoke methods in the business logic of the software, as if initiated by the GUI, without using the GUI at all. This approach not only requires major changes to the software architecture (e.g., ensuring the GUI software is not too cumbersome and coding all of the important decisions in the business logic [14]), but also does not actually test the end-user software. The third approach provides only limited testing via manual GUI testing tools [8, 30]. Popular tools include extensions of JUnit such as JFCUnit, Abbot, Pounder, and Jemmy Module [11] and capture/replay tools [10] that provide very little automation.

Capture/replay tools (also called record/playback tools) operate in two modes: Record and Playback. In the Record mode, these tools record user actions as test cases. In the

Playback mode, the recorded test cases are replayed automatically. Many of these tools operate by storing mouse coordinates, however, causing test cases to break with the slightest changes to the GUI layout. Tools such as Winrunner [21], Abbot [1], and Rational Robot [25] avoid this problem by capturing GUI widgets rather than mouse coordinates. Although playback is automated, significant effort is involved in creating the test scripts, detecting failures, and editing the tests to make them work on the modified version of software. Testers who employ these tools typically come up with a small number of smoke tests to utilize during development [15].

## 2.4 Summary

Our work most closely resembles Woit's, described in section 2.1. We extend this work in several ways. Instead of generating test cases stochastically, based on the operational distribution observed during usage, we select test cases that cover the *most likely event interactions* to be executed and ignore the least likely. We integrate our probabilistic GUI model into an infrastructure for collecting user profiles, calculating model parameters, and generating and running test cases. Also furthering Woit's work, we empirically evaluate the effectiveness of the generated test cases. With respect to other related work, our work is, to our knowledge, the first to apply a combination of user profiling and model-based testing to regression testing or testing of GUIs.

# 3 Probabilistic model of GUI interactions

## 3.1 Constructing a probabilistic EFG

A GUI consists of a set of windows, each of which contains a set of widgets such as buttons, text fields, and menu items. An event occurs when a user manipulates a widget. Events may result in changes to the GUI state (e.g. opening or closing a window, changing displayed text) and to the state of the invisible business state of the application.

We model a GUI's behavior with a directed graph called the *event flow graph*. In this graph, nodes represent events and edges represent the *follows* relationship: an edge from event  $e_1$  to event  $e_2$  exists if and only if  $e_2$  may be invoked immediately after  $e_1$ , without any intervening events. The predicate  $follows(e_2, e_1)$  denotes that  $e_2$  follows  $e_1$ . Since events can typically be executed more than once during a session with an application, an EFG is potentially cyclic.

In previous work, others have generated test cases by traversing all event interactions up to a specified length or a randomly selected subset of these [20]. However, this method ignores an important fact about GUIs (and software in general): in real usage, not all paths are equally likely to be exercised. Hence, we augment the EFG by encoding

an estimate of the probability for each path. Our method is closely related to the use of N-gram models in computational linguistics; see for example [12].

Consider an EFG whose  $E$  events (nodes) make up the set  $\{e_1, e_2, \dots, e_E\}$ , and let us augment the EFG with special nodes  $INIT$  and  $FINAL$ . By adding transitions from  $INIT$  to each initial node and from each final node to  $FINAL$ , making  $INIT$  the graph's only initial node, and making  $FINAL$  the only final node, we impose a single-entrance, single-exit constraint on the EFG. Suppose we have a collection of  $R$  paths through the EFG called  $r_1, r_2, \dots, r_R$ . Each path  $r_i, 1 \leq i \leq R$ , consists of a sequence of  $length(r_i)$  events in addition to  $INIT$  and  $FINAL$ :

$$r_i = INIT, r_i(1), r_i(2), \dots, r_i(length(r_i)), FINAL;$$

$$\forall j \ r_i(j) \in \{e_1, e_2, \dots, e_E\} \wedge$$

$$(j = length(r_i) \vee follows(r_i(j+1), r_i(j))).$$

Let  $count(e_i)$  return the number of times event  $e_i$  occurs in the paths  $r_1, r_2, \dots, r_R$ . The prior probability that a randomly selected event from any of  $r_1, r_2, \dots, r_R$  turns out to be  $e_i$  is the following:

$$P(e_i) = \frac{count(e_i)}{\sum_{j=1}^E count(e_j)}.$$

Now let us extend  $count$  and the prior probability calculation from single events to sequences of events. Let  $s$  be a length- $S$  subsequence of some path through the EFG (not necessarily in  $r_1, r_2, \dots, r_R$ ):

