Using a Pilot Study to Derive a GUI Model for Automated Testing

QING XIE
Accenture Technology Labs
ATIF M MEMON
University of Maryland

Graphical user interfaces (GUIs) are one of the most commonly used parts of today's software. Despite their ubiquity, testing GUIs for functional correctness remains an under-studied area. A typical GUI gives many degrees of freedom to an end-user, leading to an enormous *input event interaction space* that needs to be tested. GUI test designers generate and execute test cases (modeled as sequences of user *events*) to traverse its parts; targeting a sub-space in order to maximize fault detection is a non-trivial task. In this vein, in previous work, we used informal GUI code examination and personal intuition to develop an *event-interaction graph* (EIG). In this paper we empirically derive the EIG model via a pilot study, and the resulting EIG validates our intuition used in previous work; the empirical derivation process also allows for model evolution as our understanding of GUI faults improves. Results of the pilot study show that events interact in complex ways; a GUI's response to an event may vary depending on the *context* established by preceding events and their execution order. The EIG model helps testers to understand the nature of interactions between GUI events when executed in test cases and why certain events detect faults, so that they can better traverse the event space. New test adequacy criteria are defined for the EIG; new algorithms use these criteria and EIG to systematically generate test cases that are shown to be effective on four fielded open-source applications.

Categories and Subject Descriptors: D.2.5 [**Testing and Debugging**]: Model-Based Testing; K.6.3 [**Software Management**]: Software Development

General Terms: Verification, Reliability
Additional Key Words and Phrases: Graphical user interfaces, Model-based testing, test minimization, test suite management

Prologue

Graphical user interfaces (GUIs) are typically implemented as a collection of widgets associated with event-handlers designed to respond to individual events. An event-handler's response to an event may vary depending on the current state of the software, established by preceding events and their execution order. Consider the

Authors' addresses: Qing Xie Accenture Technology Labs 161 North Clark Street, Chicago, IL, 60601, USA; Atif M. Memon, 4115 A. V. Williams Building, Department of Computer Science, University of Maryland, College Park, MD 20742, USA.

following example of how the state in which an event "`Properties`[1]" is executed affects its behavior.

—*Software platform:* Microsoft Windows XP, Professional, Version 2002, Service Pack 2 (or 1).

—*Setup steps:* Launch the application "`Computer Management`" (under "`Control Panel`" → "`Administrative Tools`").

—*Setting up the state for a crash:* ① Right-click "`Computer Management (local)`"; ② choose "`Connect to another computer ...`"; ③ in the "`Another Computer`" text-field, type a very long string (keep typing until the text-field allows no more characters); ④ click the "`OK`" button; the expected error dialog opens stating that the computer cannot be managed; ⑤ click "`OK`" to close this dialog.

—*Causing the crash:* ⑥ Right-click on "`Computer Management (...)`" and select "`Properties`"; the application will crash.

The application would not have crashed if the user had typed-in a short string as the computer name. On the other hand, if the user had executed other events (except another "`Connect to another computer ...`" with a valid or a shorter computer name; this event will be denoted by ⑦) between "setting up the state" and "causing the crash," the software would still have crashed. Hence the event sequence ① through ⑤ is all that is needed to setup the state for a crash; subsequently, anytime the event ⑥ is performed, the software will crash, unless event ⑦ is performed first.

$$\underbrace{\ldots}_{\text{ⓐ}} \rightarrow \underbrace{① \rightarrow ② \rightarrow ③ \rightarrow ④ \rightarrow ⑤}_{State\ Setup} \rightarrow \underbrace{\ldots}_{\text{ⓑ}} \rightarrow \underbrace{⑥}_{Crash}$$

Although the above example reinforces popular belief that the software state plays an important role during testing, it also illustrates several important points pertinent to *generating* such sequences. First, the subsequence ⓐ is irrelevant to this crash; it can be omitted entirely. Second, ⓑ must be generated carefully so that it is either empty, or does not include ⑦. Third, the event sequence ① through ⑥ alone is sufficient to cause the crash. In practice, during testing, it is extremely difficult (although obviously desirable) to generate such potentially problematic event sequences.

Synopsis: This paper attempts to systematically generate potentially problematic sequences by empirically understanding event sequences that lead to successful fault detection. This understanding is used to derive an event-interaction graph (EIG) (previously obtained via informal GUI code examination [Memon and Xie 2005; Xie and Memon 2006]) model of problematic GUI interactions. The EIG is systematically and efficiently "covered" to yield effective test cases.

1. INTRODUCTION

GUIs are one of the most important parts of today's software [Memon 2002]. Recognizing the importance and popularity of GUIs, software developers are dedicating

[1]This event is actually `Click-on-Menu-Item-Properties`; however, in this research, for brevity, whenever possible, an event will be denoted by a unique label derived from the attributes of its associated widget, *e.g.*, button label (`Cancel`), menu-item label (`Properties`), etc.

large parts of the code, up to 50% to implementing GUIs [Memon 2001]. One reason for the popularity of GUIs is that they offer many degrees of freedom to a user to interact with the software. The allowable number of permutations of GUI input events in most non-trivial applications is extremely large. A pilot study in Section 4 shows that the number of event sequences grows exponentially with length of sequence. Events in GUIs have complex interactions. A user interacting with a GUI may perform an event sequence X that puts the GUI in such a state that a subsequent event sequence Y causes erroneous execution. Without the *context* established by the event sequence X, the event sequence Y may not have led to the error. The pilot study of Section 4 shows that many GUI events exhibit similar behavior, *i.e.*, they cause an error in the GUI in one context but not in another context. This observation shows that, in non-trivial GUI applications, there are complex dependencies between GUI events; after all, if context played no role, then an event would behave exactly the same way each time it is executed.

The above characteristics (*i.e.*, large number of permutations of events and complex event interactions) of GUIs present new challenges for quality assurance tasks such as testing. It is not sufficient to test each event in isolation (or in one state); rather, an event needs to be executed together with other events. A test designer needs to develop test cases (sequences of events) that test the enormous input interaction space of the GUI. However, in practice, GUI test designers barely test a minuscule part of the interaction space; they employ *capture/replay tools* (discussed in Section 2) that provide very little automation, with the result that very few test cases are created for GUIs, leading to inadequate testing. In our previous work on automated GUI testing, we developed a model called the *event-flow graph* (EFG) that represents the space of *all possible* event sequences that may be executed on the GUI [Memon et al. 2001]. Later, by informally examining the GUI code, we refined this model and created an *event-interaction graph* (EIG) [Memon and Xie 2005; Xie and Memon 2005; 2006].

In this paper we empirically derive the EIG model via a pilot study, and the resulting EIG validates our intuition used in previous work. We leverage our previous work on EFGs to empirically understand how GUI events behave when executed in a test case. A new term is defined: the *minimal effective event-context* (MEEC) of an event that has detected a fault in a test case as the shortest sequence of preceding events needed to detect the fault. A pilot study involving four subject applications is conducted. The study shows that for fault-detection in a specific class of GUIs, the MEEC has a well-defined structure that may be represented by four regular expressions and used to model problematic interactions. This understanding is used to derive EIGs; each regular expression is used to develop a test adequacy criterion; algorithms generate test cases to satisfy the criteria. We note that as our understanding of GUI errors improves in the future, we may obtain additional patterns of MEEC that may yield additional criteria and/or help to evolve the EIG model.

The usefulness of EIGs is demonstrated via a case study involving four well-tested popular ($all\_time\_activity^2 > 98\%$) applications downloaded from Source-

[2]Note that the "All time activity" percentage is a good indicator of the popularity of a program; more mature programs have all time activity percentages closer to 100%. Because the Sourceforge

Forge. The results show that the EIG-generated test cases uncovered errors that were important and relevant. All the errors were reported on the SourceForge bug reporting site; in response, the developers fixed some of the bugs. Because most of the testing steps were automated (even the scripts required to setup, execute, and tear-down test cases were generated automatically) the entire process executed very quickly with very little human intervention. Most of the applications had been used and tested for a number of years; the developers had never detected our reported errors before because their own tools (and testing processes) were unable to comprehensively and automatically test the applications.

The contributions of this research include:

—definition of the term *minimal effective event context* and its application to GUI test-case generation,

—a bottom-up approach that studies actual GUI test case failures to derive the EIG model,

—demonstration that parts of GUI test cases are actually sufficient for fault detection, and

—development of four new GUI test criteria.

Structure of the paper: The next section presents an overview of related work. Section 3 defines minimal effective event context. Section 4 presents a pilot study of GUI events in test cases. Section 5 represents minimal effective event contexts using regular expressions. The structural analysis of the MEECs from the pilot study is used to develop EIGs. Section 6 evaluates test cases generated from EIGs. Finally, Section 7 concludes with a discussion of ongoing and future work.

2. RELATED WORK

The use of software models to generate sequences of events (commands, method calls, data inputs) for software testing is not new. Numerous researchers have developed techniques that employ state machine models [Clarke 1998; Chow 1978; Esmelioglu and Apfelbaum 1997; Bernhard 1994; Shehady and Siewiorek 1997; White and Almezen 2000], grammars [Imanian 2005; von Mayrhauser and Crawford-Hines 1993; von Mayrhauser et al. 1994; Maurer 1990; Auguston et al. 2005], AI planning [Memon et al. 2001; Howe et al. 1997; Scheetz et al. 1999; Leow et al. 2004], genetic algorithms [Kasik and George 1996], probabilistic models [Woit 1998; 1993; Whittaker 1992; Whittaker and Thomason 1994], and graph-traversal techniques [Memon et al. 2005; Memon and Xie 2005] to generate various types of sequences. This section presents some of these techniques (emphasizing on their application to GUIs whenever possible), a discussion of GUI testing tools, and the relationship between our work and unit testing of object-oriented programs [Jorgensen and Erickson 1994; Xie et al. 2005], and delta debugging [Zeller 2002].

State Machine Models: The most well-known models used for GUI test-case generation include finite-state machine (FSM) models [Clarke 1998; Chow 1978; Esmelioglu and Apfelbaum 1997; Bernhard 1994] and their variants [Shehady and

database also includes projects that are incomplete or abandoned, the activity percentage helps to avoid them.

Siewiorek 1997; White and Almezen 2000]. The GUI's behavior is modeled as an FSM where each input event triggers a transition in the FSM; a path in the FSM represents a test case. Since FSMs have scalability problems for large GUIs, variants such as variable finite state machines (VFSM) have been used to "collapse" some states by adding explicit conditional variables to the machine's transitions [Shehady and Siewiorek 1997]. White *et al.* [White and Almezen 2000] handle the scalability problem by manually decomposing the GUI into multiple state machines called "complete interaction sequences" (CISs); each CIS models a particular (manually identified) GUI activity called a responsibility.

As can be imagined, the nature of test cases that are generated from state-machine models depend largely on the definition of "state" and the test adequacy criteria used. No one has empirically demonstrated the effectiveness of the above state machine models for GUI testing.

