# Generating Event Sequence-Based Test Cases Using GUI Run-Time State Feedback

Xun Yuan, *Member, IEEE,* and Atif M Memon, *Member, IEEE*

**Abstract**—This paper presents a *fully automatic* model-driven technique to generate test cases for Graphical user interface-based applications (GUIs). The technique uses feedback from the execution of a *"seed test suite,"* which is generated automatically using an existing structural *event-interaction graph* model of the GUI. During its execution, the run-time effect of each GUI event on all other events pinpoints *event-semantic interaction* (ESI) relationships, which are used to automatically generate new test cases. Two studies on eight applications demonstrate that the feedback-based technique (1) is able to significantly improve existing techniques and help identify serious problems in the software and (2) the ESI relationships captured via GUI state yield test suites that most often detect more faults than their code-, event-, and event-interaction-coverage equivalent counterparts.

**Index Terms**—GUI testing, automated testing, model-based testing, GUITAR testing system.

## 1 Introduction

Automated test case generation (ATCG) has become increasingly popular due to its potential to reduce testing cost and improve software quality [1]. A typical approach used for ATCG is to create an abstract model (*e.g.*, state-machine model [2], [3], event-flow model [4]) of the application under test (AUT) and employ the model to generate test cases. While successful at reducing overall testing cost, in practice, ATCG continues to be resource-intensive, especially to create and maintain the model. A few researchers have recognized that these tasks may be aided by leveraging the execution results of some existing test cases. Consequently, they have developed *automated feedback-based techniques* to augment the models [5]–[14]. These techniques require an initial test case/suite to be created, either manually or automatically, and executed on the software. Feedback from this execution is used to augment a preliminary model of the AUT and *automatically* generate additional test cases. The nature of feedback depends largely on the goal of the ATCG algorithm. A common example of feedback is a code coverage report used to automatically generate additional test cases that improve overall test coverage [8]–[11], [13], [14]. Few techniques use feedback from the AUT's *run-time state* to generate additional test cases, *e.g.*, in the form of outcomes of programmer-supplied predicates in the code to cover all non-isomorphic inputs [12], operational abstractions to cover increased program behaviors [6], [7], and non-exception-throwing method-call sequences to generate longer sequences [15].

This paper presents a new feedback-based technique for automated testing of graphical user interfaces (GUIs). GUIs lend themselves to the feedback-based approach for a number of reasons. First, existing fully automatic model-based GUI test-case generation algorithms produce test cases that exhaustively test only *two-way interactions* between GUI events; these test cases are called smoke tests [4]; they form the seed suite. Systematically generating 3-, 4-, 5-, and above multi-way test cases remains an open area of research. Second, existing tools are easily adapted to monitor and store the run-time state of the GUI. Finally, GUI testing is extremely important because GUIs are used as front-ends to most software applications and constitute as much as half of software's code [16]. A correct GUI is necessary for trouble-free execution of the application's underlying "business logic" [2], [3], [17].

The new feedback-based technique has been used in a fully automatic end-to-end process for a specific type of GUI testing. The seed test suite (in this case the smoke tests) is generated automatically using an existing *event-interaction graph* (EIG) model of the GUI, which represents *all possible sequences* of events that may be executed on the GUI. The smoke suite is executed on the GUI using an automatic test case replayer. During test execution, the run-time state of GUI widgets is collected and used to automatically identify an *Event Semantic Interaction* (ESI) relationship between pairs of events. This relationship captures how a GUI event is related to another in terms of how it modifies the other's execution behavior. Informally, event $e_x$ is ESI-related to $e_y$ iff $e_x$ influences the run-time behavior of $e_y$, where "run-time behavior" is evaluated in terms of properties of GUI widgets. The ESI relationships are used to automatically construct a new model of the GUI, called the *Event Semantic Interaction Graph* (ESIG). Because the seed suite is generated from the EIG (a structural model) and the ESI relationship is obtained in terms of event execution (a dynamic activity), the ESIG captures certain structural and dynamic aspects of the GUI. The ESIG is used to automatically generate new test cases. These test cases have an important property – each event is ESI-related to its subsequent event, *i.e.*, it was shown to influence the subsequent event during execution of the seed suite.

- *X. Yuan is currently a Software Engineer in Test at the Google Kirkland office.*
  *E-mail: xyuan@cs.umd.edu*
- *A M Memon is with the Department of Computer Science, University of Maryland, College Park, MD 20742.*
  *E-mail: atif@cs.umd.edu*

This entire process, including the scripts required to set up, execute, and tear down test cases, has been implemented and executes without human intervention. Two independent studies have been conducted on eight GUI-based Java applications to evaluate and understand this new approach. In an earlier report of this work [5], we described the first study, which used four well-tested and popular applications downloaded from SourceForge; the study demonstrated that the feedback-based technique improves existing techniques with little additional cost. The ESI relationship is successful at identifying complex interactions between GUI event handlers that lead to serious failures. We presented details of some failures, emphasizing on why they were not detected by the earlier techniques. The failures were reported on the SourceForge bug reporting site;[1] in response, the developers fixed some of the bugs. The developers had never detected our reported failures before because their own tools and testing processes were unable to comprehensively and automatically test the applications.

We now extend the research with the second study, conducted on four fault-seeded Java applications developed in house; this study shows that (1) the automatically identified ESI relationships between events help to generate test suites that detect more faults than their code-, event-, and event-interaction-coverage equivalent counterparts, (2) certain characteristics of the seeded faults prevent their detection by the earlier technique, but not the new technique, (3) several of our missed faults remain undetected because of limitations with our automated GUI-based test oracle (a mechanism that determines whether a test case passed or failed), and (4) several of the remaining undetected faults require long event sequences.

Finally, we note that 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 [18]–[22], grammars [23]–[25], AI planning [26], [27], genetic algorithms [28], probabilistic models [29], architecture diagrams [30], and specifications [31] to generate such sequences. All of these techniques are useful, in that they can be used to generate different types of test cases for different domains. All of them are based on manually created models. Our research presented in this paper is orthogonal to the other model-based techniques; we focus on enhancing an existing model (in our case the model is obtained automatically) via test execution feedback. We feel that this approach may be used for the other model-based techniques mentioned above – these other models may also be enhanced with software execution and test execution feedback.

The main contributions of this work include:

- extension of work on automated, model-based, systematic GUI test-case generation,
- definition of new relationships among GUI events based on their execution,
- utilization of run-time state to explore a larger input space and improve fault-detection,
- immersion of the feedback-based technique into a fully automatic end-to-end GUI testing process and demonstration of its effectiveness on fielded and fault-seeded applications,
- empirical evidence tying fault characteristics to types of test suites, and
- demonstration that certain faults require well-crafted combinations of test cases and oracles.

The next section discusses related literature. Section 3 introduces basic GUI concepts and reviews the EIG model that forms the basis of the new ESIG model. Section 4 defines the ESI relationship and uses it to define an ESIG. Sections 5 and 6 evaluate the new feedback-based technique. Finally, Section 7 concludes with a discussion of future work.

## 2 Related Work

To the best of our knowledge, this is the first work that utilizes run-time information as feedback for model-based GUI test-case generation. However, run-time information has previously been employed for various aspects of test automation, and model-based testing has been applied to conventional software as well as *event-driven software* (EDS). This section presents an overview of related research in the areas of model-based and EDS testing, GUI testing, and the use of run-time information as feedback for test generation.

### 2.1 Model-based & EDS Testing

Model-based testing automates some aspect of software testing by employing a model of the software. The model is an abstraction of the software's behavior from a particular perspective (*e.g.*, software states, configuration, values of variables, etc.); it may be at different levels of abstraction, such as abstract states, GUI states, internal variable states, or path predicates.

**State Machine Models:** The most popular models used for software testing are *state machine models*. They model the software's behavior in terms of its abstract or concrete states; they are typically represented as state-transition diagrams. Several types of state machine models have been used for software testing, such as *Finite State Machine Models* (FSM) [32]–[35], *UML Diagram-based Models* [36] and *Markov Chains* [37].

Various extensions of FSMs have also been used for testing. These extensions use variables to represent *context* in addition to states; the goal is to reduce the total number of states by using an orthogonal mechanism, in the form of explicit variables, to select state transitions. For example, an *extended finite state machine* (EFSM) makes use of a data state along with the input for state transformation [35]; this EFSM is used by a tool called TestMaster to generate test cases by traversing all paths from the start state to the exit state.

Because test cases for EDS are sequences of events, many practitioners and researchers have found it natural to use state machine models for testing EDS [38]–[40]. The EDS is modeled in terms of states; events form transitions between states. Algorithms traverse these machine models to generate sequences of events. For example, Campbell *et al.* have applied

1. For example, https://sourceforge.net/tracker/?func=detail&atid=535427&aid=1536078&group_id=72728.

state machine models to test object-oriented reactive systems [38]. Object states are modeled in terms of instance variable values; transitions are obtained from method invocations; test cases are sequences of method calls and are generated by traversing the model.

### 2.2 GUI Test Case Generation

Several automated techniques have been developed for GUI test case generation. All of them use a model of the software; algorithms generate test cases from the model.

