Automated GUI Testing Guided By Usage Profiles

Penelope A. Brooks and Atif M. Memon
Department of Computer Science
University of Maryland
College Park, Maryland, USA
{penelope, atif}@cs.umd.edu

ABSTRACT

Most software developed in recent years has a graphical user interface (GUI). The only way for the end-user to interact with the software application is through the GUI. Hence, acceptance and system testing of the software requires GUI testing. This paper presents a new technique for testing of GUI applications. Information on the actual usage of the application, in the form of "usage profiles," is used to ensure that a new version of the application will function correctly. Usage profiles, sequences of events that end-users execute on a GUI, are used to develop a probabilistic usage model of the application. An algorithm uses the model to generate test cases that represent events the user is most likely to execute. Reverse engineering methods are used to extract the underlying structure of the application. An empirical study on four open source GUI applications reveals that test suites generated from the probabilistic model are 0.2-22% of the size of test suites produced directly from usage profiles. Furthermore, the test suites generated from the model detect more faults per test case than those detected directly from the usage profiles, and detect faults not detected by the original profiles.

Categories and Subject Descriptors: D.2 Software Engineering: Testing and Debugging, I.6 Simulation and Modeling: Model Development

General Terms: Verification

Keywords: usage profiles, GUI testing, event-driven software

1. INTRODUCTION

The most popular user interface for software today is the graphical user interface (GUI), providing user-friendly access to the functionality provided by the application. Testing GUIs, however, presents many challenges, primarily due to the immense number of possible permutations of commands that can be executed on the GUI. Testing all possible combinations and orderings of events[1] is not practical, as the number of test cases that would be required is exponential in the number of events. Instead, testers of GUI applications attempt to limit the number of test cases that need to be executed. Due to the event-driven nature of GUI software, GUI testing requires different techniques than those used in conventional testing [13].

Although it seems the code that makes up the interface to the application would not be very substantial, it generally makes up 45-60% of the overall code and can house faults [15]. Errors in the GUI can manifest themselves as failures of the overall software. Therefore, it is important to test the GUI.

This paper presents a technique that can be used for *regression testing* of GUI applications, *i.e.*, information gathered from usage of the current version of a GUI application is used to determine whether the new application will perform as expected. Previous work on GUI regression testing showed that test suites covering all feasible pairs of events, called *smoke tests*, detect a large number of faults [14]. Many of these event sequences are unlikely to occur in actual usage of the software. For instance, the event sequence Save followed by Save As is a valid test case in the smoke test suite (represented as the ordered pair (Save, SaveAs)), but is not very likely to occur in actual usage of the software. Conversely, the event sequence that represents formatting a document (for instance, adding tables and figures, or changing font size, style, and color) is very likely, but will not be contained in the smoke test suite since formatting requires a sequence of more than two events, and smoke tests are restricted to sequences of two events. In previous work, we have shown that event sequences longer than two events reveal many faults not detected by the smoke test suite [14]. Including these test cases in the smoke test suite, however, is not a viable solution since increasing the test suite from all possible pairs of events to all possible triples, ..., n-tuples causes it to grow exponentially [14].

To counteract this exponential test suite growth, another approach, and the one used in this paper, is to automatically generate test cases based on usage information, in the form of "usage profiles," collected from end-users interacting with a fielded version of the application. Usage profiles (called operational profiles [24] or session data [6] in some literature)

[1]An event is a user action that can be performed on a GUI widget, such as clicking on a button or a menu item, or typing in a text field. In the remainder of this paper, wherever possible, events will be denoted by their corresponding widget; for example, click_on_Cancel_button will be called Cancel and click_on_menu_item_Save will be called Save.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

can be obtained by capturing event sequences users execute on the GUI application. Because they are gathered from actual usage of the software, any faults detected by using them represent faults likely to be encountered by users in the field. After collection, profiles can be used in two ways: they can be replayed directly or used to create a model of the application which can be used for testing. Using an automatic test case replayer to replay the profiles directly on a modified version of the GUI application is the most straightforward approach [16]. Due to possible structural changes to the GUI, this is not a good approach for regression testing, however, since as many as 75% of the event sequences may no longer execute on the modified application [13]. The second approach, and the focus of this paper, is to employ the usage profiles to create and update an abstract model of the GUI application and use the model to generate test cases automatically.

In this paper, a new technique is presented to capture GUI user profiles in a transparent manner and with very little overhead, employ the profiles to populate a probabilistic model of the GUI application, and generate test cases from this model. An empirical study was conducted on four open source applications to measure the benefit user profiles add to the regression testing process. After collecting user profiles, the probabilistic model was populated and used to generate test cases corresponding to the concatenation of many short, highly likely event subsequences. Test cases generated by the model were ordered by the likelihood of occurring in actual use; therefore, the test suite was composed of test cases that were most indicative of actual use of the application. The challenge of collecting profiles was mitigated by using a profiling tool that runs in the background, with very little impact on performance of the application. Two test suites were compared: one composed of test cases replayed directly from a set of collected user profiles and one composed of test cases generated from our model (populated with the same set of usage profiles). The results show that the test suites generated from our model detected more faults per test case than replaying the profiles directly, and with a fraction of the number of test cases in the original profiles suite, thereby reducing the cost of testing. Additionally, the test cases generated from the model detect faults that are not detected by replaying the original profiles; these are faults that are likely to be discovered by users of the application.