Grammars: Several researchers have modeled the set of all possible valid sequences (of commands, events) as a set of sentences that may be compactly represented using a grammar [Imanian 2005; von Mayrhauser and Crawford-Hines 1993; von Mayrhauser et al. 1994; Maurer 1990; Auguston et al. 2005]. Various types of "sentence generators" are then used to obtain different types of test cases. For example, Mayrhauser *et al.* [von Mayrhauser and Crawford-Hines 1993] use grammars to generate sequences of commands to test a robot tape library system. Imanian [Imanian 2005] extends this idea to an attributed event grammar, which models events, their precedence or inclusion relation to other events, attributes of the events, and, if applicable, time delays between input events. Auguston *et al.* [Auguston et al. 2005] use a similar approach to model a software application's environment. The environment model includes a description of hazardous states in which the system may arrive. An attributed event grammar specifies all possible event traces.

AI Planning: One way to avoid the explicit state enumeration (and hence the state-explosion problem) in GUIs is to use goal-directed search, generating states on demand. One such form of goal-directed search uses AI Planning, which was first used by Howe *et al.* [Howe et al. 1997] to generate test cases for a command-driven system. Scheetz *et al.* [Scheetz et al. 1999] and Leow *et al.* [Leow et al. 2004] have applied this technique to testing object-oriented programs.

Our own work on GUI testing has also used AI planning for test-case generation [Memon et al. 2001]. The technique requires that all events be represented in the form of pre- and postconditions, *i.e.*, partial descriptions of the state in which an event can execute and a description of how the state would change, respectively. The AI Planner uses these event specifications and *tasks* (represented in the form of initial and goal states) to generate test cases. An automated test replayer executes the test cases automatically on the GUI.

Genetic Algorithms: Kasik *et al.* [Kasik and George 1996] have used genetic algorithms to "improve" test cases so that they represent novice GUI users. This approach relies on an expert to manually generate a sequence of GUI events, and then uses the genetic algorithms to modify and lengthen the sequence, thereby mimicking a novice user. The assumption made therein is that novice users, when compared to expert users, take longer "paths" through the input event interaction space when performing activities.

Probabilistic Models: Several probabilistic models have been used to encode software usage information (GUI usage may be encoded as sequences of events) [Woit 1998; 1993; Whittaker 1992; Whittaker and Thomason 1994]. These models are used to generate test cases that will exercise the software in a fashion typical of actual operation. Whittaker *et al.* [Whittaker and Thomason 1994; Whittaker 1992] describes a method for statistical testing based on a Markov chain model of software usage, which allows test input sequences to be generated from multiple probability distributions. Woit [Woit 1993; 1998] extends this idea to a conditional-event usage testing (CEUT), which retains the benefits of traditional operational profile testing but is not limited in the class of systems to which it applies. In CEUT, the expected usage of the software is modeled in such a way as to allow specification of conditions upon expected input event sequences.

Graph Traversal: Our own past work on GUI testing has used several types of graph models (*e.g.*, *event-flow graphs* [Memon et al. 2005; Memon and Xie 2005]) to generate specific types of test cases. Our choice of these models was guided by intuition, rather than an empirical understanding of event interactions and their impact on GUI fault-detection. Consequently, we obtained the models first based on intuition and later evaluated them on several software subjects. This work takes an alternative approach – we first observe the effect of GUI interactions on fault detection via a pilot study, use the observation to create the model, and then evaluate the effectiveness of the test cases generated by the model on several new applications not used in the pilot study.

GUI Testing Tools: Some tools used for GUI testing include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[3] that require manual creation of test cases. Capture/replay (record/playback) tools [Hicinbothom and Zachary 1993] such as WinRunner[4] provide very little automation [Memon 2003], especially for *creating* tests. During the capture phase, a tester interacts with the GUI being tested. The tool records the interactions and saves them in a database. The recorded test cases can be replayed automatically on the software using the replay part of the tool. Developers/testers who employ these tools typically come up with a small number of test cases [Memon et al. 1999].

Unit Testing of Object-Oriented Programs: Jorgensen *et al.* [Jorgensen and Erickson 1994] draw similarities between testing event-driven applications and unit testing of object-oriented programs (OOPs). The primary similarity is that testing object-oriented programs involves generating sequences of method calls. However, the focus of research for OOP is on more complex problems commonly faced during generation of sequences of methods, *i.e.*, choosing method parameters and their values during testing. The relationship between parameters of different methods in an intra-class test is also considered [Xie et al. 2005].

Delta debugging: The work on delta debugging by Zeller *et al.* [Zeller 2002] is somewhat related to our notion of an event context. Delta debugging views an execution of a failing program as a sequence of program states; each state induces the subsequent state, up to the failure. The delta debugging algorithm isolates the failure-relevant variables and values by systematically narrowing the state difference

[3] `http://junit.org/news/extension/gui/index.htm`

[4] `http://mercuryinteractive.com`

between a passing run and a failing run – by assessing the outcome of altered executions to determine whether a change in the program state makes a difference in the test outcome. In our work, events cause state transitions in the GUI. We define the context of an event in terms of event sequences; Zeller *et al.* view the context of a failure as a collection of variables that caused the failure. However, instead of debugging, our focus is on test case generation.

All the above techniques are useful, in that they can be used to generate different types of test cases for different domains. The research presented in this paper is orthogonal to all the work described above; it will serve to augment all these techniques for GUI testing by specifying the types of sequences to generate for effective testing. For example, we envision that this work can be used to build better models and adequacy criteria for state machines that target problematic interactions, better sentence generators based on grammars, improved crossover and mutation operators for genetic algorithms, and axioms for AI planners.

3. MODELING THE MINIMAL EFFECTIVE EVENT CONTEXT

The overall goal of this paper is to create an abstract model of GUIs that can be used to generate potentially problematic event sequences. This section takes the first step to obtaining such a model by providing a formal definition of minimal effective event context (MEEC) of an event X. Intuitively, the MEEC of X is (one of) the *shortest* (in terms of number of events) event sequence that needs to be executed before X detects a GUI fault, *i.e.*, one that manifests itself on the visible GUI. Subsequent sections will empirically study the structure of the MEEC and use it to create the new model called an EIG.

3.1 Preliminaries

Note that some of the terms presented in this section have been defined in detail in earlier work [Memon and Soffa 2003; Memon et al. 2001; Memon 2001]. They are reproduced here only to the extent needed to understand the concepts presented in this paper.

A GUI's state is modeled as a set of *objects* (`label`, `form`, `button`, `text`, etc.) and a set of *properties* of those objects (`background-color`, `font`, `caption`, etc.). Hence, at a particular time t, the GUI can be represented by its constituent **objects** $O = \{o_1, o_2, \ldots, o_m\}$, and their **properties** $P = \{p_1, p_2, \ldots, p_l\}$. Note that all properties correspond to physical properties of widgets in the GUI. The set of objects and their properties constitutes the *state* of the GUI.

The state of a GUI is not static; events $\{e_1, e_2, \ldots, e_n\}$ performed on the GUI change its state. Events are modeled as state transducers. The function notation $S_j = e_i(S_i)$ is used to denote that S_j is the state resulting from the execution of event e_i in state S_i. An event e_i is *applied* to state S_i to obtain state S_j. Events can be strung together into sequences. Extending the function notation above, $S_j = (e_1 \circ e_2 \circ \ldots \circ e_n)(S_i)$, where $e_1 \circ e_2 \circ \ldots \circ e_n$ is an executable event sequence, denotes that S_j is the state that results from executing the specified sequence of events starting in state S_i. $e_1 \circ e_2 \circ \ldots \circ e_n$ is an *executable event sequence for a state* S_0 (S_0 is one of the states in which the software starts) iff there exists a sequence of states $S_0; S_1; \ldots; S_n$ such that $S_i = e_i(S_{i-1})$, for $i = 1, \ldots, n$.

Our earlier work modeled the set of all possible executable event sequences as

an *event-flow graph* (EFG). An EFG contains nodes (that represent events) and edges. An edge from node n_x to n_y means that the event represented by n_y may be performed *immediately after* the event represented by node n_x. This relationship is called the `follows` relationship – *i.e.*, n_y `follows` n_x. Figure 1 shows a very simple GUI and its associated EFG (the different shapes used for the events will be discussed in Section 4.6). The EFG contains eight nodes, corresponding to the eight events in the GUI (`Edit`, `Copy`, `Cut`, `Paste`, `Goto Line`, `OK`, `Cancel`, `Line Number()`). The directed edges show the flow of events. For example, there is an edge from `Edit` to `Copy` indicating that a user can execute `Copy` immediately after `Edit`; ignoring short-cut keys, there is no edge from `Copy` to `Paste`. It is important to note that an EFG is not a state-machine model. The nodes are not states – they are events; edges are not state-transitions – they represent the `follows` relationship. However, a state machine model that is equivalent to the EFG can be constructed – the state would capture the possible events that can be executed on the GUI at any instant; transitions cause state changes whenever the set of available events changes. An important property of EFGs that makes them useful is that they can be obtained automatically from an executing GUI using a reverse engineering technique implemented in a tool called the *GUI Ripper* [Memon et al. 2003]. Details of the GUI Ripper have been described in earlier reported work [Memon et al. 2003]; in summary, the Ripper is given a handle to the GUI's main window; it uses windowing API (*e.g.*, Java Swing API) to extract all those widgets that have event listeners associated with them, and executes these events in succession, thereby opening more windows; the Ripper recursively performs this depth-first traversal of the GUI's structure; the output of this step is the EFG.

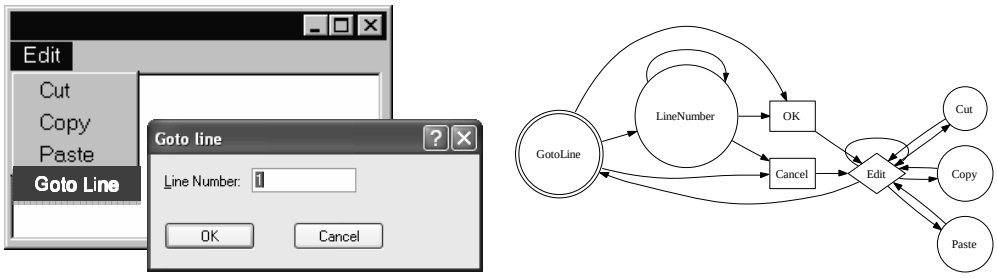


Fig. 1. A Simple GUI and its associated EFG

A set of states S_I is called the *valid initial state* set for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked. Given a GUI in state $S_i \in S_I$, *i.e.*, in a valid initial state of the GUI, new states may be obtained by performing events on S_i. These states are called the *reachable* states of the GUI. The state S_j is a *reachable state* iff either $S_j \in S_I$ or there exists an executable event sequence $e_x \circ e_y \circ \ldots \circ e_z$ such that $S_j = (e_x \circ e_y \circ \ldots \circ e_z)(S_i)$, for any $S_i \in S_I$.