**State-Based Techniques:** Finite state machines have been used to model GUIs [3], [41]. A GUI's state is represented in terms of its windows and widgets; each user event triggers a transition in the FSM. A test case is a sequence of user events and corresponds to a path in the FSM. As is the case for conventional software, FSMs for GUIs also have scaling problems; this is due to the large number of possible states and user events in modern GUIs. Several GUI-domain-specific attempts have been made to handle the scalability issue. For example, Belli [41] converted a GUI FSM into simplified regular expressions. The regular expressions were used to generate event sequences. Shehady *et al.* [2] proposed variable finite state machine (VFSM), which augmented an FSM for a GUI with global variables that can assume a finite number of values during the execution of a test case. The value of each variable is used to determine the next state and output in response to an event.

AI planning has also been used to manage the state-space explosion by eliminating the need for explicit states [26]. A description of the GUI is manually created by a tester; this description is in the form of *planning operators*, which model the preconditions and effects (post-conditions) of each GUI event. Test cases are automatically generated from tasks (pairs of initial and goal states) by invoking a planner which searches for a path from the initial to the goal state.

**Genetic Algorithms:** Test cases have been generated using genetic algorithms to mimic novice users [28]. The approach uses an expert to generate an initial event sequence manually and then uses genetic techniques to generate longer sequences. The assumption is that experts take a direct path when performing a task via the GUI, whereas novice users take longer, indirect paths.

**Directed Graph Models:** In order to reduce manual work, several new systematic techniques based on graph models of the GUI have recently been developed. They are based on *Event Flow Graphs* (EFG) [4] and *Event Interaction Graphs* (EIG). Because of its central role in this paper, we discuss the EIG model in Section 3.

### 2.3 Execution Feedback for Test Case Generation

Execution feedback refers to information that is obtained during test execution and used to guide test case generation. This is called *dynamic test case generation* and, to the best of our knowledge, was originally proposed by Miller and Spooner [14]. In their technique, the software source code is instrumented to obtain execution feedback. The overall test case generation process starts by executing an initial test. The execution feedback is collected and analyzed. The results are used to evaluate the "closeness," according to some criterion, of the execution to the desired outcome; the model used to generate test cases is then modified accordingly; a new test case is generated and executed. This loop stops when the "closeness" evaluation is satisfied.

Since then, several researchers have used the same principle for dynamic test generation.

**Object Properties:** Xie *et al.* [7] have developed a framework that uses feedback in the form of *operational abstractions* (summaries of program run-time state) and object states to generate new test cases. This framework integrates specification-based test generation and dynamic specification inferences. Specification-based test generation is based on formal specifications, which express the desired behavior of a program. However, because formal specifications are difficult to obtain, dynamic specification inference attempts to infer specifications, in the form of operational abstractions, automatically from software execution. Other researchers have also used operational abstractions, combined with symbolic execution, to guide the generation of test cases [6].

**Method-call Sequences:** Pacheco *et al.* [15] have improved random unit test generation by incorporating feedback obtained from executing previous test inputs. They build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. Boyapati *et al.* employ a feedback-based technique to obtain all non-isomorphic inputs (test cases) for a method [12].

**Code Coverage Reports:** All other techniques in this category instrument elements (lines, branches, etc.) of the program code, execute an initial test case/suite, obtain a coverage report that contains the outcomes of conditional statements, and use automated techniques to generate better test cases. The techniques differ in their goals (*e.g.*, cover a specific program path, satisfy condition-decision coverage, cover a specific statement) and their test-case generation algorithms. For example, Miller *et al.* [14] use code coverage and decision outcomes to generate floating-point test data.

Several *iterative techniques* have been used to generate a test case that executes a given program path [9], [10], [13]. The generation is formulated as a function minimization problem. The gradient-descent approach is used to gradually adjust an initial test case so that it executes the given path. Control-flow information in the form of branch-predicate outcomes is collected during software execution.

The *chaining* approach [8] has been used to generate test cases, each to cover a given program statement. An initial test case is executed; the program's control- and data-flow are used to determine whether the test case will lead to the given statement. If not, the branch function of the problematic branch is used to modify the test case. This process continues until the given statement is executed.

*Genetic algorithms* have also been used to automatically generate test suites that satisfy the *condition-decision* adequacy criterion [11], which requires that each condition in the program be true for at least one test case and false for at least one test case. A fitness function is defined for each branch. An initial test suite is obtained and executed. The fitness functions

are used to evaluate the "goodness" of each test case. If a test case covers a new condition-decision, it is considered to be "more fit." The test cases in the gene pool evolve to obtain a new generation of test cases. The process stops until a desired level of fitness is obtained.

Although the techniques discussed in this section are not directly applicable to feedback-directed GUI test case generation, many of the underlying concepts have served as the foundation for this work. For example, execution feedback is used to generate GUI test cases, the EIG model is used to generate the original seed suite, and traversal techniques from model-based testing are used to cover nodes and edges in the ESIG.

## 3 Preliminaries

The feedback-based technique utilizes an abstraction of the GUI's *run-time state* collected and analyzed during the execution of test cases that cover *two-way interactions* between GUI events in order to generate test cases that test *multi-way interactions*. This section defines these terms and introduces notations for subsequent sections.

This work focuses on the class of GUIs that accept discrete events performed by a single user; the events are deterministic, *i.e.*, their outcomes are completely predictable.[2] A GUI in this class is composed of a set $W$ of *widgets* (*e.g.*, buttons, text fields); each widget $w \in W$ has a set $P_w$ of *properties* (*e.g.*, color, size, font). At any time instant, each property $p \in P_w$ has a unique *value* (*e.g.*, red, bold, 16pt); each value is evaluated using a function from the set of the widget's properties to the set of values $V_p$. The *GUI state* at any time instant is a set of triples $(w, p, v)$, where $w \in W, p \in P_w$ and $v \in V_p$, *i.e.*, the observable state of the GUI.

A set of states $S_I$ is called the *valid initial state set* for a particular GUI if the GUI may be in any state $S_i \in S_I$ when it is first invoked. The state of a GUI is not static; events performed on the GUI change its state and hence are modeled as functions that transform one state of the GUI to another.

GUIs contain two types of windows: (1) *modal windows*[3] (*e.g.*, `FileOpen`, `Print`) that, once invoked, monopolize the GUI interaction, restricting the focus of the user to the range of events within the window until explicitly terminated (*e.g.*, using `Ok`, `Cancel`), and (2) *modeless windows* (*e.g.*, `Find/Replace`) that do not restrict the user's focus. If, during an execution of the GUI, modal window $\mathcal{M}_x$ is used to open another modal window $\mathcal{M}_y$, then $\mathcal{M}_x$ is called the *parent* of $\mathcal{M}_y$ for that execution.

The seed test suite is generated using an *event-interaction graph* (EIG) model of the GUI, which is obtained automatically using a standard GUI-reverse-engineering algorithm [4]. The EIG abstraction of the GUI represents only two types of GUI events: *termination* and *system-interaction* events. Termination events close modal windows. Other *structural* events are used to open and close menus and modeless windows, and open modal windows, but are not represented in the EIG (for reasons presented in earlier work [4]). The remaining events, called system-interaction events, do not manipulate the structure of the GUI. Directed edges between nodes encode *execution paths*, *i.e.*, sequences of events, in the GUI. For example, an edge $(e_x, e_y)$ shows that $e_y$ may be executed after $e_x$ along some *execution path*.

The basic motivation behind using a graph model to represent a GUI is that various types of existing graph-traversal algorithms (with well-known run-time complexities) may be used to "walk" the graph, enumerating the events along the visited nodes, thereby generating test cases. In earlier research [4], an algorithm called `GenTestCases` was implemented that returned all possible paths (sequences of events) in the graph bounded to a specific length (number of EIG events) of 2. These length-2 sequences are said to test all *two-way interactions* between the EIG events. This research will generate test cases for *multi-way interactions, i.e.*, longer paths in an EIG. Because EIG nodes do not represent events to open or close menus, or open windows, the sequences obtained from the EIG may not be executable. At execution time, other events needed to reach the EIG events are automatically generated, yielding an executable test case [4]. To allow clean application exit, each test case is also automatically augmented with additional events that close all open modal windows before the test case terminates.

The function notation $S_j = e_x(S_i)$ denotes that $S_j$ is the state resulting from the execution of event $e_x$ in state $S_i$. If $e_1$ and $e_2$ are two different events in a GUI's EIG, ($e_1$, $e_2$) is an edge, and $S_0 \in S_I$ is the initial state of the GUI, then $e_1(S_0)$ is the GUI state after performing $e_1$, $e_2(S_0)$ is the GUI state after performing $e_2$, and $e_2(e_1(S_0))$ is the GUI state after performing the *event sequence* $< e_1; e_2 >$.

## 4 Event Semantic Interaction Graph

The new feedback-based technique is based on the identification of sets of events that need to be tested together in multi-way interactions. We approximate this identification by analyzing feedback from the run-time state of the GUI on an initial test suite. Testing all two-way interactions between all pairs of events is already quite practical with the smoke test suite; we treat this suite as a starting point to collect the feedback. For each smoke test case $< e_1; e_2 >$, we collect states $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$.

Modal windows create special situations due to the presence of termination events. This is because user actions in modal windows do not cause immediate state changes; they typically take effect after a termination event `TERM` has been executed. Hence, each of the states $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$ must be collected after the execution of the termination event `TERM`. Similarly, problems arise when $e_1$ and $e_2$ are in two *different* modal windows; $e_1$ is in a modal window but $e_2$ is in a modeless window; $e_1$ is in a modal window whereas $e_2$ is in its parent window. All these situations require special handling.