The work presented in this paper provides the following primary contributions:

1. a new probabilistic finite state model representation of a GUI application that combines its static window/widget structure with actual usage,
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, and
4. an empirical investigation comparing the size and fault detection ability of a test suite generated by our technique to a test suite of "raw" usage profiles on four large, fielded applications.

The remainder of this paper is organized as follows. Section 2 expands on work in related areas, such as using state models in testing and usage profile-based testing. The details of the model used in this paper are elucidated in Section 3. Section 4 contains a description of the empirical studies and the results are presented in Section 5. Finally, conclusions and future work are described in Section 6.

2. RELATED WORK

Previous researchers have studied the effectiveness of collecting information from users, either at runtime or during software development, to be used in software testing. Our previous work investigated replaying usage profiles to perform regression testing on applications, but did not employ usage profiles to create a model of the application and test the application based on that model [12]. Many of the previous approaches to testing with usage profiles rely on manually generating profiles based on *a priori* knowledge of the application [23, 24]. The overall goal of testing using these methods is to develop and implement fully automated methods in which user interactions are captured, the application or application model is generated automatically from the binary code, and test cases are developed and executed.

Usage profile-based testing transitions nicely to testing GUI applications. Due to the large number of events in a GUI, user profiles can assist a tester in shrinking the scope of testing by focusing test case generation on the parts of the GUI and sequences of events most often used by actual users. In regression testing, user profiles are collected on the fielded version and used to generate test cases for the version under test. Collecting profiles on a fielded version of the software enables testers to accurately represent tasks users will execute, leading to more effective test cases.

2.1 Usage profile-based testing

Many researchers in software testing have used capture and replay tools to test software, often in conjunction with test scripts created by testers to ensure the software functions properly. The same capture/replay tools can also be used to record usage information, in the form of event sequences executed on the application, from actual users [4, 16, 18]. The information collected can then be integrated into test cases or used to generate test cases. Although this approach is not widespread, it is similar to the approach used in this work.

Operational profiles, generally gathered through ethnographic methods (customer interviews, videotaping, usage logs), can also be used to develop test cases. These profiles can be used to identify critical failures in a GUI application; be used to partition users into classes of users and develop test cases accordingly; be annotated based on expected or actual usage of the application; and be used to determine testing coverage, updating the profile to reflect success or failure [1, 5, 17, 20].

Usage data (or "session data") has also been used to test web applications, by generating test cases from a combination of events from different users' session data, creating a new virtual user [6]. While the approach presented in this paper uses a similar idea, it is applied to GUI applications rather than web applications, and a probabilistic model is populated and used to generate test cases, rather than generating test cases based solely on the user-session data.

2.2 GUI testing

There are a variety of popular methods used for testing

GUIs, including test harnesses, capture/replay tools, and model-based methods. Test harnesses invoke methods in the underlying business logic of a application, as if executed by the GUI, without actually using the GUI [10]. While test harnesses effectively isolate the behavior of the GUI, they do not test the interface code itself and, therefore, are not applicable to the approach presented here.

The second approach, using manual tools to mimic the usage of the GUI, only provides limited testing [7, 21]. JUnit, an open-source, regression testing framework used to test Java code, has been extended into JFCUnit and Jemmy Module for use in GUI testing [8]. Several capture/replay tools have also been created based on JUnit, including Abbot, Jacareto, and Pounder [8]. These tools mimic actual usage of a GUI by recording interactions with the GUI in *capture* mode and replaying the interactions in *replay* mode. A common approach of these tools is to store mouse coordinates, causing test cases to be fragile and dependent on the GUI layout, and rendering many test cases ineffective for regression testing since the GUI layout may change between versions. Tools such as Winrunner[2], Abbot[3], and Rational Robot[4] avoid this problem by capturing GUI widgets rather than mouse coordinates. These tools, however, require a significant amount of manual effort to be effective, including developing test scripts and manually detecting failures. Modifications to the GUI require changes to the scripts as well. Testers who employ these tools typically come up with a small number of tests to utilize during development [11].

Finally, test cases can be generated based on a model of the GUI. Previous research has shown the usefulness of graphs and state machine models to represent the GUI, and test cases were developed by exploring paths through the model [2, 3, 14, 22]. Graph models can be walked, either randomly or deterministically, but can not be traversed exhaustively because of the large number of possible traversals. Therefore, it is common to traverse all paths of the graph up to a specified length [14].

State machines are a well-known way to model complex software. Behavioral models, in which nodes represent program states or sets of related states and edges represent transitions between states, are one type of state-machine model which encodes the behavior of the program. Each test case is generated by traversing the model, representing the path traversed [2].

Operational models, a type of state machine, use nodes to represent user operations. Nodes, edges, or paths can be weighted with the probability that a user traverses them. Previous research has explored using Markov models in which the probability on each node depends only on the previous node [22] and using a Bayesian framework in which model parameters can be learned and updated during testing [17].

Representing a GUI with a state machine or graph model allows a wide variety of algorithms to be used for test case generation, but many of the current approaches are not based on actual usage of the software. The method presented in this paper uses a graph to model the GUI and augments it with probabilities based on observed usage.

[2]Mercury Winrunner,
http://www.mercuryinteractive.com/products/winrunner
[3]Abbot JavaGUITest Framework,
http://abbot.sourceforge.net
[4]Rational Robot,
http://www.rational.com.ar/tools/robot.html

3. PROBABILISTIC MODEL OF GUI INTERACTIONS

In developing testing strategies for GUIs, it is desirable to develop a multi-purpose model that can be used to represent all possible GUIs. Due to the variety of applications with a GUI front-end, however, such a model would be required to represent a vast number of GUIs. Therefore, we represent one sub-class of GUIs: those which take input from a single user, have a fixed number of events, and are deterministic. This class of GUIs can be represented by an *Event-Flow Graph* (EFG) and standard graph walking techniques can be used to reason about the model and generate test cases [14].