3.2 GUI Test Cases

A *test case* is a pair $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$, where $S_0 \in S_I$ is a valid initial state and $e_1 \circ e_2 \circ \ldots \circ e_n$ is an executable event sequence.

As noted earlier, if performed manually (using capture/replay tools), test-case generation is extremely labor intensive. With the event-flow model, numerous graph-traversal techniques may be used to automate it. A straightforward way to generate test cases is to start from a known initial state S_0 of the GUI and use a graph traversal algorithm, enumerating the nodes during the traversal, on the EFGs. If the event requires text input, *e.g.*, for the `Line Number()` text-box, then its value is read from a database, initialized by the software tester. An executable sequence of events $e_1 \circ e_2 \circ \ldots \circ e_n$ is generated as output that serves as a GUI test case. One such sequence for the EFG of Figure 1 is $< S_0, \texttt{Edit} \circ \texttt{Copy} \circ \texttt{Edit} \circ \texttt{Paste} \circ \texttt{Edit} \circ \texttt{GotoLine} \circ \texttt{OK} >$.

3.3 Detecting GUI Faults & Effective Event Context

During the execution of a test case $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$, all its events are executed one-by-one and a *test oracle* is used to determine if the software executed correctly for the test case. The test oracle computes the GUI's expected state, obtains the GUI's actual state, compares the two states, and determines if the actual is as expected. In earlier work [Memon et al. 2000], we defined a test oracle to contain *oracle information* that is used as the expected output and an *oracle procedure* that compares the oracle information with the actual output. Intuitively, an *oracle information generator* automatically derives the *oracle information* (expected state sequence) $S_1; S_2; \ldots; S_n$. Likewise, the *actual state* $A_1; A_2; \ldots; A_n$ is obtained from an *execution monitor*. The execution monitor may use techniques such as screen scraping and/or querying [Memon et al. 2000] to obtain the actual state of the executing GUI. An *oracle procedure* then automatically compares the two states (S_i and A_i) and determines if the GUI is executing as expected.

Definition: An event e_i in a test case *detects a fault* if the expected (S_i) and actual (A_i) states mismatched immediately after e_i was executed. □

Not all of e_i's preceding events $e_1 \circ \ldots \circ e_{i-1}$ in the test case contribute to the fault being detected, suggesting that some of these events may be removed. In general, not all events may be removed since they are necessary to establish the context in which e_i detected the fault. Hence a subsequence $e_j \circ \ldots \circ e_k$ (for $1 \leq j \leq (n-1)$ and $(j+1) \leq k \leq (n-1)$) of $e_1 \circ \ldots \circ e_n$ is sufficient for e_i to detect the fault. The resulting test case would be $< S_0, e_j \circ \ldots \circ e_k \circ e_i >$. Care must be taken that event e_j is applicable in the test case's starting state S_0, and e_i is applicable in the state S_k resulting from the execution of e_k. This leads to the definition of the *effective event context* (EEC) of an event in terms of a test case and a fault F detected by event e_i.

Definition: Given a test case $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$, the *EEC* of event $e_i \in \{e_1, e_2, \ldots, e_n\}$ that has detected a fault F is the pair $(S_0, e_j \circ \ldots \circ e_k)$, for $1 \leq j \leq (n-1)$ and $(j+1) \leq k \leq (n-1)$ such that e_j is applicable in S_0 and e_i is applicable in S_k; $e_j \circ \ldots \circ e_k$ is a subsequence of $e_1 \circ \ldots \circ e_n$; state S_k is obtained by applying $e_j \circ \ldots \circ e_k$ starting in S_0. □

The above definition precludes subsequences with deleted *intermediate* events. Although this is not strictly required (a generalization is a subject for future work), it does simplify the computation of the EEC. Had we allowed the deletion of intermediate events, we expect the computation of the EEC to be more complex and expensive.

Note that an event may have multiple EECs. The MEEC is defined as any of the EECs of minimum length needed to detect the fault.

Definition: Given a test case $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$, and a fault F that was detected by an event $e_i \in \{e_1, e_2, \ldots, e_n\}$, the *MEEC* of e_i is any of the EECs of minimum length needed to detect the fault F. □

It is non-trivial to predict the structure and length of MEECs for typical GUI test cases and faults. A deeper understanding of MEECs will help to develop better test case generation algorithms. An empirical approach is used to understand MEECs. In particular, a pilot study of real failed GUI test cases is conducted on several GUI applications and the MEEC for each test case is extracted and studied. The study is described next.

4. PILOT STUDY: EMPIRICALLY UNDERSTANDING THE STRUCTURE OF MEEC

This section describes a pilot study to help understand the structure of MEECs for GUIs. Observations from this study are used to create event-interaction graphs (EIGs).

This study attempts to answer the following questions:

Q1:. How many event sequences is a user allowed to execute in a typical GUI-based software application?

Q2:. Do GUI events interact? Is it sufficient to test each event once?

Q3:. When a GUI test case (*i.e.*, consisting of a sequence of events $e_1, e_2, e_3, \ldots, e_n$) reveals a fault at event e_i,[5] what is the role of the context established by preceding events $e_1, e_2, e_3, \ldots, e_{i-1}$? Which of the preceding events are actually needed for fault detection?

Q4:. What is the structure of the MEEC?

The following process is used to answer the above questions.

Step 1:. Take different GUI-based software subjects.

Step 2:. Artificially seed faults in them (borrowing this technique from mutation testing); hereafter, the fault-seeded versions will be referred to as "mutants."

Step 3:. Generate test cases; each test case is of the form $< S_0; e_1 \circ e_2 \circ \ldots \circ e_n >$, where S_0 is the state of the GUI in which the event sequence $e_1 \circ e_2 \circ \ldots \circ e_n$ is executed.

Step 4:. Execute each test case on each mutant. A mutant is *killed* if there is a mismatch between the mutant's GUI state and the original software's GUI state. Record the event at which the mismatch was observed.

Step 5:. For each test case $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$ that killed a mutant at event e_i, $1 \leq i \leq n$, compute the MEEC.

4.1 Step 1: Study Subjects

The study subjects are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate

[5]Recall that the GUI test case is executed one event at a time; a fault may be detected during the execution of one of the events.

students of the senior Software Engineering course. It is called TerpOffice[6] and includes TerpWord (a word-processor with drawing capability), TerpCalc (a scientific calculator), TerpPaint (an imaging tool), and TerpSpreadSheet (a compact spreadsheet program). Additional details of these applications have been discussed earlier [Memon and Xie 2005]. In summary, these applications are fairly large with complex GUIs; with the exception of TerpCalc, they are comparable in size to Microsoft's WordPad software.

4.2 Step 2: Fault Seeding

Fault seeding is a well-known technique used to introduce known faults into programs [Offutt and Hayes 1996; Harrold et al. 1997]. It has several applications – in this paper, it is used to create fault-seeded versions (*i.e.*, mutants) of the subject applications. Test cases are generated and executed simultaneously on the mutants and the original subject application. A mutant is killed if there is a mismatch between the original software's GUI state and the mutant's GUI state. The characteristics of the test cases that were successful at killing mutants are then studied.

Note that killing a mutant is a popular way to simulate the process of fault detection by a test case. In a real testing scenario, a tester creates a test case together with a description of an "expected outcome" for the software. A software that does not execute as expected *fails* on the test case; otherwise it *passes*. By using a "golden version" of the software and mutants, we are side-stepping the creation of descriptions of "expected outcomes" for each test case. A mutant that behaves exactly like the golden version on an input is observationally equivalent to the original software; hence the input (*i.e.*, the test case) has been unable to "reveal the fault" that was seeded in the code to create the mutant. Note that other researchers have shown that mutants are good representatives of actual software faults [Andrews et al. 2005].

The number and classes of faults that were seeded in this study are identical to those used in our earlier work [Memon and Xie 2005]. In short, we were careful to (1) seed faults that represented "real" faults found in GUI programs, (2) spread the faults uniformly throughout the code by computing the number of opportunities to seed the particular class of faults, (3) avoid fault interaction, and (4) employ multiple people to seed the faults. In all, 200 fault seeded versions of each application were created, each seeded with exactly one fault. We adopted a history-based approach to seed GUI faults, *i.e.*, we observed "real" GUI faults in real applications. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[7] was used by the developers to report and track faults in the previous version of TerpOffice while they were working to extend its functionality and developing the subsequent version. The reported faults are an excellent representative of faults that are introduced by developers during implementation. Some examples include *modify relational operator* (>, <, >=, <=, ==, !=), *negate condition statement*, *modify arithmetic operator* (+, -, *, /, =, ++, –, +=, -=, *=, /=), and *modify logical operator* (&&, ||).

[6] `http://www.cs.umd.edu/users/atif/TerpOffice`
[7] `http://bugs.cs.umd.edu`

4.3 Step 3: Test-Case Generation

Next, the total number (by length) of event sequences that may be executed on the subject applications was computed. The results are summarized in Figure 2. The x-axis shows the length of the event sequence; the y-axis (logarithmic scale) shows the number of sequences. The graph shows that the number of event sequences grow exponentially with length of sequence. This computation answers **Q1**. It would be extremely expensive to generate and execute all event sequences beyond $length > 3$.

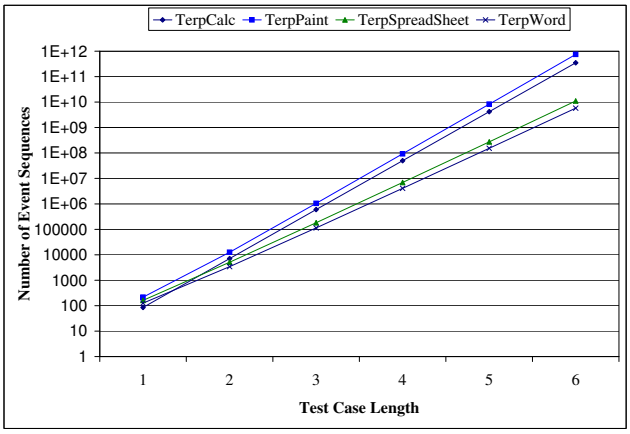


Fig. 2. Total Number of Event Sequences

Since the total number of potential event sequences (and hence the number of test cases) is enormous, in this study, a reasonable subset needed to be generated. Any process that is employed may have an impact on the results of the study (note the threats to validity in Section 6.1). Thus a process that GUI testers use in practice, *i.e.*, generate test cases that cover each event at least once, will be used here to minimize threats to external validity.

An EFG-based algorithm (shown in Figure 3) was used to generate the test cases; the EFG was represented by a function `follows`($Event$) that returns a set of events that may be executed after $Event$. For each subject application, the algorithm started in the application's start state S_0, created a list of events that could be executed in S_0 (Line 10), and chose one event $nextEvent$. It continued to make the event sequence iteratively longer by selecting another event using the `follows` relationship (Line 16). Whenever possible, it selected the least frequently used events by maintaining a *used* integer counter with each event. The $getLeastUsedEvents()$ function examined these values for a set of possible subsequent events and selected the one that had been used the smallest number of times – ties are broken by random selection. Because our GUI test case executor encounters timing problems for test cases longer than 50 events, we bounded the test-case length to 50 events; note that M is the parameter used to control test-case length.