Because of the need to precisely define all these situations and for special handling of modal windows, we use formal

2. Testing GUIs that react to temporal and non-deterministic events and those generated by other applications is beyond the scope of this research.

3. Standard GUI terminology, *e.g.*, see http://java.sun.com/products/jlf/ed2/book/HIG.Dialogs.html.

predicates. We first define six[4] cases—as predicates—in one *context, i.e.,* where $e_1$ and $e_2$ are system-interaction events in modeless windows; this situation is called *Context 1*. We will use the notation $\mathcal{P}_{n(m)}(e_1, e_2)$ to represent a predicate for case $n$ in context $m$. We then define two additional contexts; together, the six cases and three contexts yield $6 \times 3 = 18$ situations for computing run-time relationships between events. We note that all these situations are necessary because they capture distinct cases of how an event may influence another's execution.

### 4.1 Widget Modification

We first describe the cases in which an event $e_1$ influences $e_2$ by altering the way $e_2$ modifies a widget's properties.

**Case 1:** There is at least one widget $w$ with property $p$ with initial value $v$ (hence the triple $(w, p, v)$ is in $S_0$), which is not affected by the individual events $e_1$ or $e_2$ (the triple is also in $e_1(S_0)$ and $e_2(S_0)$); however, it is modified when the sequence $< e_1; e_2 >$ is executed, *i.e.,* the value of $w$'s property $p$ changes from $v$ to $v'$.

This can be formally written as $\mathcal{P}_{1(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.\ ((v \neq v') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_2(e_1(S_0))))$. It is quite straightforward to encode such a predicate in a high-level programming language. The implementation would loop through the state triples and stop when one widget satisfying the predicate is detected.

**Case 2:** There is at least one widget $w$ with property $p$ that has an initial value $v$, which is not modified by the event $e_2$; it is modified by $e_1$; however, it is modified differently by the sequence $< e_1; e_2 >$.

**Case 3:** there is at least one widget $w$ with property $p$ that has an initial value $v$, which is not modified by the event $e_1$; it is modified by $e_2$; however, it is modified differently by the sequence $< e_1; e_2 >$. Note that this case is different from Case 2 because the event sequence remains the same, *i.e.,* $e_1$ is executed before $e_2$.

**Case 4:** there is at least one widget $w$ with property $p$ that has an initial value $v$, which is modified by individual events $e_1$ and $e_2$; however, it is modified differently by the sequence $< e_1; e_2 >$.

### 4.2 Widget Creation

The above four cases all handle widgets that persist across the four states being considered, *i.e.,* $S_0$, $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$. In many cases, event execution "creates" new widgets, *e.g.,* by opening menus; the next case handles newly created widgets.

**Case 5:** there is at least one *new* widget $w$ with property $p$ and value $v$ in $e_x(S_0)$, *i.e.,* it was created by event $e_x$ (either $e_1$ or $e_2$) but did not exist in state $S_0$; it was created by the sequence $< e_1; e_2 >$ but with a different value for some property.

4. We have chosen to present only these six cases because we encountered them numerous times in our work on GUI testing. These cases are not exhaustive and we will continue to add new cases, as and when needed, in the future.

### 4.3 Event Availability

A common occurrence of event interaction in GUIs is enabling/disabling widgets, thereby effecting event availability.

**Case 6:** there exists at least one widget $w$ that was disabled in $S_0$ but enabled by $e_1$. Event $e_2$ is performed on $w$; hence $e_1$ makes $e_2$ available for execution.

### 4.4 Additional Contexts

As mentioned earlier, all the six cases were described using Context 1. We now present contexts 2 and 3 and discuss their impact on the cases.

**Context 2:** If both $e_1$ and $e_2$ are associated with widgets that are contained in one modal window with termination event `TERM`, then the definitions of $e_1(S_0)$ , $e_2(S_0)$, and $e_2(e_1(S_0))$ are modified as follows: $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $< e_1; \texttt{TERM} >$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $< e_2; \texttt{TERM} >$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $< e_1; e_2; \texttt{TERM} >$. Note that we can maintain independence between states because the sequences $< e_1; \texttt{TERM} >$, $< e_2; \texttt{TERM} >$, and $< e_1; e_2; \texttt{TERM} >$ are executed as separate smoke tests. Cases 1 through 6 apply, using these modified definitions, for $e_1$ and $e_2$ in the same modal window. The notation used for the predicates when applied in Context 2 is $\mathcal{P}_{n(2)}(e_1, e_2)$, where $n$ is the case number.

**Context 3:** If $e_1$ is associated with a widget contained in a modal window with termination event `TERM`, and $e_2$ is associated with a widget contained in the modal window's *parent* window (*i.e.,* the window that was used to open the modal window) then $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $< e_1; \texttt{TERM} >$, $e_2(S_0)$ is the state of the GUI after the execution of the event $e_2$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $< e_1; \texttt{TERM}; e_2 >$. The notation used for the predicates when applied in Context 3 is $\mathcal{P}_{n(3)}(e_1, e_2)$, where $n$ is the case number.

### 4.5 Event-Semantic Interaction (ESI) Relationship

We are now ready to formally define the ESI relationship. There is an *Event Semantic Interaction* relationship between two events $e_1$ and $e_2$ iff $\mathcal{P}_{1(1)}(e_1, e_2) \vee \mathcal{P}_{2(1)}(e_1, e_2) \vee \ldots \mathcal{P}_{6(1)}(e_1, e_2) \vee \mathcal{P}_{1(2)}(e_1, e_2) \vee \mathcal{P}_{2(2)}(e_1, e_2) \vee \ldots \mathcal{P}_{6(2)}(e_1, e_2) \vee \mathcal{P}_{1(3)}(e_1, e_2) \vee \mathcal{P}_{2(3)}(e_1, e_2) \vee \ldots \mathcal{P}_{6(3)}(e_1, e_2)$. That is, at least one of the predicates in Cases 1 through 6 evaluates to `TRUE` in at least one context; this relationship is written as $e_1 \xrightarrow{n(m)} e_2$, where the number $n$ is one of the case numbers 1 through 6; $m$ is the context number. If multiple cases apply, then one of the case numbers is used. Due to the specific ordering of the events in the sequence $< e_1; e_2 >$, the ESI relationship is not symmetric. Once all of the cases have been implemented, the feedback-based process execution is straightforward. The steps of the execution are as follows.

1) The seed suite consisting of all 2-way interactions $< e_x; e_y >$ between GUI events is executed on the software

in state $S_0$; these test cases are simple enumerations of all EIG edges. All events $e_y$ are also executed in $S_0$. The state information $e_x(S_0)$, $e_y(S_0)$, $e_y(e_x(S_0))$ is collected and stored.

2) The above predicates are evaluated for each pair of system-interaction events in the EIG that are either (1) directly connected by an edge (Context 1) or (2) connected by a path that does not contain any intermediate system-interaction events (contexts 2 and 3), *i.e.*, there is at least one termination event that closes a modal window on this path. If one of the predicates evaluates to TRUE, the two events are ESI-related.

Once all the ESIs in a GUI have been identified, a graph model called the ESI graph (ESIG) is created. The ESIG contains nodes that represent events; a directed edge from node $n_x$ to $n_y$ shows that there is an ESI relationship from the event represented by $n_x$ to the event represented by $n_y$.

### 4.6 Test-Case Generation

The ESIG may be traversed using a modified version of the GenTestCases algorithm discussed in Section 3. The differences are that (1) an ESIG may contain multiple connected components in which case the event sequences are generated for each component separately, and (2) the length of the obtained sequences is now a tunable parameter instead of a fixed number 2. Study 1 in the next section uses values 3, 4, and 5 for this parameter.

Our new implementation of the GenTestCases algorithm is based on the adjacency matrix representation of directed graphs. The key idea used in the implementation is that if we start with a 0/1 adjacency matrix representation of the ESIG, and take that matrix to the $(N-1)^{th}$ power, the $(i, j)$ entry in the resulting matrix is the number of paths of length $N$ from node $i$ to node $j$ (recall that the length is measured in number of nodes encountered along the path). In the trivial case, $N = 2$ will return the input matrix – the $(i, j)$ entry is either 0 or 1, *i.e.*, the number of length 2 paths from node $i$ to node $j$. For $N = 3$, the $(i, j)$ entry in the result matrix is the number of all length 3 paths from node $i$ to $j$.

Because we want to output actual test cases, not just count them, we use a variation of the above approach. The only difference is that instead of just counting the paths, our implementation keeps track of all the actual paths themselves. For this we had to modify the matrix multiplication algorithm and the adjacency matrix representation. The adjacency matrix is modified so that instead of 0/1, the $(i, j)$ entry of the matrix is a list of paths from $i$ to $j$. The matrix multiplication algorithm is modified so that instead of multiplying and adding entries, we concatenate pairs of paths together and union all of them (respectively) to eliminate duplicates. The final matrix entries are paths (*i.e.*, test cases) of specific lengths. We implemented the matrix-based test-case generator using the *Mathematica* package.