In developing a software model to be used in testing, it is important that it be reusable and flexible [19]. The model described here can be used for an application in any domain, given information about sequences of events for that application, rendering our model very flexible and applicable to any GUI application in the aforementioned subclass. In modeling expected usage, all users are treated as "average" users. Therefore, although each application under test can potentially have many usage models (one for each class of users), one usage model was developed per subject application for the purposes of this work. Additionally, the model presented here gives a tester the ability to represent the behavior of a new user, formed by combining the most likely event sequences observed in the pool of users.

3.1 Modeling GUI structure with an Event Flow Graph

An EFG is a specific model of the GUI for a particular application, representing all possible sequences of events that a user can execute on that GUI. Nodes in the EFG represent events, and directed edges represent the event-flow relationship between two events. An edge in the graph between events e_1 and e_2 indicates that event e_2 may be invoked *immediately after* event e_1. The predicate $follows(e_2, e_1)$ represents this relationship and denotes that e_2 *follows* e_1. EFGs are potentially cyclic, since events can typically be executed more than once during a session with an application.

For instance, a user looking for a word in a Microsoft Word document can select the Edit menu followed by the Find menu item, type the word they wish to find in the text box, and select the Find button to perform the find operation. The user can then find the next occurrence of the word by selecting Find again before selecting Cancel to close the dialog box. Each of these events is represented by a node in the EFG. Figure 1(a) shows a screenshot of the Microsoft Word GUI. For increased readability, only a few menu items for the File and Edit menus are shown. Each item (or event) in the menu is a node in the event flow graph, shown in Figure 1(b). Three types of events are shown in Figure 1(b); menu items in the top level menu bar are shown in boxes, menu items from drop down menus are shown in diamonds, and buttons in the Find/Replace window are shown in circles. The event sequence described above is indicated by bold lines. Because edges in an EFG are directed, the EFG in Figure 1(b) shows the order which must be maintained in executing events on the GUI.

3.2 Developing a probabilistic EFG

In the work presented here, a subgraph of the EFG is produced based on observed event sequences collected in usage

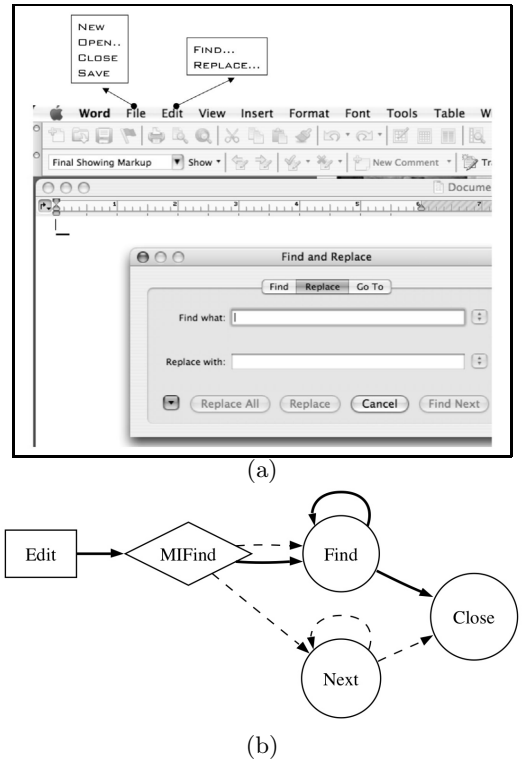

Figure 1: Example using Microsoft Word: (a) Microsoft Word GUI; (b) Example Event Flow Graph for a subset of Microsoft Word

profiles. Starting with the full EFG for the application under test, nodes and edges not observed in the usage profiles are eliminated, resulting in a smaller EFG. Next, the tailored EFG is augmented with two special nodes, $INIT$ and $FINAL$, and two sets of events, the initial and final events, are populated. The set of initial nodes contains the first event of each usage profile, and the set of final nodes contains the last event in each user profile. These initial and final node sets represent all possible first and last events, respectively, in the observed user interactions with the GUI. Transitions are added from $INIT$ to each initial node and from each final node to $FINAL$, constraining the EFG to one entry point and one exit point. This property makes the EFG conform to the rules of a Markov chain and allows the model to take advantage of the Markov assumption [9]. Using the Markov assumption of event independence and frequencies observed in the usage profiles, the EFG's nodes are weighted in preparation for generating test cases.

Our model contains a collection of R paths through the EFG called $r_1, r_2, \ldots, r_R$. Each path r_i where $1 \leq i \leq R$, consists of a sequence of n events in addition to $INIT$ and $FINAL$:

$$r_i = INIT, x_1, x_2, \ldots, x_n, FINAL;$$

$$\forall j \; e_j \in \{e_1, e_2, \ldots, e_{n-1}\} \wedge$$

$$follows(e_{j+1}, e_j)$$

where $x_1, x_2, \ldots, x_n$ and $e_1, e_2, \ldots, e_{n-1}$ are events in the EFG, r_i denotes a path, and each path r_i contains only events with a *follows* relationship between them. Valid

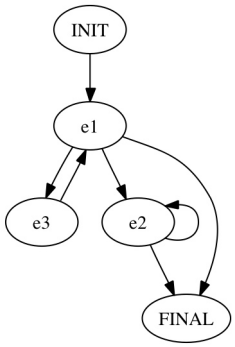

Figure 2: Example Event Flow Graph

paths can also be formed by the concatenation of two paths, *e.g.*, r_a and r_b, provided the first event of r_b *follows* the last event of r_a in the EFG.