$$s = s(1), s(2), \dots, s(S)$$

$$\forall j \ s(j) \in \{INIT, e_1, e_2, \dots, e_E, FINAL\} \wedge$$

$$(j = length(s) \vee follows(s(j+1), s(j))).$$

The prior probability that a randomly selected, length- $S$  subsequence from any of  $r_1, r_2, \dots, r_R$  turns out to be  $s$  is

$$P(s) = \frac{count(s)}{\sum_{s_i \in subs(S)} count(s_i)},$$

where  $count(s)$  returns the number of times  $s$  occurs as a subsequence of  $r_1, r_2, \dots, r_R$  and  $subs(S)$  is the set of all length- $S$  subsequences in  $r_1, r_2, \dots, r_R$ .

Given that  $s$  immediately precedes  $e_i$ , the conditional probability of  $e_i$  is

$$P(e_i|s) = \frac{P(s(1), s(2), \dots, s(S), e_i)}{\sum_{j=1}^E P(s(1), s(2), \dots, s(S), e_j)}.$$

Note that  $P(e_i|s)$  can be thought of as  $P(e_i)$  when  $s$  has length 0. This is not the same as  $P(e_i|INIT)$ . Rather, this

is the probability of  $e_i$  given *no information* about the events that precede it.

We annotate each event (node) in the EFG with a table containing the event's prior probability and its probability conditioned on each subsequence in  $\{r_1, r_2, \dots, r_R\}$  up to some maximum subsequence length, or history,  $H$ . This annotated EFG is our *probabilistic EFG*, and each set of entries for all length- $h$  histories,  $0 \leq h \leq H$ , succinctly encodes a probabilistic finite state machine whose  $O(E^h)$  nodes correspond to length- $h$  histories and whose edges are labeled with conditional probabilities.

For example, consider the following set of paths through the EFG shown in Figure 1:

$$r_1 = INIT, e_1, FINAL$$

$$r_2 = INIT, e_1, e_3, e_1, e_2, FINAL$$

$$r_3 = INIT, e_1, e_2, e_2, e_2, e_1, FINAL$$

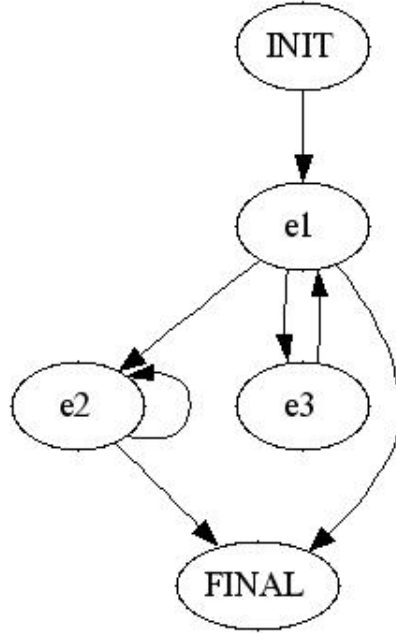


Figure 1. Example Event Flow Graph

Table 1 shows the conditional probability table of event  $e_1$  for histories up to length 5. Zero-probability entries are omitted. The first data row in the table gives the prior probability of  $e_1$ .

### 3.2 Generating test cases

The algorithm in Figure 2 generates test cases by constructing and traversing the probabilistic EFG. In the pseudocode shown, sets of event sequences, such as the input set of user profiles and the output set of test cases, are stored as

h	$P(e_1 h)$
-	0.3125
<i>INIT</i>	1.0
$e_2$	0.25
$e_3$	1.0
$e_1, e_3$	1.0
$e_2, e_2$	0.5
<i>INIT</i> , $e_1, e_3$	1.0
$e_2, e_2, e_2$	1.0
$e_1, e_2, e_2, e_2$	1.0
<i>INIT</i> , $e_1, e_2, e_2, e_2$	1.0

**Table 1. Table showing probabilities relating to Figure 1.**

matrices in which each row holds an event sequence and the  $i$ th column of a row holds the  $i$ th event in the sequence. The algorithm takes two parameters: *Profiles*, the set of user profiles, and *maxHistory*, the maximum number of previous events on which the probability calculations are to be conditioned. The output is *TestSuite*, a set of test cases.