## 5 Study 1: Evaluating the Feedback-based Technique on Fielded Applications

The test cases obtained from the modified GenTestCases algorithm can be generated and executed automatically on the GUI. The only unavailable part is the *test oracle*, a mechanism that determines whether an AUT executed correctly for a test case. In this first study, an AUT is considered to have *passed* a test case if it did not "crash" (terminate unexpectedly or throw an uncaught exception) during the test case's execution; otherwise it *failed*. Such crashes may be detected automatically by the script used to execute the test cases. The EIG and ESIG, and their respective test cases may also be obtained automatically. Hence, the entire end-to-end feedback-based GUI testing process for "crash testing" could be executed without human intervention. Note that, in the next section (Study 2), this work is extended by employing a more "powerful" test oracle to detect additional failures.

Implementation of the crash testing process included setting up a database for text-field values. Since the overall process needed to be fully automatic, a database containing one instance for each of the text types in the set {*negative number, real number, long file name, empty string, special characters, zero, existing file name, non-existent file name*} was used. Note that if a text field is encountered in the GUI, one instance for each text type is tried in succession. The overall process was implemented in the GUITAR GUI testing system (http://guitar.sourceforge.net).

This process provided a starting point for a feasibility study to evaluate the ESIG-generated test cases. The following questions needed to be answered to determine the usefulness of the overall feedback-based process:

**S1Q1:** In how many ESI relationships does a given event participate? How many test cases are required to test two-way interactions in an ESIG? How does this number grow for 3-, 4-, ..., n-way interactions?

**S1Q2:** How do the ESIG- and EIG-generated test suites compare in terms of fault-detection effectiveness? Do the former detect faults that were not detected by the latter?

To answer the above questions while minimizing threats to external validity, this study was conducted using four extremely popular GUI-based open-source software (OSS) applications downloaded from SourceForge. The fully-automatic crash testing process was executed on them and the cause (*i.e.*, the *fault*) of each crash in the source code was determined.

**STEP 1: Selection of subject applications.** Four popular GUI-based OSS (CrosswordSage 0.3.5, FreeMind 0.8.0, GanttProject 2.0.1, JMSN 0.9.9b2) were downloaded from SourceForge. These applications have been used in our previous experiments [5]; details of why they were chosen have been presented therein. In summary, all the applications have an active community of developers and a high all-time-activity percentile on SourceForge. Due to their popularity, these applications have undergone quality assurance before release. To further eliminate "obvious" bugs, a static analysis tool called *FindBugs* [42] was executed on all the applications; after the study, we verified that none of our reported bugs were detected by FindBugs.

**STEP 2: Generation of EIGs & seed test suites.** The EIGs of all subject applications were obtained using reverse engineering. In this study, only two-way interactions were tested by the seed test suites. The seed test suites contained 920; 51,316; 29,033; and 4634 test cases for CrosswordSage,

FreeMind, GanttProject, and JMSN, respectively.

**STEP 3: Execution of the seed test suite.** The entire seed suite executed without any human intervention. It executed in 0.39, 30.83, 22.89, and 2.68 hours on CrosswordSage, FreeMind, GanttProject, and JMSN, respectively. In all, 163, 66, 14, and 34 test cases caused crashes; these crashes were caused by 5, 4, 3, and 3 *faults* (as defined earlier) for CrosswordSage, FreeMind, GanttProject, and JMSN, respectively. The GUI's run-time state was recorded during test execution. All faults were fixed in the applications.

Note that debugging and fault-fixing was necessary due to two reasons. First, had we not done so, the longer test cases that we will generate in the next few steps may contain these short test cases as subsequences; the longer tests may hence also crash due to the faults previously detected by the seed suite, yielding no new useful results. Second, this is what would happen in a real situation; a fault will be fixed after it was detected. However, this is a threat to internal validity because an obvious fix in one place may lead to a new fault at another place in the application. To minimize the threat, we reran the seed test suite to ensure the quality of the fixes. Of course, this does not preclude the possibility of introducing faults that are exposed by longer event sequences. To completely eliminate this threat, we later verified that the faults detected by our longer ESIG suites were not caused by these fixes.

**STEP 4: Generation of the ESIG.** The above feedback was used to obtain the ESIs for each application. To address **S1Q1**, the number of ESI relationships in which each event participates is shown in Figure 1. Each event in the GUI has been assigned a unique integer ID; all event IDs are shown on the x-axis. The y-axis shows the number of ESI relationships in which the event participates.

The result shows that certain events dominate (around 25%) the ESI relationship in GUIs. Manual examination of these "dominant" events revealed that the nature of the subject applications, *i.e.*, most of them have a single dominant object (crossword puzzle, mind map, project schedule, messenger window) that are the focus of most events, is such that several key events influence a large number of other events. In the future, we will create a classification of these dominant events. Moreover, several events participate in very few or no ESI relations. These events include parts of the `Help` menu that has no interaction with other application events, and windowing events such as scrolling for which no developer-written code exists.

The ESIs were used to obtain the ESIGs and, subsequently, additional test cases. The number of test cases was 3592, 160,629, 199,127, and 18,144 for CrosswordSage, FreeMind, GanttProject, and JMSN, respectively.

**STEP 5: Execution of the test cases.** To address **S1Q2**, all the newly-generated test cases were executed. The execution lasted for several days. In all, 68, 157, 109, and 20 test cases caused crashes; they were caused by 3, 3, 3, and 1 faults for CrosswordSage, FreeMind, GanttProject, and JMSN, respectively. These faults had not been detected by the two-way test cases. We manually verified that the faults were not introduced by our bug fixes of STEP 4. The result shows that the ESIG-based test cases help to detect faults not detected by earlier techniques.

## 6 Study 2: Digging Deeper via Seeded Faults and In-House Applications

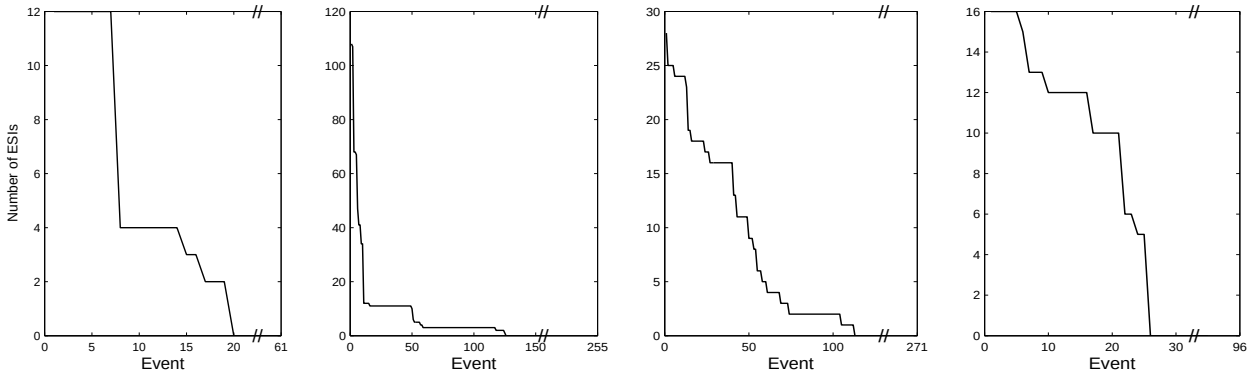
Although the previous study demonstrated the usefulness of the ESIG-based technique, it also raised some important questions. One fundamental question that comes to mind pertains to the cause(s) of the added effectiveness, *i.e., "Is the added effectiveness an incidental side-effect of the events, event interactions, and lines-of-code that the ESI test cases cover and their length; or is it really due to targeted testing of the identified ESI relationships?"* The empirical study presented in this section is designed specifically to address the question of how the fault-detection effectiveness of the suite obtained by the feedback-based technique compare to that of other "similar" suites, where similarity is quantified in terms of statement coverage, event coverage, edge coverage, and size (number of test cases).

This question will be answered by selecting four pre-tested GUI-based applications, and generating and executing 2-way EIG-based and 3-way ESIG-based test suites on them. We will generate additional test suites that are similar to the ESIG-based suite in terms of the aforementioned characteristics and are at least 3-way interacting, and compare their fault-detection effectiveness. Fault detection effectiveness will be measured on a per-test-suite basis in terms of number of faults detected. We will also study the faults, pinpointing reasons for why some of them remain undetected by our technique. Because of space constraints, only select results are presented in this section; the interested reader can find complete results in a technical report [43].

### 6.1 Preparing the Subject Applications & Test Oracles

Four open-source applications, called the TerpOffice suite, consisting of Paint, Present, SpreadSheet and Word, have been selected for the study.[5] Table 1 shows key metrics for TerpOffice. These applications are selected very carefully for a number of reasons. In particular, to minimize threats to external validity, the selected applications are non-trivial, consisting of several GUI windows and widgets. For reasons described later, artificial faults were seeded in the applications – this required access to source code, bug reports, and a CVS development history. To avoid (the often difficult) distinction between GUI code and underlying "business logic," GUI-intensive applications were selected, *i.e.*, most of the source-code implemented the GUI. Finally, the tools implemented for this research, in particular for reverse engineering, are well-tuned for the Java Swing widget library – the applications had to be implemented in Java with a GUI front-end based on Swing components. As is the case with all empirical studies, the choice of subject applications introduces some significant

5. Detailed specifications, requirements documents, source code CVS history, bug reports, and developers' names are available at `http://www.cs.umd.edu/users/atif/TerpOffice/`.