In principle, the process used in the `GenerateTestSuite` algorithm may waste resources by generating test cases that are redundant, *i.e.*, copies (which is why

```
TestSuite /* Returns generated test suite for a given EFG*/
ALGORITHM :: GenerateTestSuite(
  N = Number of test cases needed;                                    1
  M = Maximum length of a test case;                                  2
  E = Set of events in the GUI;                                       3
  S0 = Start state for all test cases.)                               4

  TestSuite = φ;                                                      5
      /* initialize all events "used" counter to 0 */
  FORALL ei ∈ E DO                                                    6
   used(ei) = 0;                                                      7
      /* initialize test case to empty */
  tc = Φ;                                                             8
  WHILE TRUE DO                                                       9
      /* get list of all initial events */
   nextEventSet = events(S0);                                         10
   WHILE TRUE DO                                                      11
      /* Get the event that has not been used much yet. */
    nextEvent = getLeastUsedEvent(nextEventSet);                      12
      /* Add the event to the test case */
    tc = Append(tc, nextEvent);                                       13
      /* Update the used entry for the event. */
    used(nextEvent)++;                                                14
      /* done if maximum length reached */
    IF Length(tc) ≥ M THEN BREAK;                                     15
      /* Which events can come next? */
    nextEventSet = follows(nextEvent);                                16
      /* Is this a Termination event (has no follows) */
    IF nextEventSet == φ THEN BREAK;                                  17
   END WHILE;                                                         18
   TestSuite = Union(TestSuite, tc);                                  19
   testCases = SizeOf(TestSuite);                                     20
   IF testCases ≥ N THEN BREAK;                                       21
  END WHILE;                                                          22
  RETURN TestSuite;                                                   23
```

Fig. 3. Generate Test Cases from an Event-Flow Graph

we use the set *Union* operation in Line 19 to eliminate duplicates). However, the space of all possible test cases for our subject applications was so large (each event's `follows` set had a large number of events) that a test case was never re-generated; each iteration yielded a unique test case. As Figures 4 and 5 show (for a subset of these test cases), the algorithm yielded test cases that had both good event coverage and length distribution. Because each test case needed to be executed on 200 fault-seeded versions, and each test case's execution takes an average of 30 seconds, we capped the number of test cases at 10k per application, keeping the total CPU time around 15 days per application; in reality, the total running time was almost two months per application due to various execution platform problems.

4.4 Step 4: Test Execution

The above algorithm was able to generate a large number of long test cases that contained all the events in the software. A test executor executed each generated

test case automatically on the subject applications and all the mutants. It performed all the events in each test case and compared the mutant's GUI state with the original software's GUI state. Events were triggered on the GUI using the native OS API. The test cases executed on four machines (Pentium 4, 2.2GHz, each with 256MB RAM) simultaneously, one application per machine.

As expected, not all test cases were successful at killing mutants. Only those that killed at least one mutant were stored; the rest were discarded. In all, 1119 (300, 300, 282, and 237 for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord, respectively) test cases successfully killed at least one mutant. In all, 163 (96, 23, 11, and 33 for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord, respectively) mutants were killed. The longest successful test case had 50 events and the shortest one had 1 event. Figure 4 shows the length distribution of these test cases. The graph has four lines, one for each application. The x-axis is the length of the test case and the y-axis is the number of test cases. As the lines show, the test cases varied in length.

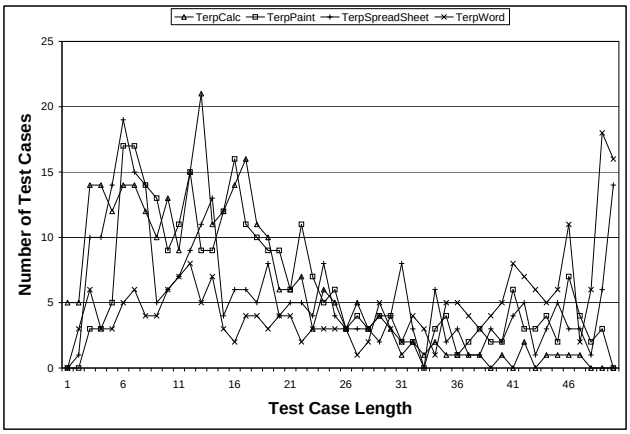


Fig. 4. Test Case Length Freq. Distribution

Figure 5 shows the event distribution of all the test cases. The figure shows four column graphs; the x-axis shows all the events in each application; the y-axis shows the number of times a particular event was executed by a test case. The graphs show that the generated test cases had good event coverage.

There were many complex interactions between the events in the subject applications. If an event e_x killed a mutant M_y in a test case, it failed to kill M_y in many other test cases. Moreover, if e_x occurred in test case T_i multiple times, it killed M_y at only one point. This observation showed that the context of an event seriously affected its ability to kill a mutant. The data has been compressed into 4 plots shown in Figure 6. The x-axis in these plots represents individual events in the GUI. The dotted line shows the number of times a particular event existed in some test case but failed to kill a mutant. The solid line shows the number of times the event killed a mutant. These plots show that while many events killed one or more mutants, the same event failed to kill a mutant in many instances.

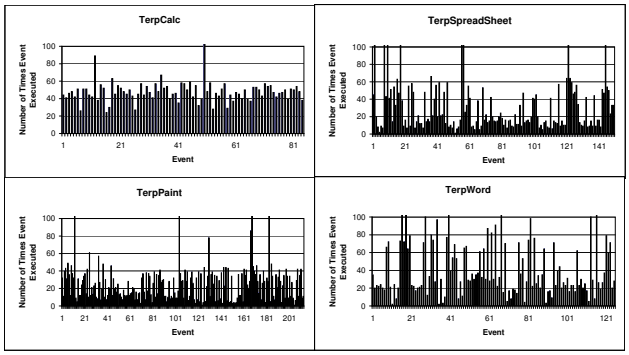


Fig. 5. Event Distribution

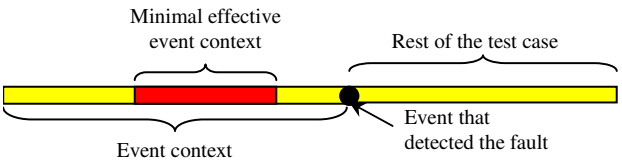
For example, event #120 in TerpWord killed at least one mutant in 300 test cases; however, the same event, although it existed in 600 other test cases, failed to kill even a single mutant. Also, note that the presented results show test cases that killed at least one mutant; among the 10k total test cases per application, there were many test cases that did not kill a single mutant but comprised of events that were otherwise successful in other contexts at killing mutants. This observation addresses **Q2**.

The output of this step was a set of fault matrices, one for each application. The rows in a fault matrix represent test cases; columns represent faults; entry (i, j) is $TRUE$ if test case i detected fault j; otherwise it is $FALSE$. In our four fault matrices, a total of 1119 rows had at least one $TRUE$ entry. The number of columns that had at least one $TRUE$ entry was 96, 23, 11, and 33 for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord, respectively.

4.5 Step 5: Studying Predecessor Events

For a test case $< S_0, e_1 \circ e_2 \circ \ldots \circ e_n >$ in which the event e_x (for $1 \leq x \leq n$) killed a mutant $\mathcal{M}$, all possible subsequences $< e_i \circ \ldots \circ e_j >$ of $< e_1 \circ e_2 \circ \ldots \circ e_{x-1} >$ were created keeping only those in which the first event (e_i) was applicable in S_0 and e_x `followed` the last event (e_j). Test cases were then obtained by appending e_x to the chosen subsequences. Starting from the shortest of these test cases, they were executed on the same mutant that was killed by the original test case, stopping when one successfully killed $\mathcal{M}$. The predecessor events in this (shortest) test case form the *Minimal Effective Event Context* (MEEC formally defined in Section 3).

For illustration, each test case is shown as a horizontal line with 2 levels of shading. The dark band shows the MEEC.



The above shaded-horizontal-line visualization is now stacked for all test cases per

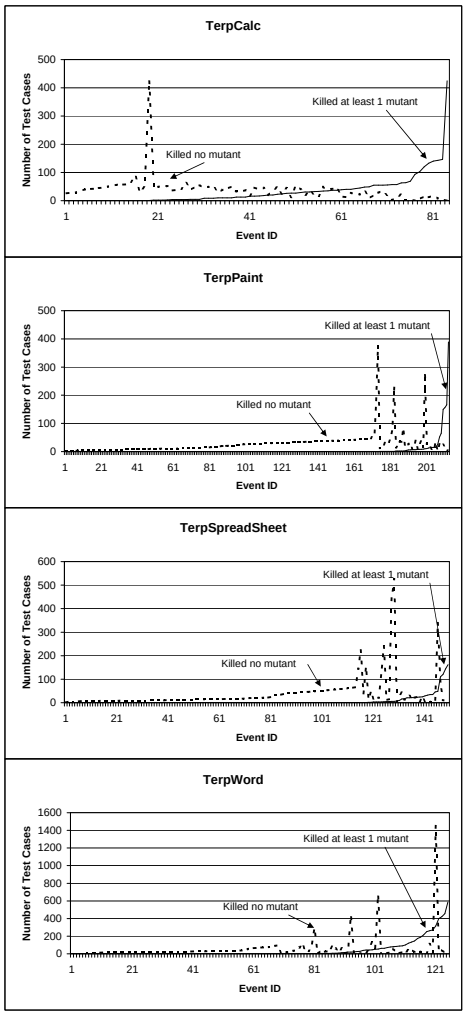


Fig. 6. Events Interactions

application to summarize the results. Figure 7 shows the results for TerpCalc. The x-axis shows the event number in the test case. The y-axis represents test case failures. The result for TerpCalc shows that the average length of the MEEC for TerpCalc was 2.21 events. This result showed that even though the entire test case was long (50 events in many cases), large parts of the test case were in fact useless for fault detection. The test cases would still be able to detect all the faults even if all the events except those in the MEEC were ignored. Figures 8, 9, and 10 show the same results for TerpPaint, TerpSpreadSheet, and TerpWord respectively. The average length of the MEEC was 3.57, 4.62, and 3.86 respectively. This result answers **Q3**.