Consider the following usage profiles, *i.e.*, a set of paths through the EFG, shown in Figure 2:

$$\begin{aligned} r_1 &= INIT, e_1, e_2, FINAL \\ r_2 &= INIT, e_1, e_3, e_1, e_2, FINAL \\ r_3 &= INIT, e_1, e_2, e_2, e_2, FINAL \end{aligned}$$

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

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

Following the example given above, the counts and probabilities for each event are:

$$\begin{aligned} count(e_1) &= 4 \; P(e_1) = 0.4 \\ count(e_2) &= 5 \; P(e_2) = 0.5 \\ count(e_3) &= 1 \; P(e_3) = 0.1 \end{aligned}$$

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

$$s_i = x_1, x_2, \ldots, x_S$$

$$\forall j \; e_j \in \{INIT, e_1, e_2, \ldots, e_{n-1}, FINAL\} \wedge$$

$$follows(e_{j+1}, e_j).$$

For a subsequence length of 2 (*i.e.*, $S = 2$), some valid subsequences for the example are:

$$\begin{aligned} s_1 &= e_3, e_1 \\ s_2 &= e_1, e_2 \\ s_3 &= e_2, e_2 \end{aligned}$$

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

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

where $count(s)$ returns the number of times s occurs as a subsequence of $r_1, r_2, \ldots, r_R$ and $subs(S)$ is the set of all length-S subsequences in $r_1, r_2, \ldots, r_R$. For each subsequence given above, the count and probability are:

$$count(s_1) = 1 \; P(s_1) = 0.17$$
$$count(s_2) = 3 \; P(s_2) = 0.50$$
$$count(s_3) = 2 \; P(s_3) = 0.33$$

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

$$P(e_i|s) = \frac{P(s_1, s_2, \ldots, s_S, e_i)}{\Sigma_{j=1}^{E} P(s_1, s_2, \ldots, s_S, e_j)}.$$

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 $P(e_i|INIT)$, which is the probability that event e_i is the first event in the sequence, occurring immediately after $INIT$. Rather, $P(e_i|s)$ is the probability of e_i given *no information* about the events that precede it.

A *probabilistic EFG* (PEFG) is created by annotating each event (node) in the EFG with a table containing the event's prior probability and its probability conditioned on each subsequence in $\{r_1, r_2, \ldots, r_R\}$ up to some maximum subsequence length, or history, H. Each set of entries for all length-h subsequences, $0 \leq h \leq H$, succinctly encodes a probabilistic Markov model whose $O(E^h)$ nodes correspond to length-h subsequences and whose edges are labeled with conditional probabilities. Table 1 shows a conditional probability table of event e_1 for subsequences of events up to length 5 for the EFG shown in Figure 2. Zero-probability entries are omitted. The first data row in the table gives the prior probability of e_1.

h	$P(e_1\|h)$
-	0.3125
$INIT$	1.0
e_2	0.25
e_3	1.0
e_1, e_3	1.0
e_2, e_2	0.5
$INIT, e_1, e_3$	1.0
e_2, e_2, e_2	1.0
e_1, e_2, e_2, e_2	1.0
$INIT, e_1, e_2, e_2, e_2$	1.0

Table 1: Probabilities relating to Figure 2

3.3 Generating test cases

In general, there are two ways to use a probabilistic test case generation model. Test cases can be generated that exercise the most probable sequence of events, given a range of starting points and test case lengths. Alternatively, test cases (*i.e.*, event sequences) can be generated based on concatenating pairs of events that have the highest probability of occurring. For the study presented in this paper, the latter method is used, in order to determine the effectiveness of highly probable pairs of events. All possible test cases are generated, based on the events observed during usage profile collection, up to a specified history provided by the tester.

The algorithm in Figure 3 generates test cases by constructing and traversing the probabilistic EFG. In the pseudocode shown, sets of event sequences, such as the input set

```
Input: Profiles, maxHistory

# Step 1:  Construct Subs
for each row Row of Profiles
   for m from 1 to maxHistory
      for each unique length-m subseq Seq of Row
         Subs := Subs unioned with {Seq}
      end
   end
end

# Step 2:  Construct Distributions
for each row index i of Subs
   m := length of Subs(i)
   numTotal := number of length-m subseq in Profiles
   for each unique element e in Profiles
      Seq := row i of Subs concatenated with e
      numSeq := number of occurrences of Seq in Profiles
      Distributions(e, i) := numSeq / numTotal
   end
end

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

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

Output: TestSuite
```

Figure 3: Test case generation algorithm

of usage profiles and the output set of test cases, are stored as matrices in which each row holds an event sequence and the ith column of a row holds the ith event in the sequence. The algorithm takes two parameters: `Profiles`, the set of usage 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 `Subs` matrix is constructed by adding in each unique event subsequence up to length `maxHistory` in `Profiles`. In Step 2, the probability distribution $P(e|i)$ is computed for each unique event `e` in `Profiles` and each row `i` in `Subs` and is stored in `Distributions`. For each unique event `e`, Step 3 calculates the probability of each path from `INIT` to `e`. Finally, in Step 4, the matrix `TestSuite` is constructed by adding a legal test case (i.e. one that begins with `INIT`) for each column maximum in `Distributions`.