(a) CrosswordSage-0.3.5 (b) FreeMind-0.8.0 (c) GanttProject-2.0.1 (d) JMSN-0.9.9b2

Fig. 1. ESI Distribution in OSS

threats to external validity of the results; these (and other) threats have been noted in Section 7.

TABLE 1
TerpOffice Applications

| Subjects | Windows | Widgets | LOC | Classes | Methods |
|---|---|---|---|---|---|
| Paint | 16 | 301 | 11,803 | 330 | 1,253 |
| Present | 11 | 322 | 10,847 | 292 | 2,057 |
| SpreadSheet | 9 | 176 | 5,381 | 135 | 746 |
| Word | 26 | 617 | 9,917 | 197 | 1,380 |
| TOTAL | 62 | 1,416 | 38,398 | 954 | 5,436 |

For the purpose of this study, a GUI fault is a mismatch, detected by a test oracle, between the "ideal" (or expected) and actual GUI states. Hence, to detect faults, a description of ideal GUI execution state is needed. This description is used by test oracles to detect faults in the subject applications. There are several ways to create this description. First is to manually create a formal GUI specification and use it to automatically create test oracles. Second is to use a capture/replay tool to manually develop assertions corresponding to test oracles and use the assertions as oracles to test other versions of the subject applications. Third is to develop the test oracle from a "golden" version of the subject application and use the oracle to test fault-seeded versions of the application. The first two approaches are extremely labor intensive since they require either the development of a formal specification or the use of manual capture/replay tools; the third approach can be performed automatically and has been used in this study.

Several faults were seeded in each application. In order to avoid fault interaction and to simplify the mapping of application failure to underlying fault, multiple versions of each application were created; each version was seeded with exactly one fault. Hence, a test case detects a fault $i$ if there is a mismatch between version $i$ (*i.e.*, the version that was created by seeding fault $i$) and the original. A mismatch is detected by comparing, between the golden and fault seeded version, the values of all the properties of all the GUI widgets being displayed, after each event.

The process used for fault seeding was similar to the one used in earlier work [4], [44]. Details will not be replicated here. In summary, during fault seeding, 12 classes of known faults were identified, and several instances of each fault class were artificially introduced into the subject program code in source code statements that were covered by the smoke test cases, thereby ensuring that these statements were part of executable code. Care was taken so that the artificially seeded faults were similar to faults that naturally occur in real programs due to mistakes made by developers; the faults were seeded "fairly," *i.e.*, an adequate number of instances of each fault type were seeded. Several graduate students were employed to seed faults in each subject application; they created 263, 265, 234, and 244 faulty versions for Paint, Present, SpreadSheet, and Word, respectively.

### 6.2 Generating and Executing the ESIG-Based Test Suite

The reverse engineering process was used to obtain the EIGs for the original versions of each application. The sizes of the EIGs, in terms of nodes and edges, are shown in Table 2. These numbers are important as they determine the number of generated test cases and their growth in number as test-case length increases.

TABLE 2
ESIG vs. EIG Sizes

| | Paint | Present | SpreadSheet | Word |
|---|---|---|---|---|
| $\#EIG\ Nodes$ | 300 | 321 | 175 | 616 |
| $\#EIG\ Edges$ | 21,391 | 32,299 | 6,782 | 28,538 |
| $\#ESIG\ Nodes$ | 102 | 50 | 45 | 75 |
| $\#ESIG\ Edges$ | 233 | 233 | 197 | 204 |

The EIGs were then used to generate all possible 2-way test cases, *i.e.*, the smoke tests. The numbers generated were exactly equal to the number of edges in the EIGs – it was quite feasible to execute such numbers of test cases in little more than a day on our 50 machines in parallel. The test cases were executed on their corresponding "correct" applications; the GUI state was collected and stored. The reader should note that it is impractical to generate and execute all possible length 3 test cases for these EIGs.

While new software versions were being obtained (via fault seeding as discussed in Section 6.1), the 2-way EIG-based test

TABLE 3
ESIG vs. EIG Fault Detection

| | Paint | Present | SpreadSheet | Word |
|---|---|---|---|---|
| Total Faults | 263 | 265 | 234 | 244 |
| 2-way EIG-detected Faults | 147 | 139 | 139 | 183 |
| 3-way ESIG-detected Faults (only new faults) | 47 | 52 | 39 | 36 |

suites and GUI state were used to obtain all possible 3-way ESIG covering test cases. The sizes of the ESIGs are shown in Table 2. The table shows that the ESIGs are much smaller than the corresponding EIGs. Due to the small number of nodes and edges, the number of 3-way covering test cases was 2531, 2080, 2069, and 2345 for Paint, Present, SpreadSheet, and Word, respectively. As noted earlier, there is a unique set of length 3 test cases for an ESIG; hence, there is a single ESIG test suite per application.

The 2-way EIG- and 3-way ESIG-based test cases were then executed on the fault-seeded versions of the applications. The number of faults detected is shown in Table 3. Note that the last row reports the number of "new" faults detected by ESIG. This table shows that ESIG-based suites are able to detect a large number of faults missed by the EIG.

### 6.3 Developing "Similar" Suites

As mentioned earlier, this study required the development of several new test suites. To minimize threats to validity, the suites needed to satisfy a number of requirements, discussed next.

From previous studies, we know that statement, event, and EIG-edge coverage, and size (number of test cases) play an important role in the fault-detection effectiveness of a test suite [44]. For example, a small test suite that covers few lines of code will most likely detect fewer faults than another larger suite that covers many more lines. To allow fair comparison of fault-detection effectiveness, we needed test suites that have the *same statement, event, and edge coverage, and size (number of test cases)* as that of ESIG-based test suites.

Previous studies have also shown that long test cases (number of EIG events) fare better than short ones in terms of the number of faults that they detect [5]. Because we did not want the new suites to have any disadvantage, we ensured that all their test cases had at least 3 EIG events (note that all our ESIG test cases have exactly 3 ESIG/EIG events).

It is non-trivial to generate these test suites. For example, consider the problem of generating a GUI test suite that covers specific lines of code. Because of the different levels of abstraction between GUI events and code, one would need to manually examine the source code, the relationship between events and underlying code, and carefully tailor each event in every test case to ensure that it covers a specific line. Because there are no automated techniques to do this task, the process will be very resource intensive.

Moreover, because the above criteria (same statement, event, and edge coverage, and size) may be met by a large number of test suites (with varying fault-detection effectiveness), the process of generating different suites and comparing them to the ESIG-based suites needed to be repeated several times. In this study, we generated 700 "similar" test suites per application and compared their fault-detection effectiveness to the ESIG suite.

GUI test cases are expensive to execute – each test case can take up to 2 minutes to execute (on average, each requires 30 seconds). Our 700 suites each for Paint, Present, SpreadSheet, and Word, contained 1,054,064; 860,324; 850,808; and 974,235 test cases, respectively; in all 3,739,431 test cases. Each test case needed to be run on each fault-seeded version; this task would have taken several years on our 50-machine cluster – clearly impractical. Other researchers, who have also encountered similar issues of practicality, have circumvented this problem by creating a *test pool* consisting of a large number of test cases that can be executed in a reasonable amount of time [45]. Each test case in the pool is executed only once and it's execution attributes *e.g.*, time to execute and faults detected are recorded. Multiple test suites are created by carefully selecting test cases from this pool. The execution of these suites is "simulated" by combining the attributes of constituent test cases using appropriate functions (*e.g.*, *set union* for faults detected). This research will also employ the test pool approach to create a large number of test suites. The test-pool-based approach will introduce some threats to validity, which we will note in Section 7.

Finally, we did not want to introduce any human bias when generating these test cases. We used a randomized guided mechanical process. A related approach was employed by Rothermel *et al.* [46] to create sequences of commands to test command-based software. In their approach, each command was executed in isolation and test cases were "assembled" by concatenating commands together in different permutations. Since GUI events (commands) enable/disable each other, most arbitrary permutations result in unexecutable sequences. Hence, we used the EIG model to obtain only executable sequences.

We generated test cases in batches of increasing lengths, measured in terms of the number of EIG events. We required that each EIG edge be covered by at least $N$ test cases of a particular batch. Moreover, we required that each fault-seeded statement be covered by at least $M$ test cases of the overall pool. The test-case generation process started by generating (using a process described in the next paragraph) the batch of length-3 test cases until each EIG edge was covered by at least $N$ test cases; they were all executed and their statement coverage was evaluated; the next (and all subsequent) batch was generated ONLY IF each fault-seeded statement was not yet covered by at least $M$ test cases.

The process of generating each batch of length $i$ test cases uses the following algorithm:

1) Initialize a `frequency` variable for each EIG edge to $zero$.
2) For each event $e_x$ in the EIG, do
   a) Add the single event $e_x$ to a new empty test case $t$.
   b) Form a list of all outgoing edges from $e_x$.
   c) Select the edge ($e_x$, $e_y$) that has the lowest `frequency`, breaking ties via random selection.