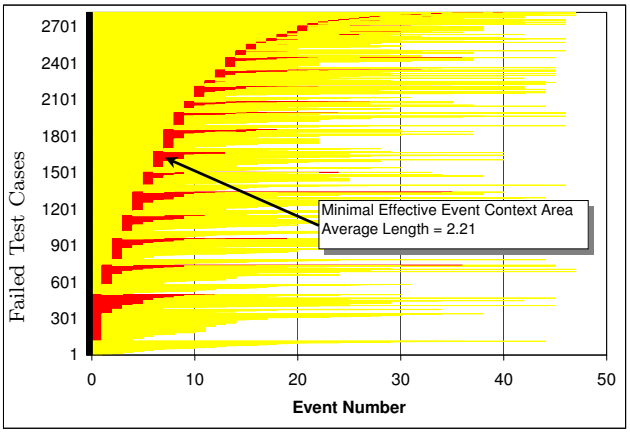


Fig. 7. MEEC for TerpCalc

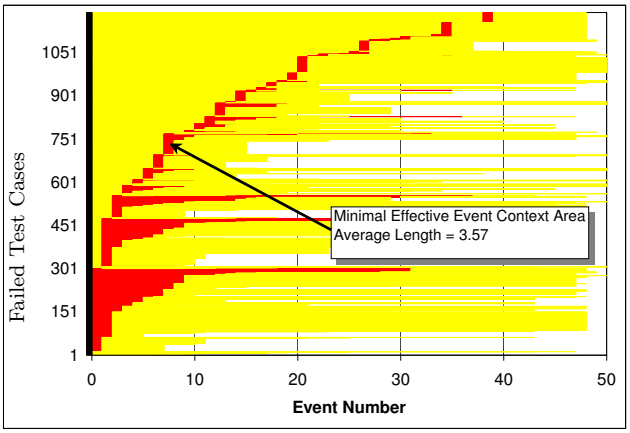


Fig. 8. MEEC for TerpPaint

4.6 Structure of MEEC

The following classification of GUI events helped to further understand the structure of MEECs:

—*reachability events* (denoted by a symbol `R`) that are used to open windows/menus. One subset of `R` of interest is `W`, the set of events that open windows. The remaining events in `R` are used to open menus. For the example of Figure 1, `R` = {`Edit`, `Goto Line`}; `W` = {`Goto Line`}. Events from class `R` will be shown as a diamond, except for `W` that will be shown as a double-circle.

—other events that are used to manipulate the structure of the GUI include *termination events* (`T`) that close windows. In Figure 1, `T` = {`OK`, `Cancel`}. Each `T` event will be shown as a rectangle.

—events that do not manipulate the structure of the GUI are called *system-interaction events* (denoted by symbol `S`); for Figure 1, `S` = {`Cut`, `Copy`, `Paste`}. Each `S` event

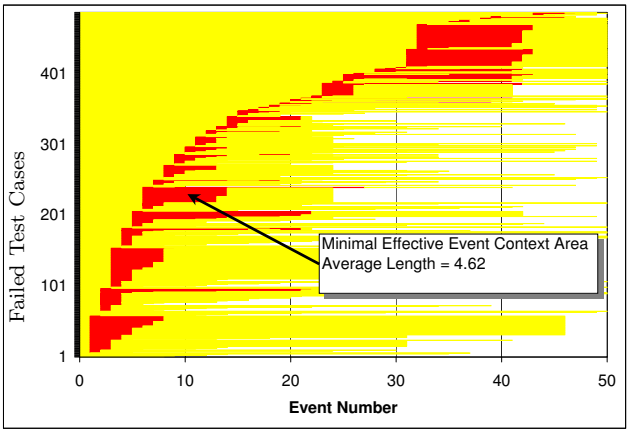


Fig. 9. MEEC for TerpSpreadSheet

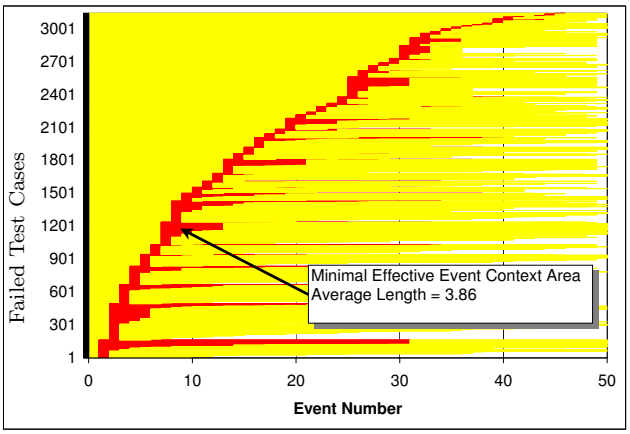


Fig. 10. MEEC for TerpWord

will be shown as a circle.

Note that a similar classification was used in our previous work [Memon 2001]. The above event classes were then used to create an abstraction of the MEECs – each event was replaced by one of the symbols `S`, `T`, or `R`, depending on its function in the GUI. The resulting strings were then compactly represented using regular expressions. A largely manual two-step process was used for this compaction. First, all repetitions were compacted using the well-known "*" and "+" operators. A large number of MEECs ended up as R*, R*S, and R*SR+. In the second step, we examined the remaining MEECs and found that they could be represented using R*SR*(SR*)+. The result of this compaction process yielded a total of four regular expressions R*, R*S, R*SR+, and R*SR*(SR*)+, each of which was assigned a "pattern ID" 1, 2, 3, and 4 respectively. The failed test cases were then partitioned by pattern ID. The MEECs did not contain any event of class `T`.

Pattern ID	MEEC Structure	e_x	# Faults	Unique Faults	% Unique Faults
TerpCalc					
Total Faults Detected				**96**	
1	**R***	**S**	37	37	**38.5%**
		W	1	1	**1.0%**
2	**R*S**	**S**	54	40	**41.7%**
		W	1	0	**0.0%**
3	**R*SR+**	**S**	10	3	**3.1%**
4	**R*SR*(SR*)+**	**S**	42	15	**15.6%**
TerpPaint					
Total Faults Detected				**23**	
1	**R***	**S**	14	14	**60.9%**
		W	6	6	**26.1%**
2	**R*S**	**S**	1	1	**4.3%**
		T	2	2	**8.7%**
TerpSpreadSheet					
Total Faults Detected				**11**	
1	**R***	**S**	4	4	**36.4%**
		T	1	1	**9.1%**
		W	3	3	**27.3%**
2	**R*S**	**S**	3	3	**27.3%**
3	**R*SR+**	**S**	2	0	**0.0%**
TerpWord					
Total Faults Detected				**33**	
1	**R***	**S**	9	9	**27.3%**
		T	4	4	**12.1%**
		W	15	15	**45.5%**
2	**R*S**	**T**	3	2	**6.1%**
3	**R*SR+**	**S**	1	1	**3.0%**
		T	1	1	**3.0%**
4	**R*SR*(SR*)+**	**S**	2	1	**3.0%**

Table I. Regular Expression Table

The result of this overall process is shown in Table I. Column `MEEC Structure` of this table shows the regular expression. Column e_x shows the type of event that killed the mutant. The number of faults is shown in Column `# Faults`. The numbers in `# Faults` are somewhat misleading because a single fault may be manifested as multiple failures. Consequently, multiple test cases may detect the same fault, causing us to count it several times for each pattern; which is why the sum of all faults does not add to the `Total Faults Detected` value. This result answers **Q4**.

An alternative measure, shown in the column `Unique Faults`, shows the number of faults that were detected by test cases with Pattern i *but not with Pattern* $i-1$. This measure will be useful when developing new test case generation techniques based on these results. The main idea of defining this measure is that it is less expensive to generate event sequences using "Pattern $i-1$" than with "Pattern i."

Table I illustrates several important points. First many faults (38.5%) in TerpCalc were detected by test cases with Pattern 1, *i.e.*, zero or more events from `R` were followed by an event in `S`; only one fault was detected when using an event of type `W` after zero or more `R` events. Pattern 1 was also effective in TerpPaint (87%), TerpSpreadSheet (72.7%), and TerpWord (84.8%); event types `W` and `T` for e_x played more significant roles in these applications. Second, Pattern 2 was very effective for TerpCalc (41.7%) when e_x was an S type of event; the same pattern was less effective for other applications. Third, Patterns 3 and 4 were not very effective in any application, except TerpCalc.

This analysis showed that parts of test cases with well-defined structures are in fact sufficient for fault detection in GUIs. As shown in Table I, not all mutants were killed by our test cases. Hence we suspect that the derived set of four patterns is incomplete. In future work, we plan to generate additional test cases, kill additional mutants, and perhaps obtain new patterns that may help to evolve the EIG model. Moreover, the four patterns are specific to our four subject applications, all of which have very little back-end code; most of the application code is for GUI manipulation. An end-user uses these "desktop" applications via simple widget operations; the software responds fairly quickly. We expect that the patterns will change for other application types that have large, complex, long-running back-end code.

However, the four derived patterns provide us with a good starting point to model potentially problematic event sequences and use the model to *generate* test cases for desktop applications. We make the following observations.

—A large number of faults are detected with test cases that execute a number of reachability events (*i.e.*, R*), followed by either a window opening event (`W`) or a system-interaction event (`S`); the reachability events are needed to simply "reach" the event that killed the mutant. According to Pattern 1, it is important to test all `S`, `W` and `T` events at least once. In terms of EFGs, this essentially means that each node of type `S`, `T`, and `W` is covered by the test suite. We call this the **STW-event coverage criterion**.

—For Pattern 2, it is important to test interactions between event pairs (`S`, `S`), (`S`, `T`), and (`S`, `W`). In terms of EFGs, this means that the test suite should cover such edges. We call this the **STW-interaction coverage criterion**.

—For Pattern 3, it is important to test specific types of paths between two `S` events, and `S` and `T` events. These paths should only contain `R` types of events. We call this the **SS-ST-path coverage criterion**.

—For Pattern 4, it is important to test paths between multiple `S` events, where each path contains only `R` type of events. We call this the **S^+-path coverage criterion**.

The STW-event and STW-interaction coverage criteria may be satisfied by generating specific types of event sequences from an EFG. However, satisfying the SS-ST-path and S^+-path coverage criteria require the computation of paths between pairs of events; the event-interaction graph (EIG) will model these paths. The next section formally describes an EIG and outlines a method to transform an EFG to EIG.

5. EVENT-INTERACTION GRAPH

As mentioned earlier, there is a correspondence between executable event sequences and paths in an EFG. Intuitively, an event-flow path represents a sequence of events that can be executed on the GUI. Formally, an event-flow-path is defined as follows:

Definition: There is an *event-flow-path* from node n_x to node n_y iff there exists a (possibly empty) sequence of nodes $n_j; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ in the event-flow graph E such that $\{(n_x, n_j), (n_{j+k}, n_y)\} \subseteq edges(E)$ and $\{(n_{j+i}, n_{j+i+1}) for\ 0 \leq i \leq (k-1)\} \subseteq edges(E)$. □

In the above definition, the function *edges* takes an event-flow graph as input and returns a set of ordered-pairs, each representing an edge in the event-flow graph. The notation $< n_1; n_2; \ldots; n_k >$ is used for an event-flow path. Two examples of event-flow paths for the GUI in Figure 1 are <`Copy`; `Edit`; `Goto Line`; `Cancel`> and <`Line Number(2)`; `OK`; `Edit`; `Copy`; `Edit`; `Paste`>.