In step 1, the *Histories* matrix is constructed by adding in each unique event subsequence up to length *maxHistory* in *Profiles*. In step 2, the probability distribution  $P(e|h)$  is computed for each unique event  $e$  in *Profiles* and each row  $h$  in *Histories* and is stored in *Distributions*. For each unique event  $e$ , step 3 calculates the most probable path from *INIT* to  $e$ . Finally, in step 4, the matrix *TestSuite* is constructed by adding in a legal test case (i.e. one that begins with *INIT*) for each column maximum in *Distributions*.

## 4 Empirical studies

We conducted a series of empirical studies to evaluate the effectiveness of our model-based test case generation technique (PEFG) in relation to two other techniques. Those techniques were to replay the user profiles “as-is” (*PROFILES*), without using the probabilistic EFG, and to test each 2-way event interaction (EFG), or in other words each EFG edge. The studies tested the following hypotheses:

1. The PEFG technique reveals faults that neither *PROFILES* nor EFG reveals.
2. Reducing the history parameter in the PEFG technique (down to some minimum length) significantly decreases the cost of testing without significantly decreasing the number of faults detected.

Under the assumption that the user profiles used to populate the probabilistic EFG are representative of usage in the

field, the first hypothesis implies that PEFG detects more of the faults that are likely to affect users. The second hypothesis asserts that, if the minimum sufficient history length is known or estimated for some application, the time required to test the application can be greatly reduced without loss of fault detection ability.

### 4.1 Measurement

These studies measure two dependent variables: the number of faults detected and the cost of testing. To measure fault detection ability, the result of running each test case on a set of fault-seeded versions of the subject application was compared to the result of running the test case on the “golden,” or unmodified, version. Fault seeding is often used to evaluate testing techniques when no specification or test oracle for the subject application is available. The golden version becomes the *de facto* test oracle. In this study, each fault-seeded version contained a single fault in order to avoid complex results caused by fault interactions.

For GUI applications, the number of faults a test suite detects depends on the rigor of the oracle procedure used to decide if a test case passes or fails [16]. For a test case that exercises some widget, the oracle procedure may check the state of the widget, the state of all widgets in the exercised widget’s window, or the state of all widgets in all visible windows. The oracle procedure may perform the check after each step (event) in the test case or only after the last step. In these studies, the oracle procedure checked the state of all widgets in all visible windows after the final step in each test case.

As a proxy for cost, we measured test suite size. In the context of this study, we believe this approximation is reasonable since the overhead (set-up and tear-down time) required to run each test case seemed to consume more computation time than any other activity. We did not measure the human effort for activities such as recruiting users to observe.

### 4.2 Infrastructure

These studies were carried out with the GUI Testing Framework (GUITAR) [20]. To capture user profiles, we used a tool in GUITAR’s family of applications called the Profiler [22]. (The Profiler does not currently belong to GUITAR’s canonical, publicly available set of tools.) Running the subject application through Java Reflection, the Profiler attaches its own event handlers to each *JButton*, *JTextArea*, and *JMenuItem* that becomes visible. When one of the Profiler’s event handlers is triggered, the Profiler records an identifier for the widget and the type of event. Because the Profiler’s output is incompatible with the latest version of GUITAR, we used several scripts, described

```

Input: Profiles, maxHistory

# 1. Construct Histories
for each row Row of Profiles
  for h from 1 to maxHistory
    for each unique length-h subsequence Seq of Row
      Histories := Histories unioned with {Seq}
    end
  end
end

# 2. Construct Distributions
for each row index i of Histories
  l := length of Histories(i)
  numTotal := number of length-l subsequences in Profiles
  for each unique element e in Profiles
    Seq := row i of Histories concatenated with e
    numSeq := number of occurrences of Seq in Profiles
    Distributions(e, i) := numSeq / numTotal
  end
end

# 3. Construct BestPrefixes
for each unique element e in Profiles
  i := index such that Histories(i) = e
  Prefixes(e) := row indices of Histories for rows whose first element is INIT and
    whose last sequence element is e
  BestPrefixes(e) := element j of Prefixes(e) such that Distributions(j, e) is the
    maximum in its column
end

# 4. Generate TestSuite
Maxes := index (indices) of row(s) with maximum value in each column of Distributions
for each element index i of Maxes
  KeyEvents := row i of Histories
  Prefix := BestPrefixes(first element of KeyEvents)
  TestCase := Prefix concatenated with KeyEvents
  TestSuite := TestSuite unioned with {TestCase}
end

Output: TestSuite

```

**Figure 2. Test case generation algorithm**

in Section 4.5, to convert the user profiles to a compatible format.

GUITAR's JavaGUIReplayer tool executed each of our test cases, recording the GUI's state upon completing each test case. For each test case, the OracleInfoVerifier tool compared the recorded state for each fault-seeded version to the recorded state for the golden version.