4. EMPIRICAL STUDY

We conducted an empirical study to evaluate the effectiveness of our model-based test case generation technique (PEFG) in relation to replaying the usage profiles "as-is" (PROFILES), without using the probabilistic EFG. The study answered the following questions:

1. Is the PEFG test suite more effective at fault detection than the PROFILES test suite?

2. Is the cost of testing using the PEFG technique less than that of the PROFILES technique?

3. Does reducing the history parameter in the PEFG technique (down to some minimum length) decrease the cost of testing without decreasing the effectiveness of the test suite?

Under the assumption that the usage profiles used to populate the probabilistic EFG are representative of usage in the field, the first question implies that the PEFG technique is more effective at finding faults than replaying the usage profiles directly. In order to compare the results of test suites of different sizes, test suite effectiveness was normalized by computing *fault density*, that is, the number of faults detected by a test suite, divided by the number of test cases in that test suite. More formally,

$$\left| \frac{F(\text{PEFG}, h)}{|\text{PEFG}|} \right| > \left| \frac{F(\text{PROFILES})}{|\text{PROFILES}|} \right| \quad (1)$$

where $F(suite, history)$ is the number of faults detected by *suite* with a history value of *history*[5]. Test suite effectiveness was determined by running each test case on each subject application. Failed test cases are those that caused the application to encounter an uncaught exception during test case execution. After collecting the set of failures, each was manually linked to the fault that caused it. This approach has been used before and has been found to be useful [25,26].

The second question relates to the cost of generating and executing test cases, and proposes that the PEFG technique will be cheaper than executing the profiles directly as test cases, or

$$c(\text{PROFILES}) > c(\text{PEFG}, h), \quad (2)$$

where $c(suite, history)$ is the cost of testing that *suite* with a history value of *history*. As a proxy for cost, the size of the test suite was used. In the context of this study, using the size of the test suite rather than the length of the test cases is reasonable since the overhead required to run each test case, including starting the test case execution framework, starting the application, and ending the application and framework, consumed more computation time than any other activity. The goal of developing a cost value for each test suite is to use it as a comparison measure. Computing cost based on the size of the test suite allows valid comparison of our method to work performed by others using this suite of tools, since the cost of executing test cases will be the same.

The third question asserts that, if the minimum sufficient history length is known or estimated for some application, the cost of testing the application can be reduced without loss of fault detection ability, determined by the following:

$$\begin{gathered} F(\text{PEFG}, h) \approx F(\text{PEFG}, h-n) \wedge \\ c(\text{PEFG}, h) > c(\text{PEFG}, h-n) \end{gathered} \quad (3)$$

where $h-n$ represents the minimum sufficient history length.

4.1 Infrastructure

The GUI Testing FrAmewoRk (GUITAR) was used to perform the study [14]. The JavaGUIRipper, one of the tools in the GUITAR suite, was used to glean the structure of the subject applications. By using Java Reflection, the JavaGUIRipper creates an XML file that represents the windows, menu items, and buttons present in the GUI, including the actions that are executed when those items are selected.

Another tool in the suite, the JavaGUIReplayer, was used for test case execution. The JavaGUIReplayer is a framework that opens the application under test and replays test cases. Each event is executed on the GUI, and the state of the GUI is recorded after each step. The state is saved in XML files that can be examined to determine which test cases failed and why.

Usage profiles can be captured by a tool in GUITAR's family of applications called the Profiler [16]. (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.

4.2 Subject applications

Four popular open source applications developed in Java Swing were chosen for this study and downloaded from SourceForge:

1. CrosswordSage 0.3.5[6], a popular tool for creating and solving professional-looking crossword puzzles with built-in word suggestion capabilities, with an all-time activity rate of 76.87%.

2. FreeMind 0.8.0[7], a very popular mind-mapping application, with an all-time activity rate of 99.96%.

3. GanttProject 2.0.1[8], a project scheduling application featuring Gantt chart, resource management, calendars, and the option to import/export MS Project, HTML, PDF, and spreadsheets, with an all-time activity rate of 99.85%.

4. jMSN 0.9.9b2[9], a clone of MSN Messenger, including instant messaging, file sharing, and additional chat features standard in MSN Messenger, with an all-time activity rate of 93.81%.

These applications were chosen for several reasons. First, they all have an active developer community and high all-time-activity scores on SourceForge, with three of the applications above 90%. CrosswordSage was chosen partially because it is fairly new (first released in 2005) and yet has an activity score of almost 80%. These applications have also been released in several versions and have undergone quality assurance prior to each release.

Preparation for testing the applications included setting up a database for text-field values. In order to automate test case replaying, a database that contains one instance for each of the text types in the set {*negative number, real number, long file name, empty string, special characters, zero, existing file name, non-existent file name*} was used. Note that if a text field is encountered in the GUI (represented as

[5] *History* only applies to the PEFG test suite, given as a parameter to the model

[6] http://sourceforge.net/projects/crosswordsage
[7] http://sourceforge.net/projects/freemind
[8] http://sourceforge.net/projects/ganttproject
[9] http://sourceforge.net/projects/jmsn

an event called `type-in-text`), one instance for each text type is tried in succession. We also set up the test oracle to detect crashes for these applications, where a crash is defined as an uncaught exception thrown during test case execution.