According to Patterns 3 and 4, it is important to model all paths between pairs of two `S` types of events and an `S` event and a `T` event. However, these paths should not contain intermediate `T` or `S` types of events. This leads to the definition of an important property of paths.

Definition: An event-flow-path $< n_1; n_2; \ldots; n_k >$ is *interaction-free* iff none of $n_2, \ldots, n_{k-1}$ represent termination or system-interaction events. □

For example the path <`Copy`; `Edit`; `Goto Line`; `Cancel`> is interaction-free because `Edit` and `Goto Line` are neither termination or system-interaction events. However, <`Line Number(2)`; `OK`; `Edit`; `Copy`; `Edit`; `Paste`> is not interaction free because `OK` is a termination event.

Definition: A system-interaction event (and termination event) e_x *interacts-with* system-interaction and termination event e_y iff there is at least one interaction-free event-flow-path from the node n_x (that represents e_x) to the node n_y (that represents e_y). □

For example, event `Cut` interacts with event `Copy` because there is at least one interaction-free event flow path from `Cut` to `Copy`; the path is <`Cut`; `Edit`; `Paste`>. Note that an event may interact-with itself. Also note that "e_x interacts-with e_y" does not necessarily imply that "e_y interacts-with e_x."

The interacts-with relationship is used to create the event-interaction graph. This graph contains nodes, one for each `S` and `T` types of events. An edge from node n_x (that represents e_x) to node n_y (that represents e_y) means that e_x interacts-with e_y.

The EIG for the EFG of Figure 1 is shown in Figure 11. Note that the number of nodes in an EIG is smaller than in the corresponding EFG; the number of edges may be more because each edge may potentially represent a path.

The EIG can be obtained automatically from the EFG. The simplest way to visualize this transformation is to identify all `R` types of events and remove them one by one, adjusting the edges on a per-node basis. Each deletion has the side-effect of yielding a *mapping* that will be used to obtain executable test cases. Figure 12 shows an intermediate graph for this transformation that is obtained after removing `Edit`. The mapping (`Edit`, `Cut`), (`Edit`, `Copy`), (`Edit`, `Paste`), and (`Edit`, `GotoLine`) are generated for the events `Cut`, `Copy`, `Paste`, and `GotoLine`, respectively. Subsequent removal of `GoToLine` yields the EIG and mapping (`GotoLine`, `Cancel`), (`GotoLine`,

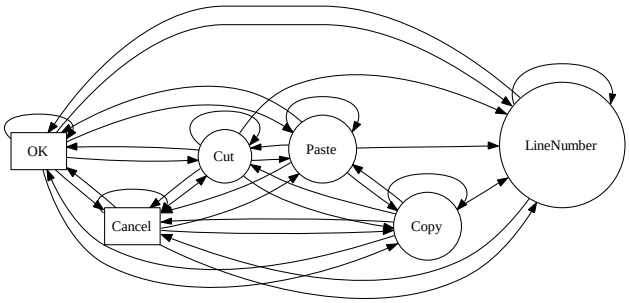


Fig. 11. Event Interaction Graph

`OK`), (`GotoLine`, `LineNumber`), for `Cancel`, `OK`, and `LineNumber`, respectively.

We note that the intuition behind an EIG and interaction-free paths is similar to that used by Weyuker and Reps for data-flow criteria [Rapps and Weyuker 1982], except at a different level of abstraction. Weyuker and Reps use a program-flow graph (PCG) to represent execution paths in an imperative style code; they use the notion of def-clear paths (*i.e.*, ones that do not redefine a variable) to define data-flow relationships. In our case, we model software at the GUI event level of abstraction, and use interaction-free paths to capture the event-interaction relationships.

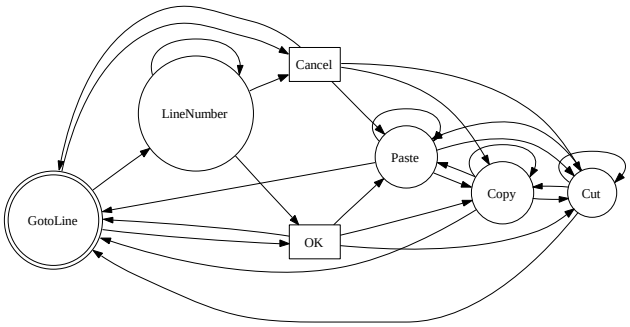


Fig. 12. An Intermediate Transformation Step from EFG to EIG

We remind the reader that our original motivation for defining the EIG is to generate test cases that satisfy the four criteria defined earlier. All the edges of an EIG may be covered to generate event sequences for Pattern 3 discussed earlier. An example of an event sequence obtained in this way is <`Cut`; `Paste`>. Similarly, *pairs* of adjacent edges between `S` events may be covered for Pattern 4. An example is <`Cut`; `Copy`; `Paste`>. The remaining question is how to execute the obtained event sequences as they are missing important reachability events (recall that events were deleted from the EFG to obtain the EIG; hence sequences generated from the EIG will not contain reachability events). To enable execution, the mapping is used to generate the intermediate reachability events. Using the mapping (`Edit`, `Cut`),

(`Edit`, `Copy`), and (`Edit`, `Paste`) for events `Cut`, `Copy`, and `Paste`, respectively, the test cases obtained from the sequences <`Cut`; `Paste`> and <`Cut`; `Copy`; `Paste`> are $< S_0, \texttt{Edit} \circ \texttt{Cut} \circ \texttt{Edit} \circ \texttt{Paste} >$ and $< S_0, \texttt{Edit} \circ \texttt{Cut} \circ \texttt{Edit} \circ \texttt{Copy} \circ \texttt{Edit} \circ \texttt{Paste} >$, respectively for initial state S_0.

Because our derived EIG model is exactly the same as the one used in a earlier reported study [Memon and Xie 2005], we have already evaluated some EIG-based test cases on the same subject applications and faults used in the study of Section 4. In that work, the EIG-based test cases had detected 128, 156, 55, and 125, for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord, respectively as opposed to 96, 23, 11, and 33 respectively for the test cases generated in the Pilot study. However, this comparison is not sufficient for the current work because we have now derived the EIG model from the same faults/applications. Hence, in the next section, we describe a case study using four different applications and faults to evaluate EIG-based test cases.

6. CASE STUDY: EVALUATING THE EIG-BASED TEST CASES

The test cases obtained from the EIG may be automatically executed by a "robot." Starting in the test case's initial state (*i.e.*, S_0), the robot plays the test case one event at a time. However, the test cases also need to be combined with a test oracle to determine whether a software under test executes correctly for the input event sequence. This case study will employ an automated test oracle that observes *software crashes*, *i.e.*, abnormal termination of the application. Hence, for this study, a test case fails if the software crashes, otherwise it passes.

The algorithms used to obtain the EIG and test cases have been implemented in a software tool called GUITAR (`guitar.cs.umd.edu`). Together with GUITAR's existing reverse engineering tools and test case replayer, they yield a fully automatic software system that provides an end-to-end solution for GUI testing.

To minimize threats to external validity (others discussed in Section 6.1), this case study is conducted on four new applications (not used in Section 4). They have several characteristics that make them different from the TerpOffice applications. They were not developed in-house by students; rather, they were developed by "professional" developers. Also, they are some of the most widely-used and evolving applications available on SourceForge.net; they have extensive bug reports and have undergone significant quality assurance before release. In fact FreeMind, GanttProject, and JMSN have been around for 7, 4, and 5 years, respectively; downloaded by 3,468,917, 32,341,733, and 252,133 users, respectively; have 439, 732, and 33 bug reports, respectively, of which 279, 453, and 20, respectively have been fixed. We expected to find very few problems in these applications except in CrosswordSage, which is relatively new.

Moreover, the original MEEC patterns were derived using mismatches between fault-seeded versions and their faultless counterparts. Which is why a different approach to fault detection is being used here – software crashes are used as the basis for software faults.

Several versions of these popular Open Source Software (OSS) were downloaded from SourceForge.net. The primary goal of this study was to evaluate the effectiveness of the test cases obtained using the EIG model, and hence the MEEC-based

approach. The secondary goals include determining whether the test cases reveal serious problems in fielded GUI-based software; multi-version applications are selected to determine whether some of these problems persist across different versions of the software.

More specifically, the following questions need to be answered:

RQ1:. Do popular GUI-based OSS have serious problems (crashes) that can be detected by the EIG-generated test cases?

RQ2:. What is the nature of the problems?

RQ3:. Do these problems persist across multiple versions of the OSS?

RQ4:. What are the common causes of the problems?

Subject Applications: The following four applications with GUIs developed using Java Swing are used in this study:

1. **FreeMind**[8], which is a premier free mind-mapping[9] software written in Java. It has an all time activity of 100%. Versions 0.0.2, 0.1.0, 0.4, 0.7.1, 0.8.0RC5 and 0.8.0 are tested.

2. **GanttProject**[10], which is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets). It has an all time activity of 99.96%. Versions 1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, and 2.pre1 are tested.

3. **JMSN**[11], which is a pure Java Microsoft MSN Messenger clone, including Instant messaging, File Send/Receive, msnlib (for developers), and additional chat log, etc. It has an all time activity of 97.68%. Versions 0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, and 0.9.9b1 are tested.

4. **CrosswordSage**[12], which is a tool for creating (and solving) professional looking crosswords with powerful word suggestion capabilities. When tested, it had an activity percentile (last week) of 98.21%. Versions 0.1, 0.2, 0.3.0, 0.3.1, 0.3.2, and 0.3.5 are tested.

The overall process (reverse engineering, EIG creation, test-case generation and execution) executed on each version without any human intervention in 5-8 hours; one machine per application. The reverse engineering, model creation, test case generation steps took 2-3 minutes per application. The test cases were designed to satisfy the STW-event, STW-interaction, and SS-ST-path coverage criteria. Coverage of the S^+-path coverage criterion is a subject for future research. The test case execution took the remaining time. If needed, test case execution could be greatly sped up by splitting up the test suite for each application across multiple machines.