### 4.3 Subject application

In these studies, the subject application was TerpWord 3.0, a member of the TerpOffice suite [28]. Developed by

undergraduate software engineering students at the University of Maryland, TerpWord is an open-source word processor comparable to Microsoft WordPad. Implemented in Java, TerpWord contains 4,893 lines of code. Most of this code pertains to the GUI; the underlying business logic is relatively simple. We selected TerpWord because of its substantial size and its availability to other researchers. However, the fact that Terpword was developed by students poses a threat to external validity.

To compensate for some weaknesses of the Profiler, such as its inability to handle dynamic GUI components like recently-opened-document menus that grow and shrink, we

made minor modifications to TerpWord. Since the version of TerpWord used to collect profiles differed slightly from the golden version used in the replaying stage, we constructed a mapping from Profiler outputs to JavaGUIReplayer inputs to bridge this gap.

An attempt was made to seed realistic faults in the subject application. The 295 seeded faults were created by examining Bugzilla reports for bugs that had been repaired in TerpWord and inserting similar bugs into the source code.

#### 4.4 Collection of user profiles

To elicit realistic user session data, we conducted a user study in which we asked participants to perform a task in TerpWord. The eight participants were members of an undergraduate software engineering class at the University of Maryland. The study was conducted in place of one scheduled class meeting, and students could opt out without penalty to their grade.

After signing a consent form, each participant worked at his or her own Linux workstation. We began by asking the participants to familiarize themselves with the GUI components available in TerpWord. During this time, we observed participants throughout the room to ensure that they understood the parts of the application they would need to complete the task. After the practice period, participants were instructed to execute a script which ran TerpWord through the Profiler and informed subjects about the task they were to perform. Participants were given roughly ten minutes to complete the task, in which they modified the text and formatting of a one-page document to match a printed document as closely as possible. This task was chosen to exercise only parts of the GUI that a novice user would understand (such as items in the File and Edit menus) and to elicit a different sequence of interactions with the GUI from each participant.

Since the sample of user profiles collected was too small to populate the probabilistic EFG effectively, we augmented our set of user profiles with roughly 900 functional test cases created by the developers of TerpWord 3.0 using the Profiler. Hereafter, we refer to the profiles from the user study and the functional test cases collectively as “user profiles.”

#### 4.5 Generation and execution of test suites

The user profiles were processed in several stages to create replayable test cases. First, the identifiers for widgets were translated from those used by the Profiler to those used by JavaGUIReplayer. Next, the user profiles were distilled into a matrix of integers (Profiles in Figure 2) and a mapping from each integer in the matrix to the tex-

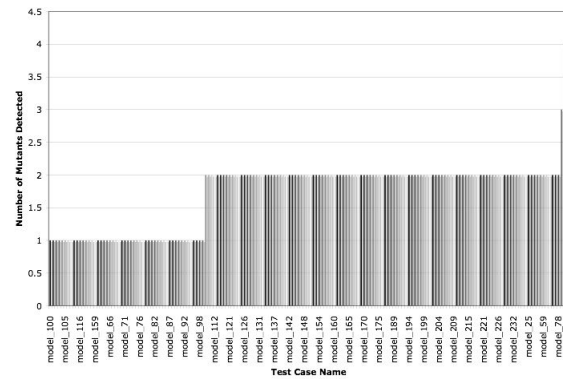


Figure 3. Number of mutants detected by each test case in PEFG

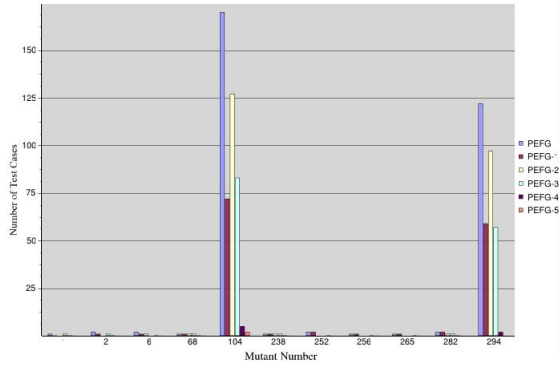
tual event identifier it represents. From this matrix, test cases were generated by a Matlab implementation of our test case generation algorithm with the `maxHistory` parameter set to 5. The original matrix encoded the test suite for the PROFILES technique, and the matrix output by Matlab (TestSuite in Figure 2) encoded the test suite for PEFG. Finally, using the mapping from integers to event identifiers, both matrices were expanded into a form JavaGUIReplayer could read.