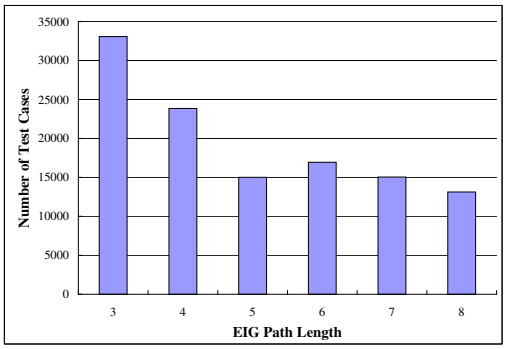



Fig. 2. Histogram of Test Case Lengths in Pool (Paint)

Add $e_y$ to the test case $t$.

d) Follow the selected edge to its destination event $e_y$.

e) Starting at $e_y$, recurse the `frequency`-based selection and follow-the-edge process (described in Steps 2b through 2d and this recursive step) until the desired length is obtained, adding events into the test case $t$.

3) Add the test case $t$ to the suite.

4) If all EIG events have been covered and all `frequency` $\geq N$, stop; otherwise go to the next EIG event (via the iteration of Step 2 above).

The above algorithm was guaranteed to stop because all faults had been seeded in lines that were executable by the smoke tests; the count for each statement would ultimately reach $M$ and stop. Finally, all the ESIG-based test cases were added to the pool.

In this study, we set $N = 10$ and $M = 15$. This choice was dictated by the availability of resources. As described earlier, all the test cases needed to be executed on the fault-seeded versions of their respective application. Even with 50 machines running the test cases in parallel, the entire process took over four months.

The total number of test cases per application is shown in Column 2 of Table 4. The length distribution of Paint's[6] test cases is shown as a histogram in Figure 2. As expected, longer tests were able to cover more EIG edges than the short ones; hence fewer long test cases were needed to satisfy our coverage requirements.

After all the runs had completed, we had several matrices per application: (1) the fault matrix, which summarized the faults detected per test case and (2) for each coverage criterion (event, edge, statement), a coverage matrix, which summarized the coverage elements covered per test case.

This test pool was then used to obtain coverage-adequate suites. For example, event-adequate suites were obtained by maintaining sets of test cases that covered each ESIG event. Test cases were selected randomly without replacement from each set and duplicates eliminated, ensuring that each event was covered by the resulting suite. A similar process was used for edge and statement coverage. The process was repeated

TABLE 4
Test Pool and Average-Suite Sizes

| | Test Pool | $T_E$ Event | $T_I$ Edge | $T_S$ Stmt. | $T_R$, $T_{R+E}$ $T_{R+I}$, $T_{R+S}$ |
|---|---|---|---|---|---|
| Paint | 119,583 | 103 | 190 | 123.64 | 2531 |
| Present | 231,680 | 50 | 264 | 18.24 | 2080 |
| SpreadSheet | 191,966 | 45 | 173 | 14.08 | 2069 |
| Word | 192,042 | 84 | 248 | 30.35 | 2345 |

100 times to yield 100 suites. The average size of the suites is shown in Columns 3–5 of Table 4.

Finally, $T_R$ was constructed using random selection without replacement ensuring that the final size of $T_R$ was the same as that of the ESIG suite. A total of 100 such suites per application were obtained. Similarly, each of the suites $T_E$, $T_I$, $T_S$ were augmented with additional test cases, selected without replacement at random from the pool to yield $T_{R+E}$, $T_{R+I}$, $T_{R+S}$, respectively. The sizes of all these suites was equal to the size of the ESIG suite. Finally, 100 more suites that shared *all* the characteristics of interest in this study (*i.e.*, event, edge, statement, and size) with the ESIG suite were constructed; the symbol $T_{R+E+I+S}$ will be used for these suites.

Note that the fault-detection effectiveness of each test suite can be obtained directly from the fault matrix of the test pool without rerunning the test cases. The results are shown in Figure 3 as distributions. The box-plots provide a concise display of each distribution, each consisting of 100 data points. The line inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover 90% of the distribution; outliers are shown as points beyond the whiskers. Visual inspection of the plots shows that the fault-detection effectiveness of the ESIG-generated test suite (shown as an asterisk) is better than that of most individual similar-coverage and similar-sized suites. Some suites that lie in the whiskers and outliers do detect more faults than the ESIG suite. However, we remind the reader that unlike the ESIG suite, there is no systematic and automatic way to generate these suites.

As demonstrated above, box-plots are useful to get an overview of data distributions. However, valuable information is lost in creating the abstraction. For example, it is not clear *how many* test suites detect specific numbers of faults; we are especially interested in the number of suites that do better than the ESIG suites. This is important to partially understand our EIG and ESIG suites. We now show the number of test suites that detected specific numbers of faults. Figure 4 shows eight histograms for Paint,[7] one for each box in the box-plot. The x-axis represents the number of faults; the y-axis shows the number of test suites that detected the particular number of faults. To allow easy visual comparison, we have used the same x-axis and y-axis scales for all eight plots. The vertical dotted line represents the number of faults detected by the ESIG suites.

For all applications, $T_E$, $T_I$, $T_S$, $T_R$, $T_{R+E}$ consistently did worse than the ESIG suites. For a small percentage of

6. Similar histograms are presented in [43].

7. Plots for other applications are shown in [43].

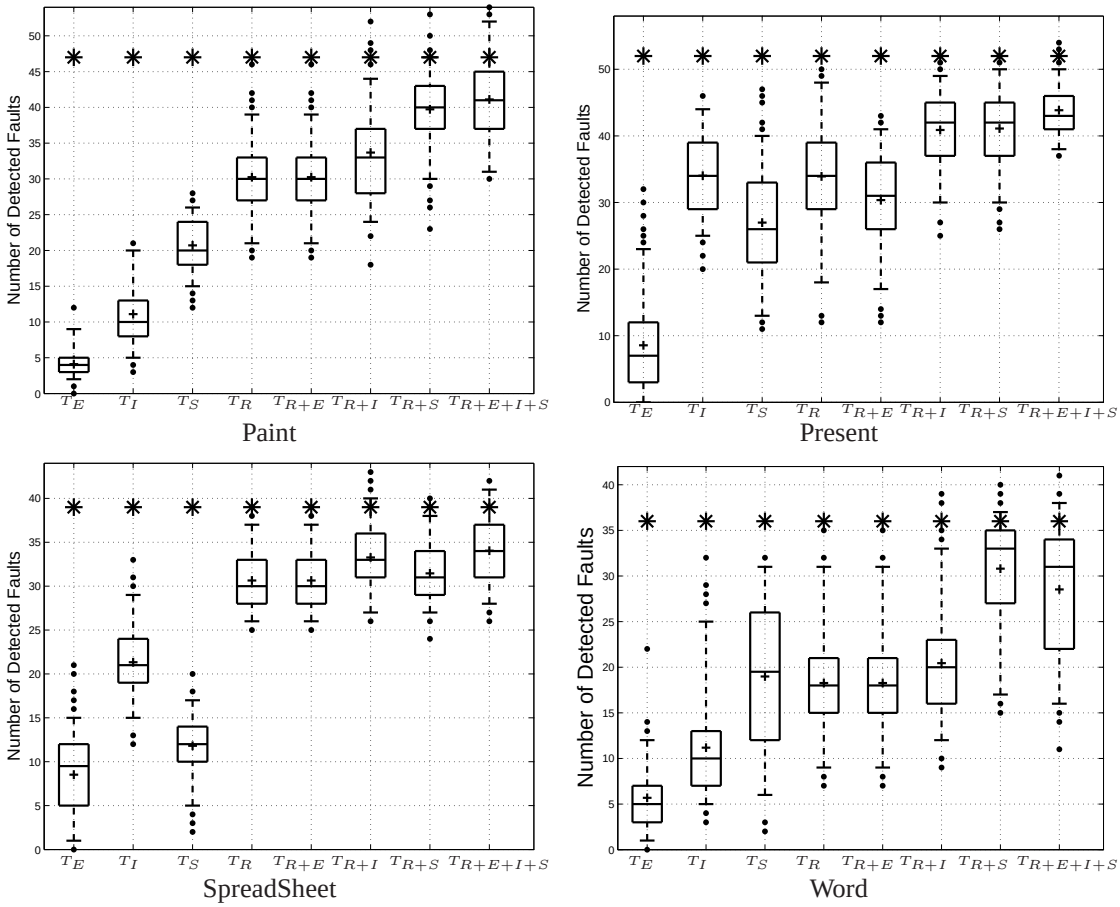



Fig. 3. Fault Detection Distribution

TABLE 5
Percentage of Suites that Outperformed ESIG

| Test Suite | Paint | Present | SpreadSheet | Word |
|---|---|---|---|---|
| $T_{R+I}$ | 4 | 0 | 6 | 2 |
| $T_{R+S}$ | 4 | 0 | 2 | 11 |
| $T_{R+E+I+S}$ | 15 | 2 | 8 | 9 |

test suites (shown in Table 5), $T_{R+I}$, $T_{R+S}$, and $T_{R+E+I+S}$ did better than the ESIG suites. It should be noted that the ESIG approach does better than most test suites considered in this study. And that the ESIG-based approach is the only fully automatic approach to generate the GUI test suites; all other suites were obtained from the test pool *after* they had been executed and statement coverage obtained. Moreover, the performance of the $T_R$ test suites indicates that the size of a suite plays a very important role in fault-detection. We study this issue in the next section.

The results discussed thus far have been based on visual examination of the data. We now want to determine whether the differences in the number of faults across coverage criteria are statistically significant. In particular, we want to study the differences between the ESIG suite (a single value "number of faults detected") and all the other "similar" suites (100 values per distribution).