Manual setup included setting up a database for text-field values. Because the overall study execution needed to be fully automatic, a "default" 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,*

[8]http://sourceforge.net/projects/freemind
[9]http://en.wikipedia.org/wiki/Mind_map
[10]http://sourceforge.net/projects/ganttproject
[11]http://sourceforge.net/projects/jmsn
[12]http://sourceforge.net/projects/crosswordsage

Subjects	Versions						
	0.0.2	**0.1.0**	**0.4**	**0.7.1**	**0.8.0RC5**	**0.8.0**	**Total**
FreeMind	1550	1964	4118	13658	50872	52216	124378
	1.6	**1.9.11**	**1.10.3**	**1.11**	**1.11.1**	**2.0.pre1**	**Total**
GanttProject	1240	3705	3878	4015	4015	4414	21267
	0.9a	**0.9.2**	**0.9.5**	**0.9.7**	**0.9.8b7**	**0.9.9b2**	**Total**
JMSN	1015	1107	1156	1218	1591	1777	7864
	0.1	**0.2**	**0.3.0**	**0.3.1**	**0.3.2**	**0.3.5**	**Total**
CrosswordSage	101	134	818	818	876	1524	4271
Total							157780

Table II. Number of Test Cases Generated for Each Version of Each Application

Subjects	Versions						
	0.0.2	**0.1.0**	**0.4**	**0.7.1**	**0.8.0RC5**	**0.8.0**	**Total**
FreeMind	2	5	4	4	5	4	24
	1.6	**1.9.11**	**1.10.3**	**1.11**	**1.11.1**	**2.0.pre1**	**Total**
GanttProject	3	4	3	3	3	3	19
	0.9a	**0.9.2**	**0.9.5**	**0.9.7**	**0.9.8b7**	**0.9.9b2**	**Total**
JMSN	2	2	1	2	3	3	13
	0.1	**0.2**	**0.3.0**	**0.3.1**	**0.3.2**	**0.3.5**	**Total**
CrosswordSage	0	0	3	3	2	5	13
Total							69

Table III. Number of Crashes for Each Version of Each Application

non-existent file name}. Note that if a text field is encountered in the GUI (represented as an event called `type-in-text`), one instance for each text type is tried in succession.

The number of test cases generated for each application version is shown in Table II. These test cases revealed a total of 69 crashes. Note that the same crash is counted several times if they were detected in different versions. The results are summarized in Table III for each version of each application. This result answers question **RQ1**.

To address question **RQ2**, all the crash logs were manually examined and the test cases that caused the crash were identified. The analysis of the results is summarized next. Note that version numbers are shown in parenthesis. Each listed bug will be referred by its *bug number* in later discussions.

FreeMind: 1. NullPointerException when trying to open a non-existent file (0.0.2, 0.1.0);

2. FileNotFoundException when trying to save a file with a very long file name (0.0.2, 0.1.0, 0.4);

3. NullPointerException when clicking on some buttons on the main toolbar when no file is open (0.1.0);

4. NullPointerException when clicking on some menu items if no file is open (0.1.0, 0.4, 0.7.1, 0.8.0RC5);

5. NullPointerException when trying to save a "blank" file (0.1.0);

6. NullPointerException when adding a new node after toggling folded node (0.4);

7. FileNotFoundException when trying to import a non-existent file (0.4, 0.7.1,

0.8.0RC5, 0.8.0);

8. FileNotFoundException when trying to export a file with a very long file name (0.7.1, 0.8.0RC5, 0.8.0);

9. NullPointerException when trying to split a node in "Edit a long node" window (0.7.1, 0.8.0RC5, 0.8.0);

10. NumberFormatException when setting non-numeric input while expecting a number in "preferences setting" window (0.8.0RC5, 0.8.0);

Gantt Project: 1. NumberFormatException when setting non-numeric inputs while expecting a number in "New task" window (1.6);

2. FileNotFoundException when trying to open a non-existent file (1.6);

3. FileNotFoundException when trying to save a file with a very long file name (1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

4. NullPointerException after confirming any preferences setting (1.9.11);

5. NullPointerException when trying to save the content to a server (1.9.11);

6. NullPointerException when trying to import a non-existent file (1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

7. InterruptedException when trying to open a new window (1.10.3);

8. Runtime error when trying to send e-mail (1.11, 1.11.1, 2.pre1);

JMSN: 1. InvocationTargetException when trying to refresh the buddy list (0.9a, 0.9.2);

2. FileNotFoundException when trying to submit a bug/request report because the submission page doesn't exist (0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, 0.9.9b2);

3. NullPointerException when trying to check the validity of the login data (0.9.7, 0.9.8b7, 0.9.9b2);

4. SocketException and NullPointerException when stopping a socket that has been started (0.9.8b7, 0.9.9b2);

Crossword Sage: 1. NullPointerException in Crossword Builder when trying to delete a word (0.3.0, 0.3.1);

2. NullPointerException in Crossword Builder when trying to suggest a new word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

3. NullPointerException in Crossword Builder when trying to write a clue for a word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

4. NullPointerException when loading a new crossword file (0.3.5);

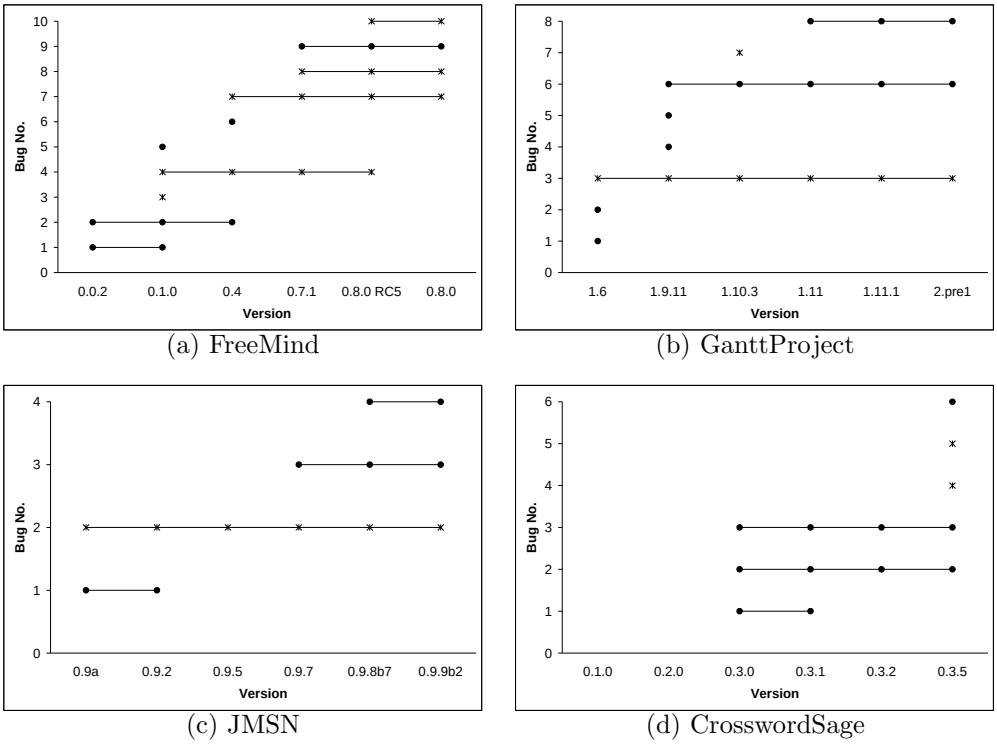
5. NullPointerException when splitting a word (0.3.5);

6. NullPointerException when publishing the crossword (0.3.5);

The above list of severe problems show that fielded GUI-based OSS have problems that are quickly uncovered using the EIG-based testing process. In the future, we expect to detect a larger number of faults with test cases that satisfy the S^+-path coverage criterion. This result answers question **RQ2**.

To answer question **RQ3**, the history of each bug was studied. Figure 13 gives an overview of bug history across versions of each application. The x-axis represents the versions; the y-axis uses the bug numbers assigned earlier. Each bug that led to one crash is represented by a small filled circle in Figure 13; bugs that led to multiple crashes are represented by an asterisk. If the same bug persisted across multiple versions, the circles (and asterisks) are connected by a horizontal line. For example, many crashes are caused by Bug#3 in FreeMind (several toolbar buttons should be

disabled if there is no file opened). Figure 13 shows that many bugs are persistent across versions. For example, Bug#4, #7, #8, #9 and #10 in FreeMind persisted across several versions before they were discovered and fixed. The same observation holds for the other applications. In fact, Bug#3 in GanttProject appeared in the first version tested (version 1.6 was chosen because it is the first version with default language English); it exists in all versions, including the latest version. This result answers question **RQ3**.



(a) FreeMind (b) GanttProject

(c) JMSN (d) CrosswordSage

Fig. 13. Bug History Over Versions

To answer question **RQ4**, the reasons for the crashes were studied. Four reasons were identified for these crashes: (1) *Invalid text input.* Many crashes were detected because the software does not check the validity and size of text input. For example, some text boxes in GanttProject and FreeMind expect an integer input; providing a string resulted in a crash. In some instances, a "very long" text input also resulted in a crash, such as providing a "very long" text input as the file name while saving such a file sometimes leads to FileNotFoundException. (2) *Widget enabled when it should be disabled.* One challenge in GUI design is to identify allowable sequences of interactions with widgets and to disallow certain sequences. Designers often disable certain widgets in certain contexts. In these open-source applications, it is found that several instances of widgets were enabled when they should really have

been disabled. When the tests executed the incorrectly enabled widget in an event sequence, the software crashed. (3) *Object declared but not initialized.* Some of the crashes were Java NullPointerExceptions. It turned out that as the software was evolving, one developer, not seeing the use of an object, commented out a part of the code, which was responsible for object initialization. Another developer continued to use the object in another part of the code. The software crashed when the uninitialized object was accessed. (4) *Obsolete external resources.* Some of the crashes in JMSN were caused by test cases that were trying to retrieve information from a web page that is no longer available. This result answers question **RQ4**.

This study showed that the abstraction of the EIG model derived from the MEEC structure was important for the detection of these problems; exercising interactions between certain types of events was essential. For example we encountered several bugs using test cases that were of the form of Pattern 3 of MEEC (R*SR+): (1) Crash 4 of FreeMind was detected by the execution of < `File` ∘ `Close` ∘ `Edit` ∘ `SetAllVisible` >, where `Close` and `Set All Visible` are system-interaction events, and `File` and `Edit` are menu-open events, (2) Crash 6 of FreeMind was detected by the execution of < `Edit`∘`Node`∘`ToggleFolded`∘`Edit`∘`Node`∘ `NewNode` >, where `Edit` and `Node` are menu-open events, and `Toggle Folded` and `New Node` are system-interaction events, and (3) Crash 8 of GanttProject was detected by < `ClickOnAnExistingResource`(`GainFocus`)∘`Resources`∘`SendEmail` >, where `Resources` is a menu-open event and the rest are system-interaction events. The event handlers for many of these events were distributed across many classes of the application, making it expensive (and in many cases impossible) to detect these problems via static analysis. Indeed, we checked these applications using the FindBugs tool [Spacco et al. 2006], which failed to reveal these problems.

Figure 13 leads to another observation. There are fewer bugs in the first version than in later versions. For example, there are two crash-causing bugs in Version 0.0.2 of FreeMind. Typically, the first version of an OSS is relatively simple and is developed by a small group of core developers. This version typically undergoes QA before its first release; hence it is reasonably stable. Versions 0.1.0 and 0.2.0 of CrosswordSage have no bugs because they are very simple. The only change that was made from Version 0.1.0 to Version 0.2.0 was a new help document. As the developer community grows, the application becomes more complex and prone to bugs. For example, Bug#10 in FreeMind was first introduced when a new "preference setting" functionality was added. Similarly, there was a new feature added to Version 0.3.0 of Crossword Sage; this new feature introduced some bugs that were detected. There were more features added in Version 0.3.5; bugs were detected in the added part of code.