For each test case, JavaGUIReplayer and OracleInfoVerifier were run on the golden version and each fault-seeded version using about 40 machines in a cluster of PCs running Linux. For reasons that are unclear and need to be investigated, JavaGUIReplayer was unable to replay a significant number of the test cases. In our results, we include only test cases that replayed successfully on the golden version.

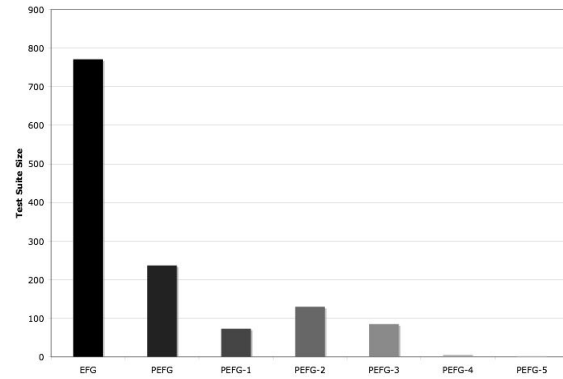
#### 4.6 Results

Of the 295 fault-seeded versions, the EFG test suite failed on (i.e. at least one test case failed on) 210 of them. The PEFG test suite failed on 11 fault-seeded versions, 3 of which the EFG test suite did not detect. Figure 3 depicts the results for PEFG by test case, showing that most of the test cases detected 1 or 2 mutants, with a few detecting 3 or 4. (One test case was excluded from this analysis because the results incorrectly indicate that this test case detected 293 of the faults.) Figure 4 breaks down the same results by fault-seeded (mutant) version. Due to a problem with the experiment’s infrastructure that requires further investigation, our results incorrectly indicate that the PROFILES test suite detected no faults.

The PROFILES technique uses fewer test cases (about 600) than EFG’s approximately 750 test cases. The PEFG



**Figure 4. Number of test cases in PEFG for various history lengths that detected each mutant. Mutants that were not detected by any test case in PEFG are omitted.**



**Figure 5. Number of test cases in each test suite**

technique further reduces the test suite size to approximately 200. Figure 5 illustrates this result.

Of the 11 test cases in the PEFG test suite, 6 are redundant in terms of faults detected. With optimal test case selection, the PEFG test suite could be reduced to 5 test cases without any loss of fault detection ability.

To analyze the effects of history length on test suite size and fault detection ability, we isolated the subset of the PEFG test cases that our algorithm generated based on a high-probability length- $(h+1)$  event interaction, for  $h$  from 1 to 5. Each subset, PEFG-1 through PEFG-5, consists of test cases that are generated when `maxHistory = h` but not necessarily when `maxHistory < h`. A test case may appear in more than one of these subsets.

Figure 5 shows that reducing the maximum history length from 5 to 3 reduces the test suite size very little, but Figure 4 shows that the corresponding reduction in fault detection ability is negligible as well. PEFG-1 detects 10 of the 11 faults that PEFG detects, and PEFG-1 and PEFG-3 together detect all 11 faults.

## 5 Conclusions and future work

Although the number of faults revealed by the PEFG test suite was much smaller than the number revealed by EFG, the results are promising from several perspectives. Of the 295 seeded faults, the PEFG technique detected 3 that EFG did not. If the user profiles collected reflect how real users interact with the GUI, then the faults that PEFG detected may be considered the most severe since they are the faults most likely to affect users. Hence, the PEFG technique can uncover potentially severe faults that the EFG technique misses. Using both techniques in tandem can detect more faults while only increasing the cost above EFG alone by a

small fraction (about  $1/4$  in the case of TerpWord 3.0).

To strengthen the claims made in this work, we will extend this study to include three more applications in the TerpOffice 3.0 suite. Our proposed study includes TerpPresent (an editor for presentation slides), TerpPaint (a drawing program), and TerpSpreadSheet (a spreadsheet editor). For these applications and TerpWord 3.0, we will obtain more complete results and try a wider range of values for the history parameter.

Additionally, we are investigating ways to improve the test case generation algorithm. Currently, the event sequences that the algorithm selects contain at least one highly probable  $n$ -way event interaction, but the probability of the event sequence in total may be very low. For example, if  $P(e_2|e_1) = 0.999$ , the algorithm will construct a test case that contains the sequence  $e_1, e_2$ , even if  $e_1$  is only exercised in 0.001% of the user profiles collected. We will modify our algorithm to explore different ways to compute the likelihood of a test case. Another possible variation of the algorithm is to traverse the least likely paths in the PEFG to reveal rarely-encountered faults that may otherwise be difficult to track down.