For illustration, the solid line superimposed on the histograms (Figure 4) shows the normal distribution approximation; informally, the data seems to follow the normal distribution quite well. We also confirmed normality by using QQ-plots [43]. Based on normality, we use a *single sample t-test* to test for the significance of the difference between the observed fault-detection of the ESIG suite and the mean of each distribution. The null hypothesis is that the two values are equal; the alternate hypothesis is that they are unequal. Note that a separate test is needed per pair (mean of each distribution, fault-detection of the ESIG suite value). The resulting p-values were all more than 0.99. Hence we reject the null hypothesis and accept the alternate hypothesis; there is a significant difference between fault-detection of the ESIG suite and the mean fault-detection of each of the "similar" suites. The ESIG suites are better at detecting faults compared to same-sized test suites that cover essentially the same events, edges, and statements. This result helps to answer the primary question raised in this study.

### 6.4 Discussion

We now present details of why the ESIG-based test suites were able to detect more faults than other test suites. We specifically looked at three related issues: reachability, manifestation, and number of test cases. We note that the first two issues are

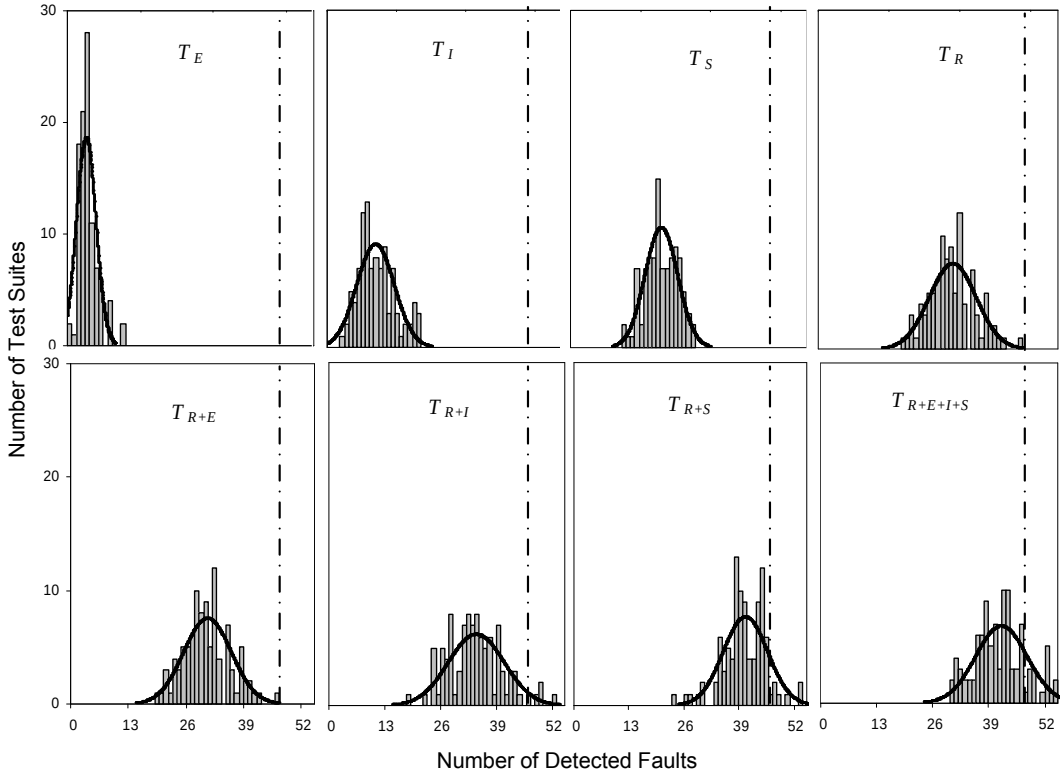



Fig. 4. Histogram for Paint

related to the RELAY model [47] of how a fault causes a failure on the execution of some test. We define *reachability* as the coverage of the statement in which a fault was seeded, and *manifestation* as the fault leading to a failure so that our GUI-based test oracle could detect it. As observed in [47], both are necessary conditions for fault detection. The data in Figure 3 indicates that the ESIG-based test suites were able to outperform their coverage-adequate equivalent counterparts. Hence, they must have been more successful, than their counterparts, at the combination of reachability and manifestation of several faults. We feel that this behavior is due to the nature of the ESI relationship, which is based on observed GUI state, and hence the software's output. Executing tests that focus only on ESI events increases the likelihood that a fault will be revealed on the GUI, and hence detected by our GUI-based test oracle. Although we will not attempt to analyze this behavior in great detail here (indeed this is a direction for future research), we will provide some quantitative data to show evidence of this phenomenon by studying the test cases in the pool.

Figure 5 shows two tables, one for Paint and the other for Word. Each table has three columns (to save space, the columns are wrapped): Column 1 ($ID$) is the fault ID; Column 2 ($f$) is the number of test cases that detected the fault as well as executed the statement in which the fault was seeded; Column 3 ($\sim f$) is the number of test cases that did not detect the fault even though each executed the statement in which the fault was seeded; the fault ID is shaded if at least one ESI test case detected it. For example, the statement that contained Fault 21 of Paint was executed by a total of 845+11 test cases (marked with an oval), of which 11 detected it; the fault was detected by at least one ESI test case. On the other hand, Fault 127 of Paint was not detected by any ESI test case; it was however detected by 42 of the total 42+267 test cases. Hence, a statement-coverage adequate test suite would have a probability of $\frac{42}{42+267}$ of detecting this fault (assuming that faults are independent). The data in Figure 5 is in fact sorted by this probability, giving us a sense of the "hardness" of the fault for statement-coverage adequate test suites. This data helps us to better interpret the results of Figure 3. First, the ESIG test suites did detect many of the seeded faults. Second, they did better than $T_S$ because they detected many of the "hard" faults (this was most apparent in Paint and SpreadSheet [43]). Third, some faults were detected by many of the test cases that executed the statement. For example, Fault 136 in Paint was detected by 747 of the total 747+446 test cases that executed the statement in which it was seeded. The fault was seeded in the handler of a termination event that closes the `Attributes` window and applies the attributes (if any have changed) to the current image on the main canvas. The seeded fault caused the image size to be computed incorrectly, resulting in an incorrectly sized image whenever at least one attribute in the `Attributes` window is modified via the GUI. Because there are many different ways of modifying the attributes, a large number of test cases are able to detect this fault (12 in the ESIG test suite and 735 in the rest of the test pool). In general, statement coverage adequate test suites do really well for these types of faults that can be triggered in many different ways.

Fourth, several faults were detected by very few test cases.

**Paint**

| ID | f | ~f | ID | f | ~f | ID | f | ~f | ID | f | ~f | ID | f | ~f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **158** | 1 | 1247 | **22** | 9 | 796 | 75 | 15 | 707 | **245** | 3 | 115 | 73 | 28 | 703 |
| **159** | 1 | 1173 | 77 | 9 | 735 | 92 | 15 | 698 | **248** | 3 | 115 | 72 | 31 | 744 |
| **65** | 3 | 721 | 52 | 4 | 311 | 71 | 15 | 696 | **255** | 3 | 115 | 44 | 15 | 339 |
| **36** | 4 | 86[illegible] | **21** | 11 | 845 | 99 | 17 | 769 | **256** | 3 | 115 | 86 | 11 | 180 |
| **52** | 5 | 830 | 52 | 13 | 874 | **125** | 2 | 90 | 261 | 3 | 115 | 14 | 1 | 15 |
| **64** | 5 | 772 | 56 | 5 | 331 | **126** | 2 | 90 | **262** | 3 | 115 | 120 | 15 | 220 |
| 33 | 6 | 856 | **19** | 13 | 850 | 84 | 17 | 762 | 42 | 10 | 381 | **32** | 10 | 122 |
| **34** | 7 | 843 | **20** | 14 | 891 | **48** | 16 | 717 | **47** | 21 | 762 | **253** | 9 | 109 |
| 30 | 6 | 722 | **177** | 2 | 125 | 80 | 16 | 714 | 46 | 10 | 359 | 119 | 19 | 216 |
| **179** | 1 | 120 | 96 | 12 | 711 | 94 | 16 | 705 | 93 | 21 | 748 | **259** | 11 | 107 |
| 180 | 1 | 118 | 55 | 6 | 354 | 83 | 16 | 704 | 98 | 21 | 730 | 59 | 33 | 314 |
| **246** | 1 | 117 | 100 | 12 | 704 | 82 | 17 | 743 | 68 | 21 | 729 | 58 | 36 | 337 |
| 249 | 1 | 117 | **247** | 2 | 116 | 79 | 17 | 719 | 54 | 11 | 379 | 123 | 25 | 207 |
| **191** | 1 | 116 | **250** | 2 | 116 | 88 | 1 | 42 | 45 | 12 | 383 | 123 | 30 | 270 |
| **192** | 1 | 116 | **260** | 2 | 116 | 81 | 16 | 664 | 30 | 4 | 120 | 127 | 42 | 267 |
| 89 | 7 | 773 | 76 | 14 | 781 | 70 | 18 | 746 | 31 | 4 | 127 | 134 | 35 | 273 |
| **211** | 1 | 110 | 26 | 2 | 108 | 91 | 18 | 738 | 43 | 12 | 373 | **178** | 28 | 99 |
| **212** | 1 | 110 | **18** | 18 | 920 | **251** | 2 | 82 | 67 | 23 | 707 | **229** | 28 | 98 |
| **50** | 10 | 934 | 66 | 15 | 764 | **252** | 2 | 82 | **124** | 3 | 89 | 265 | 35 | 272 |
| 101 | 24 | 2206 | 69 | 14 | 688 | **17** | 24 | 978 | 97 | 27 | 770 | **136** | 747 | 446 |
| 102 | 23 | 2093 | 37 | 17 | 834 | 78 | 19 | 762 | 41 | 15 | 402 | 137 | 35 | 260 |
| 85 | 8 | 720 | 74 | 16 | 755 | 95 | 19 | 761 | 87 | 5 | 127 | | | |