4.3 File formats and examples

Usage profiles and test cases are both stored in an XML format understood by the JavaGUIReplayer. An example of one usage profile, the corresponding entries in the map and matrix files, and a test case resulting from the model are shown below. For the sake of space, only a few attributes are shown, and only on the first step. In a full profile or test case, there are 16 attributes for each step.

Figure 4 shows a partial usage profile for GanttProject. This profile is composed of six steps, which together make a new GanttProject, and set some of the project specifics in the "Create new project" window. Finally, Cancel is selected, which will cancel all of the user actions for creating the new project.

The following lines, each representing one usage profile in the "matrix" file to be used by the test case generation algorithm, represent a usage profile in an abbreviated format; each number represents one event in the profile.

```
1 2 3 4 7 9
1 2 10 3 12 7
1 2 3 12 5 7
```

The "map" file is the translator between the profile and the matrix. The portion of the map file shown here describes the events in the sample usage profile and one line of the matrix.

```
1 <Window>GanttProject_1</Window><Component>New..._R_33
          </Component><Action>doClick</Action>
2 <Window>Create new project_2</Window><Component>AutoText_R_0
          </Component><Action>setText_String_5</Action>
3 <Window>Create new project_2</Window><Component>AutoText_R_1
          </Component><Action>setText_String_5</Action>
4 <Window>Create new project_2</Window><Component>AutoText_R_9
          </Component><Action>setText_String_5</Action>
7 <Window>Create new project_2</Window><Component>Cancel_R_23
          </Component><Action>doClick</Action>
9 <Window>Create new project_2</Window><Component>AutoText_R_2
          </Component><Action>setText_String_5</Action>
```

After running the test case generation algorithm (Section 3), a map and matrix were produced for the generated test suite. The resulting matrix file was expanded into test case events, based on the numbers assigned in the mapping, and test cases were generated. The following line is an example of a test case generated by the algorithm; this exact sequence of events did not occur in the set of profiles.

```
1 2 10 3 12 7 1
```

An example of a generated test case is shown in Figure 5.

4.4 Generation and execution of test suites

The usage profiles were processed in several stages to create replayable test cases. First, the usage profiles were distilled into a matrix of integers (`Profiles` in Figure 3) and a mapping from each integer in the matrix to the textual event identifier it represents. From this matrix, test cases were generated by a Matlab implementation of our test case generation algorithm.

Based on previous research which showed that event sequences of length 3 reveal substantially more faults than

```
<Profile>
      <Step>
            <Window>GanttProject_1</Window>
            <Component>New..._R_33</Component>
            <Action>doClick</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Attributes>
                  <Property>
                        <Name>IconImage</Name>
                        <Value>IconImage_24</Value>
                  </Property>
                  <Property>
                        <Name>Type</Name>
                        <Value>RESTRICTED</Value>
                  </Property>
                  <Property>
                        <Name>ReplayableAction</Name>
                        <Value>doClick</Value>
                  </Property>
                  <Property>
                        <Name>Visible</Name>
                        <Value>TRUE</Value>
                  </Property>
                  <Property>
                        <Name>Enabled</Name>
                        <Value>TRUE</Value>
                  </Property>
            </Attributes>
      </Step>
      <Step>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_0</Component>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
      </Step>
      <Step>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_2</Component>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
      </Step>
      <Step>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_1</Component>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
      </Step>
      <Step>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_9</Component>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
      </Step>
      <Step>
            <Window>Create new project_2</Window>
            <Component>Cancel_R_23</Component>
            <Action>doClick</Action>
            <WindowFlag>FALSE</WindowFlag>
      </Step>
</Profile>
```

Figure 4: Partial usage profile for GanttProject

event sequences of length 1 or 2, `maxLength` was set to 5 for this study [14]. Fault detection effectiveness for each sequence length up to 5 is examined, to determine the optimal level. In some applications, a sequence length of 5 required roughly twice as much time for test case generation and processing.

The original matrix, such as that shown above, encoded the test suite for the PROFILES technique, and the matrix output by Matlab (`TestSuite` in Figure 3) encoded the test suite for PEFG. Finally, using the mapping from integers to event identifiers, both matrices were expanded into XML test cases for JavaGUIReplayer.

Each test case was run using the JavaGUIReplayer, us-

```
<Testcase>
      <Step>
            <Action>doClick</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>GanttProject_1</Window>
            <Component>New..._R_33</Component>
            <Attributes>
                  <Property>
                        <Name>Title</Name>
                        <Value>New..._R_33</Value>
                  </Property>
                  <Property>
                        <Name>Class</Name>
                        <Value>javax.swing.JMenu$1</Value>
                  </Property>
                  <Property>
                        <Name>Enabled</Name>
                        <Value>TRUE</Value>
                  </Property>
                  <Property>
                        <Name>Visible</Name>
                        <Value>TRUE</Value>
                   </Property>
                   <Property>
                        <Name>Type</Name>
                        <Value>RESTRICTED</Value>
                   </Property>
            </Attributes>
      </Step>
      <Step>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_0</Component>
      </Step>
      <Step>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_2</Component>
      </Step>
      <Step>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_9</Component>
      </Step>
     <Step>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_4</Component>
      </Step>
      <Step>
            <Action>setText_String_5</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>Create new project_2</Window>
            <Component>AutoText_R_1</Component>
      </Step>
      <Step>
            <Action>doClick</Action>
            <WindowFlag>FALSE</WindowFlag>
            <Window>GanttProject_1</Window>
            <Component>New..._R_33</Component>
      </Step>
</Testcase>
```

Figure 5: Partial generated test case

ing one machine per application in a cluster of PCs running Linux. The overall process executed in approximately 4 hours per application. Pre-processing the usage profiles for the model took approximately 5 minutes, model creation took approximately one hour, and test case generation took about 30 minutes. Finally, test case execution took 2-3 hours. There is no interaction between the test cases, meaning the results of one do not influence the results of another.