Our future work will also explore ways, in addition to varying the history length, to reduce the number of test cases our algorithm generates without sacrificing fault detection ability. We are investigating ways to prune test cases likely to be redundant based on information in the probabilistic EFG. Based on work by Weyuker [29], we will define coverage criteria, including possible adequacy/stopping criteria, for our technique.

## 6 Acknowledgments

We would like to thank our advisor, Atif Memon, for his help in developing the ideas of this paper and suggesting revisions. We would also like to thank Bin Gan, Xun Yuan, and Qing Xie for helping us use GUITAR. Finally, we would like to credit Xun Yuan for creating the fault-seeded versions of the subject application and obtaining the data for the EFG test suite.

## References

- [1] Abbot JavaGUITest Framework. <http://abbot.sourceforge.net>, 2003.
- [2] J. Berstel, S. C. Reghizzi, G. Roussel, and P. S. Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Trans. Softw. Eng. Methodol.*, 14(2):124–167, 2005.
- [3] J. M. Clarke. Automated test generation from a behavioral model. In *Proceedings of the Eleventh International Software Quality Week*. Software Research, Inc., May 1998.
- [4] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [5] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 139–150, New York, NY, USA, 2004. ACM Press.
- [6] D. Donovan, C. Dislis, R. Murphy, S. Unger, C. Kenneally, J. Young, and L. Sheehan. Incorporating software reliability engineering into the test process for an extensive GUI-Based network management system. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 44, Washington, DC, USA, 2001. IEEE Computer Society.
- [7] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.
- [8] M. Finsterwalder. Automating acceptance tests for GUI applications in an extreme programming environment. In *Proc. Second Intl Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pages 114–117, 2001.
- [9] M. L. Hammontree, J. J. Hendrickson, and B. W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 431–432, New York, NY, USA, 1992. ACM Press.
- [10] J. Hicinbothom and W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proc. Human Factors and Ergonomics Society 37th Ann. Meeting*, page 1042, 1993.
- [11] JUnit. Testing resources for extreme programming. <http://junit.org/news/extension/gui/index.htm>, 2004.
- [12] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice-Hall, 2000.
- [13] B. Marick. When should a test be automated? In *Proc. 11th Intl Software/Internet Quality Week*, 1998.
- [14] B. Marick. Bypassing the GUI. *Software Testing and Quality Engineering Magazine*, pages 41–47, 2002.
- [15] A. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Phd, Dept. of Computer Science, Univ. of Pittsburgh, July 2001.
- [16] A. Memon and Q. Xie. Using transient/persistent errors to develop automated test oracles for event-driven software. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 186–195, Washington, DC, USA, 2004. IEEE Computer Society.
- [17] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 30–39, New York, NY, USA, 2000. ACM Press.
- [18] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM Press.
- [19] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM Press.
- [20] 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.
- [21] Mercury Interactive WinRunner. <http://www.mercuryinteractive.com/products/winrunner>, 2003.
- [22] A. Nagarajan and A. M. Memon. Refactoring using event-based profiling, 2003.
- [23] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 82–92, New York, NY, USA, 1998. ACM Press.
- [24] S. Özekici, I. K. Altinel, and E. Angün. A general software testing model involving operational profiles. *Probab. Eng. Inf. Sci.*, 15(4):519–533, 2001.
- [25] Rational Robot. <http://www.rational.com.ar/tools/robot.html>, 2003.
- [26] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, New York, NY, USA, 2000. ACM Press.

- [27] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, New York, NY, USA, 2004. ACM Press.
- [28] TerpOffice 3.0. <http://www.cs.umd.edu/atif/newsite/terpoffice.htm>, 2004.
- [29] E. J. Weyuker. Using operational distributions to judge testing progress. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1118–1122, New York, NY, USA, 2003. ACM Press.
- [30] L. White, H. AlMezen, and N. Alzeidi. User-based testing of GUI sequences and their interactions. In *Proc. 12th Intl Symposium on Software Reliability Engineering*, pages 54–63, 2001.
- [31] J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.
- [32] D. Voit. Conditional-event usage testing. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 23. IBM Press, 1998.
- [33] D. M. Voit. Specifying operational profiles for modules. In *ISSTA*, pages 2–10, 1993.