6.1 Threats to Validity

Because the results of this work are based on empirical studies, the overall research should be considered keeping in mind several threats to validity. Some of these threats have been mentioned at relevant places in the paper; this section consolidates all these threats.

First the pilot study was based on four in-house Java applications that have very little back-end code. We expect that the nature and types of patterns will be different for other types of applications, *e.g.*. form-based GUIs with complex

database back-ends. The artificial seeded faults were instantiated from a dozen classes of popular faults. While every effort was made to seed the faults fairly, the EIG model may need to be revised for other instances and classes of faults. The original algorithm used to generate test cases and the test oracle used to determine mismatches resulted in mutants that could not be killed. As our test cases and oracles improve, we may be able to derive other patterns to further evolve the EIG model. Finally, the process used to determine the MEEC did not consider the deletion of intermediate events. We feel that this deletion may yield more compact MEEC structures.

Second, the case study results are also based on four applications, which also have a fairly simple back-end. We selected applications that were not developed in house. In the same spirit, we also used a different test oracle, one based on software crashes rather than GUI state mismatches. In the future, we intend to improve the test oracles to detect functionality failures beyond crashes.

7. CONCLUSIONS & FUTURE WORK

This paper addressed a fundamental problem of testing GUIs. Because GUI test cases are sequences of events, generating long test cases can be prohibitively expensive (and impossible for large GUIs). A new concept called the *minimal effective event context* (MEEC) was presented and used to empirically demonstrate that for fault detection, the MEEC is short and has a well-defined structure, which may be represented by four compact regular expressions. This result was used to automatically develop a reduced event-interaction model of the software. This model, called an event-interaction graph (EIG), was used to generate and execute test cases on four software applications; the regular expressions provided four test coverage criteria, namely STW-event, STW-interaction, SS-ST-path, and S^+-path coverage criteria. These test cases were effective at revealing a large number of faults. We showed several examples of bugs detected by test cases that were of the form of Pattern 3 (`R*SR+`) of MEECs. Since SourceForge has a bug reporting/tracking tool for each project, some bugs were reported. For example, Bug#4 in FreeMind for version 0.8.0RC5 was reported (bug #1245216 in SourceForge[13]). In response to the report, the developers fixed this bug in release 0.8.0. This showed that the bugs found by the testing were relevant. We do however note that some software problems, such as "easter eggs" can only be detected by carefully hand-crafted event sequences.

The regular expression model of MEEC provided a strong starting point for this work; in the future we expect to derive additional patterns of MEECs that will help to further refine the EIG model and obtain additional coverage criteria. Examination of the faults that could not be detected by any of the test cases may be used as a starting point for the work.

The structure of the MEEC was based solely on the dissection of failed test cases. It should be noted that a successful test case may be also be manipulated (*e.g.*, events deleted) so that it fails. In the future, examination of these additional test cases may provide valuable insights into test case behavior that may be used to further evolve test cases.

[13] http://sourceforge.net/tracker/index.php?func=detail &aid=1245216&group_id=7118&atid=107118

The MEEC was used to develop a graph model of the GUI. In the future, the same patterns may be used to improve other types of models such as state machines and grammars. Data mining techniques may be used to study characteristics of MEECs and to extract patterns of events that are effective for fault detection.

The MEEC may be used to improve regression testing. For example, instead of storing an entire test case for regression testing, its MEEC may be extracted and used for regression testing. This is especially useful for GUI testing because large parts of test cases make the test case obsolete from one GUI version to another [Memon and Soffa 2003].

All other detected bugs will be reported, especially the ones in the latest versions of all the applications. By default, all the applications were tested in one machine configuration on Windows 2000 Professional. It is observed that altering this "default" configuration helps to uncover more bugs. In a preliminary study, GanttProject was tested in a new configuration with a much lower memory setting than the default configuration. Bug#4 and Bug#7 surface only in this low memory configuration. In case of Bug#4, the application tries to repaint all the GUI windows/widgets after the preferences setting have changed; in low memory, this causes a substantial delay for the user. Any event performed during the slow repainting process causes an uncaught NullPointerException exception. In case of Bug#7, the application requires additional time to open new windows; if a user performs a new event during this time, the result is an uncaught InterruptedException exception. In the future, a model of the system configuration will be used to detect additional problems.

A surprising result is that some bugs existed across applications. This was due to shared open-source GUI components. For example, Bug#2 in FreeMind and Bug#3 in GanttProject are identical since both these applications share a *FileSave* component. This component throws a FileNotFoundException when given a very long file name, which cannot be handled by the Windows operating system. This particular bug does not show up after Version 0.4 of FreeMind; however, the same bug still shows up when the user tries to *export* a file with a very long file name. This observation shows that OSS that use shared components must "sanitize" inputs before passing them to the shared components. In the future, techniques will be developed to test GUI components.

Although this research has been conducted on GUI software, it is reasonable to expect that it has applicability to other non-GUI applications. In the near future, we will extend it to test object-oriented applications, where a test case is a sequence of method calls (which may be modeled as a sequence of events), objects have state, and events have interactions.

Acknowledgments

We thank the anonymous reviewers whose comments and suggestions helped to extend the empirical study, reshape its results, and improve the flow of the text. 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.

REFERENCES

Andrews, J. H., Briand, L. C., and Labiche, Y. 2005. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM Press, New York, NY, USA, 402–411.

Auguston, M., Michael, J. B., and Shing, M.-T. 2005. Environment behavior models for scenario generation and testing automation. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*. ACM Press, New York, NY, USA, 1–6.

Bernhard, P. J. 1994. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology 3,* 3 (July), 201–220.

Chow, T. S. 1978. Testing software design modeled by finite-state machines. *IEEE trans. on Software Engineering SE-4, 3*, 178–187.

Clarke, J. M. 1998. Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference*. Pnsqc/Pacific Agenda, Portland, OR.

Esmelioglu, S. and Apfelbaum, L. 1997. Automated test generation, execution, and reporting. In *Proceedings of Pacific Northwest Software Quality Conference*. Pnsqc/Pacific Agenda, Portland, OR, 127–142.

Harrold, M. J., Offut, A. J., and Tewary, K. 1997. An approach to fault modelling and fault seeding using the program dependence graph. *Journal of Systems and Software 36,* 3 (Mar.), 273–296.

Hicinbothom, J. H. and Zachary, W. W. 1993. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*. SPECIAL SESSIONS: Demonstrations, vol. 2. Human Factors and Ergonomics Society, Santa Monica, CA, 1042.

Howe, A., von Mayrhauser, A., and Mraz, R. T. 1997. Test case generation as an AI planning problem. *Automated Software Engineering 4*, 77–106.

Imanian, J. A. 2005. Automatic test case generation for reactive software systems based on environment models. Ph.D. thesis, Naval Postgraduate School, Monterey, CA.

Jorgensen, P. C. and Erickson, C. 1994. Object-oriented integration testing. *Commun. ACM 37,* 9, 30–38.

Kasik, D. J. and George, H. G. 1996. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*. ACM Press, New York, 244–251.

Leow, W. K., Khoo, S. C., and Sun, Y. 2004. Automated generation of test programs from closed specifications of classes and test cases. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 96–105.

Maurer, P. M. 1990. Generating test data with enhanced context-free grammars. *IEEE Software 7,* 4 (July), 50–55.

Memon, A., Banerjee, I., and Nagarajan, A. 2003. GUI Ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 260–269.

Memon, A. M. 2001. A comprehensive framework for testing graphical user interfaces. Ph.D. thesis, Department of Computer Science, University of Pittsburgh.

Memon, A. M. 2002. GUI testing: Pitfalls and process. *IEEE Computer 35,* 8 (Aug.), 90–91.

Memon, A. M. 2003. Advances in GUI testing. *Advances in Computers, ed. by Marvin V. Zelkowitz 58*, 150–203.

Memon, A. M., Nagarajan, A., and Xie, Q. 2005. Automating regression testing for evolving GUI software. *Journal of Software Maintenance 17,* 1, 27–64.

Memon, A. M., Pollack, M. E., and Soffa, M. L. 1999. Using a goal-driven approach to generate test cases for GUIs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 257–266.

Memon, A. M., Pollack, M. E., and Soffa, M. L. 2000. Automated test oracles for GUIs. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, New York, NY, USA, 30–39.

Memon, A. M., Pollack, M. E., and Soffa, M. L. 2001. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering 27,* 2 (Feb.), 144–155.

Memon, A. M. and Soffa, M. L. 2003. 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*. ACM Press, New York, NY, USA, 118–127.

Memon, A. M., Soffa, M. L., and Pollack, M. E. 2001. 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*. ACM Press, New York, NY, USA, 256–267.

Memon, A. M. and Xie, Q. 2005. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering 31,* 10 (Oct.), 884–896.

Offutt, A. J. and Hayes, J. H. 1996. A semantic model of program faults. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*. ACM Press, New York, NY, USA, 195–200.

Rapps, S. and Weyuker, E. J. 1982. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 272–278.

Scheetz, M., von Mayrhauser, A., France, R., Dahlman, E., and Howe, A. E. 1999. Generating test cases from an OO model with an ai planning system. In *Proceedings of The 10th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 250–259.

Shehady, R. K. and Siewiorek, D. P. 1997. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*. IEEE Press, Washington - Brussels - Tokyo, 80–88.

Spacco, J., Hovemeyer, D., and Pugh, W. 2006. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. ACM Press, New York, NY, USA, 133–136.

von Mayrhauser, A. and Crawford-Hines, S. 1993. Automated testing support for a robot tape library. In *Proceedings of The Fourth International Software Reliability Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 6–14.

von Mayrhauser, A., Mraz, R. T., and Walls, J. 1994. Domain based regression testing. In *Proceedings of The International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 26–35.

White, L. and Almezen, H. 2000. Generating test cases for GUI responsibilities using complete interaction sequences. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*. IEEE Computer Society, Washington, DC, USA, 110.

Whittaker, J. A. 1992. Markov chain techniques for software testing and reliability analysis. Ph.D. thesis, University of Tennessee, Knoxville, TN, USA.

Whittaker, J. A. and Thomason, M. G. 1994. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng. 20,* 10, 812–824.

Woit, D. 1998. Conditional-event usage testing. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, Indianapolis, Indiana, USA, 23.

Woit, D. M. 1993. Specifying operational profiles for modules. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*. ACM Press, New York, NY, USA, 2–10.

Xie, Q. and Memon, A. M. 2005. Rapid "crash testing" for continuously evolving GUI-based software applications. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, Washington, DC, USA, 473–482.

Xie, Q. and Memon, A. M. 2006. Automated model-based testing of community-driven open source GUI applications. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 145–154.

Xie, T., Marinov, D., Schulte, W., and Notkin, D. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*. Spinger, Berlin, 365–381.

Zeller, A. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes 27,* 6, 1–10.