4.5 Threats to Validity

The results of this study should be interpreted with some deference to threats to validity. First, due to our desire to use the existing GUITAR infrastructure, and to compare our results to those posted by previous graduate student researchers, we used subject applications developed in Java. Therefore, we have no information on how the results would translate to other development languages. Also, although the GUI for each application is different, they do not reflect all possible classes of GUIs. Furthermore, the majority of the application code is written for the GUI, meaning the results may not be consistent for applications with a simple GUI and complex underlying business logic. Second, although the applications chosen for this study have undergone quality assurance, they are open-source and developed by volunteers, leading to the possibility that they are more prone to bugs than professionally developed software.

5. RESULTS

The fault density of the PEFG test suite was better than the PROFILES test suite on all four applications. The test suites produced by the model, referred to as the PEFG test suites, were consistently smaller than the original PROFILES test suites, reducing the time and computation resources required for producing and executing the test suites. Table 2 shows the difference in the test suite sizes. Finally, for the set of usage profiles used as input, the most effective history length was 5.

5.1 Fault Detection

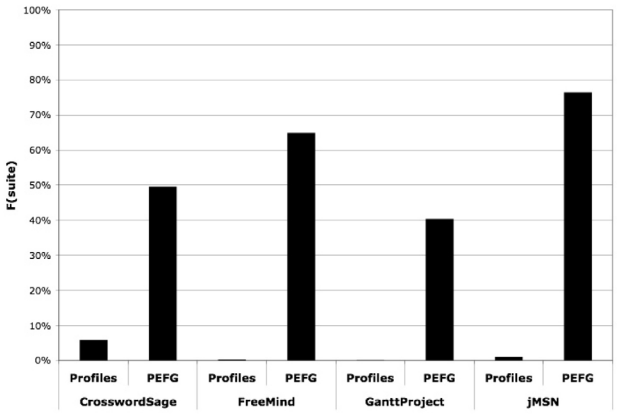

Figure 6: Fault density

In order to determine if the PEFG test suite detects more faults per test case than the PROFILES test suite, Equation 1 was computed, shown in Figure 6. The PEFG test suite for CrosswordSage produced 123 faults, compared to 111 faults detected by the PROFILES test suite. In all four applications, the fault density of the PEFG test suite was better than that of the PROFILES test suite. FreeMind's PEFG test suite detected 111 faults, while the PROFILES test suite detected 200. jMSN's PEFG and PROFILES test suites detected 1135 and 76 faults, respectively. There was a rather large difference in test suite size for GanttProject (Table 2), which greatly impacted the fault density for the test suites (Figure 6). The PEFG test suite found 308 faults, while the PROFILES test suite found 258.

Additionally, the PEFG test suite found faults not found in the PROFILES test suite in two applications. The PEFG test suite found 5 faults in CrosswordSage and 2 faults in jMSN that were not detected by the PROFILES suite. The faults are uncaught exceptions related to incorrect state of the application, resulting in an error of "Wrong index for this event."

5.2 Cost

Application	PROFILES	PEFG
CrosswordSage	1903	248
FreeMind	58301	171
GanttProject	199139	762
jMSN	6852	1484

Table 2: PROFILES and PEFG test suite sizes for each application

The cost of executing each test suite, *i.e.*, the size, was computed using Equation 2. Comparing these costs, as shown in Table 2, it can be seen that the PEFG test suites are consistently smaller than those of the PROFILES test suites. Therefore, less resources are required to generate and execute the test suites.

5.3 History

Using Equation 3, the fault detection and cost of each history level can be compared, as shown in Figure 7. It is important to note that for any history, h, all prior histories are also used, so the faults detected using a history of n include $\sum_{h=1}^{n} F(suite, n)$. Using a history of 5 in preparing the PEFG test cases produced test cases that detect 100% of faults.

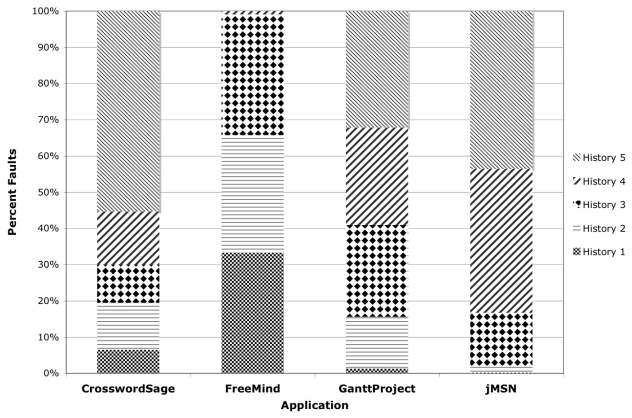

Figure 7: Faults for varying levels of history

5.4 Discussion

The fault detection effectiveness of the PEFG test suite shows the validity of the model-based test case generation technique presented here. Although the model can be improved to detect more faults, the test cases do detect faults not detected by using the original profiles as test cases. One of the motivating factors of this research is to detect faults that will be discovered by users and are not detected by other approaches, which has been shown to some degree in this work.

By generating the PEFG test suite and drastically decreasing the size of the test suite – roughly 0.2%-22% of the size of the PROFILES test suites – the cost of testing has been reduced. This can be partially explained by the fact that many users will execute not only the same events, but also very similar event sequences. The PEFG test suite would then be much smaller because the use of probabilities effectively collapses the sequences and gives more weight to the common sequences.

The best results are obtained by using a history value of 5, yielding 100% fault detection. We initially hypothesized that a history value of 3 would provide sufficient fault detection, but this is not the case in these test suites, which may be due to the length of the usage profiles. In general, longer usage profiles, much like longer test cases, will train the model more effectively and thus generate better test cases. The usage profiles in this study contain, on average, 9 events, which is relatively short. Ideally, profiles of 20 events or more are desirable to provide a more accurate insight into actual usage of the application.

6. CONCLUSIONS AND FUTURE WORK

The model presented in this paper was demonstrated in the context of automated regression testing of GUI applications. The foundation of the model is collecting usage information from real users of the application, and applying that information to test case generation. Well-known probabilistic methods were used to develop a model, which was in turn used to generate test cases. Processing the collected profiles through the model produced smaller test suites with better fault detection per test case than the PROFILES test suites.

User-profile based testing naturally lends itself to applications in regression testing. In order to avoid the complexity of unexecutable test cases, regression testing of multiple versions of the open source applications was not demonstrated in this study. Collecting usage profiles on one version and using them to populate a model of a subsequent version will require some extensions to the model in order to map from one EFG to the other, and is a planned extension of this research. Based on work by Weyuker [20], we will also define coverage criteria for this technique, which will help to identify new parts of the application that are not covered by any of the existing usage profiles.

In the future, the test case generation algorithm can be improved. Currently, the algorithm selects event sequences that contain at least one highly probable n-tuple of events, but the probability of the whole event sequence, and therefore the test case, 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 usage profiles collected. We will modify our algorithm to use techniques which consider the probability of a whole sequence of events in generating a test case. Another 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 detect.

Many of the generated test cases for CrosswordSage did not complete because of one event they had in common: the Save event. The test cases failed when the Save event was replayed, due to the event not being enabled at that point

in the test case execution, because the test cases did not first execute Open or New. The current test case generation algorithm does not have any notion of dependency, and does not enforce business logic such as "*Execute an* Open *or* New *before a* Save." This will be addressed in future versions of the algorithm.

Finally, training the model to detect different classes of users, and generating test cases based on these classes, will provide another level of insight into the usage of the application. Because the process is automated, developing test suites based on the user class will not be more difficult, and could uncover faults that would not be detected in the single user class model presented here.

Acknowledgments

We would like to thank the GUITAR group for their help in creating many of the file manipulation tools used in generating test cases, assistance with the GUITAR suite, and in producing results for the PROFILES test suites. 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] 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.

[2] J. M. Clarke. Automated test generation from a behavioral model. In *Proc. of the Eleventh Int'l Software Quality Week*, May 1998.

[3] 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 *Proc. of the 21st Int'l Conf on Software engineering*, pages 285–294. IEEE Computer Society Press, 1999.

[4] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proc. of the 4th Int'l workshop on Software and performance*, pages 139–150. ACM Press, 2004.

[5] 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 *Proc. of the 12th Int'l Symposium on Software Reliability Engineering*, page 44. IEEE Computer Society, 2001.

[6] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. of the 25th Int'l Conf on Software Engineering*, pages 49–59. IEEE Computer Society, 2003.

[7] M. Finsterwalder. Automating acceptance tests for GUI applications in an extreme programming environment. In *Proc. Second Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pages 114–117, 2001.

[8] JUnit. Testing resources for extreme programming. http://junit.org/news/extension/gui/index.htm, 2004.

[9] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice-Hall, 2000.

[10] B. Marick. Bypassing the GUI. *Software Testing and Quality Engineering Magazine*, pages 41–47, 2002.

[11] A. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Phd, Dept. of Computer Science,Univ. of Pittsburgh, Jul 2001.

[12] A. M. Memon. Employing usage profiles to test a new version of a gui component in its context of use. *Software Quality Control*, 14(4):359–377, 2006.

[13] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conf/11th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, pages 118–127, New York, NY, USA, 2003. ACM Press.

[14] 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.

[15] B. A. Myers. User interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, 1995.

[16] A. Nagarajan and A. M. Memon. Refactoring using event-based profiling, 2003.

[17] 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.

[18] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proc. of the 2000 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis*, pages 158–167. ACM Press, 2000.

[19] G. H. Walton and J. H. Poore. Generating transition probabilities to support model-based software testing. *Softw. Pract. Exper.*, 30(10):1095–1106, 2000.

[20] E. J. Weyuker. Using operational distributions to judge testing progress. In *Proc. of the 2003 ACM symposium on Applied computing*, pages 1118–1122. ACM Press, 2003.

[21] L. White, H. AlMezen, and N. Alzeidi. User-based testing of GUI sequences and their interactions. In *Proc. 12th Int'l Symposium on Software Reliability Engineering*, pages 54–63, 2001.

[22] J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.

[23] D. Woit. Conditional-event usage testing. In *CASCON '98: Proc. of the 1998 Conf of the Centre for Advanced Studies on Collaborative research*, page 23. IBM Press, 1998.

[24] D. M. Woit. Specifying operational profiles for modules. In *ISSTA '93: Proc. of the 1993 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis*, pages 2–10. ACM Press, 1993.

[25] 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.

[26] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE'07, Proc. of the 29th Int'l Conf on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.