**Word**

| ID | f | ~f | ID | f | ~f |
|---|---|---|---|---|---|
| **156** | 126 | 2336 | **81** | 8 | 61 |
| **209** | 68 | 1238 | **85** | 8 | 61 |
| **157** | 131 | 2331 | **25** | 10 | 72 |
| **158** | 130 | 2146 | **29** | 10 | 72 |
| **88** | 4 | 65 | **114** | 12 | 72 |
| **93** | 4 | 65 | **57** | 11 | 58 |
| **210** | 74 | 1185 | **97** | 11 | 58 |
| **211** | 74 | 1185 | **116** | 14 | 70 |
| **212** | 70 | 1072 | **117** | 17 | 65 |
| **155** | 167 | 2295 | **118** | 20 | 64 |
| **159** | 175 | 2101 | **120** | 21 | 62 |
| **102** | 5 | 57 | **121** | 21 | 62 |
| **30** | 7 | 75 | **160** | 54 | 153 |
| **66** | 6 | 63 | **115** | 27 | 57 |
| **67** | 6 | 63 | **119** | 30 | 54 |
| **79** | 6 | 63 | **32** | 76 | 100 |
| **80** | 6 | 63 | 6 | 122 | 129 |
| **83** | 6 | 63 | **24** | 56 | 28 |
| **101** | 6 | 56 | 122 | 60 | 24 |
| **91** | 7 | 62 | | | |
| **76** | 7 | 55 | | | |
| **103** | 7 | 55 | | | |

Fig. 5. Test Cases Covered Faulty Statements and Their Fault Detection.

Fault 34 of Paint is an example. This fault flips the conditional statement in an event handler of a type of curve tool. The condition is to check whether the curve tool is currently selected; if yes, then the curve stroke is set according to the selected line type for the curve tool. Due to the fault, the curve stroke is incorrectly set, resulting in an incorrect image to be drawn on the main canvas only when the event sequence: $< SelectCurveTool; SelectLineType; DrawOnCanvas >$ is executed. If the first two events are not executed, then $DrawOnCanvas$ does not trigger the fault even if the statement containing the seeded fault is executed. Hence, although there are many test cases that cover the faulty statement (843+7=850), only 7 test cases in the test pool detected the fault. One of them is the ESI test case; ESI-relationships were found between the three events. In general, statement coverage adequate test suites do not do well for faults that are executed by many event sequences but manifest as failures in very few cases.

The size of a suite seems to play a very important role in fault-detection. Indeed, the $T_R$ test suites, which were the same size as the ESIG-based suites, did better (in most cases) than their coverage-adequate counterparts. We feel that this behavior is an artifact of the density of our fault matrices. A large number of test cases are successful at detecting many "easy" faults. Even if test cases are selected at random, given adequate numbers, they will be able to detect a large number of these easy faults. For example, given 192,042 test cases in the pool for Word and the size of $T_R$ being 2345, the probability that Fault 24, which is executed by 84 test cases, is detected by at least one test case in $T_S$ is 0.49 – this is quite high. In Figure 6, we show the probability that a random suite of its corresponding ESI-suite size would detect a particular fault. The figure shows two tables for Word and Paint. Each table has two (wrapped) columns. Column 1 ($ID$) is the fault ID; Column 2 ($p$) is the probability that the fault is detected by at least one test case in $T_R$, which is the same size as the

**Paint**

| ID | p | ID | p | ID | p | ID | p | ID | p |
|---|---|---|---|---|---|---|---|---|---|
| **158** | 0.02 | **245** | 0.06 | 77 | 0.18 | 71 | 0.27 | 93 | 0.36 |
| **159** | 0.02 | **248** | 0.06 | **253** | 0.18 | 41 | 0.27 | 98 | 0.36 |
| **179** | 0.02 | **255** | 0.06 | **50** | 0.19 | 44 | 0.27 | 68 | 0.36 |
| 180 | 0.02 | **256** | 0.06 | 42 | 0.19 | 120 | 0.27 | 102 | 0.39 |
| **246** | 0.02 | 261 | 0.06 | 46 | 0.19 | 74 | 0.29 | 67 | 0.39 |
| 249 | 0.02 | **262** | 0.06 | **32** | 0.19 | **48** | 0.29 | 101 | 0.40 |
| **191** | 0.02 | **124** | 0.06 | **21** | 0.21 | 80 | 0.29 | **17** | 0.40 |
| **192** | 0.02 | **36** | 0.08 | 54 | 0.21 | 94 | 0.29 | 123 | 0.41 |
| **211** | 0.02 | 57 | 0.08 | 86 | 0.21 | 83 | 0.29 | 97 | 0.44 |
| **212** | 0.02 | 30 | 0.08 | **259** | 0.21 | 81 | 0.29 | 73 | 0.45 |
| 88 | 0.02 | 31 | 0.08 | 96 | 0.23 | 37 | 0.30 | **178** | 0.45 |
| 14 | 0.02 | **52** | 0.10 | 100 | 0.23 | 99 | 0.30 | **229** | 0.45 |
| **177** | 0.04 | **64** | 0.10 | 45 | 0.23 | 84 | 0.30 | 72 | 0.48 |
| **247** | 0.04 | 56 | 0.10 | 43 | 0.23 | 82 | 0.30 | 59 | 0.51 |
| **250** | 0.04 | 87 | 0.10 | **51** | 0.24 | 79 | 0.30 | 58 | 0.54 |
| **260** | 0.04 | **33** | 0.12 | **19** | 0.24 | **18** | 0.32 | 128 | 0.57 |
| 26 | 0.04 | 90 | 0.12 | **20** | 0.26 | 70 | 0.32 | 127 | 0.59 |
| **125** | 0.04 | 55 | 0.12 | 76 | 0.26 | 91 | 0.32 | **200** | 1.00 |
| **126** | 0.04 | **34** | 0.14 | 69 | 0.26 | 78 | 0.33 | **137** | 1.00 |
| **251** | 0.04 | 89 | 0.14 | 66 | 0.27 | 95 | 0.33 | **136** | 1.00 |
| **252** | 0.04 | 85 | 0.16 | 75 | 0.27 | 119 | 0.33 | **134** | 1.00 |
| **65** | 0.06 | **22** | 0.18 | 92 | 0.27 | **47** | 0.36 | | |

**Word**

| ID | p | ID | p |
|---|---|---|---|
| **88** | 0.05 | **118** | 0.22 |
| **93** | 0.05 | **120** | 0.23 |
| **102** | 0.06 | **121** | 0.23 |
| **66** | 0.07 | **115** | 0.28 |
| **67** | 0.07 | **119** | 0.31 |
| **79** | 0.07 | 160 | 0.48 |
| **80** | 0.07 | **24** | 0.49 |
| **83** | 0.07 | 122 | 0.52 |
| **101** | 0.07 | **209** | 0.57 |
| **30** | 0.08 | **212** | 0.58 |
| **91** | 0.08 | **210** | 0.60 |
| **76** | 0.08 | **211** | 0.60 |
| **103** | 0.08 | **32** | 0.61 |
| **81** | 0.09 | **6** | 0.78 |
| **85** | 0.09 | **156** | 0.79 |
| **25** | 0.12 | **158** | 0.80 |
| **29** | 0.12 | **157** | 0.80 |
| **57** | 0.13 | **155** | 0.87 |
| **97** | 0.13 | **159** | 0.88 |
| **114** | 0.14 | | |
| **116** | 0.16 | | |
| **117** | 0.19 | | |

**> 0.25**

Fig. 6. Probability of Detecting Faults by Random Test Cases.

ESIG suite. This data shows that many of these difficult faults are detected by at least one ESIG-based test case, improving their fault-detection effectiveness. Moreover, 16 faults, marked in the figure, in Word have a detection probability of more than 0.25. This number is much larger for the other three applications, helping to understand why $T_R$ and the other suites that included randomly selected test cases did so well.

Finally, we wanted to examine why some of the faults were not detected. We manually examined each fault and tried to manually devise ways of manifesting the fault as a failure. We determined that:

1) several of the faults were in fact manifested as failures on the GUI but our automated test oracle was not capable of examining these parts of the GUI,
2) few faults caused failures in non-GUI output, which we

TABLE 6
Undetected Faults Classification

| | Ignored Widget Properties | Non-GUI Failure | Longer Sequence | Masked Error | Crash | Total |
|---|---|---|---|---|---|---|
| Paint | 0 | 0 | 1 | 6 | 0 | 7 |
| Present | 13 | 0 | 4 | 0 | 0 | 17 |
| SpreadSheet | 9 | 2 | 8 | 5 | 3 | 27 |
| Word | 5 | 0 | 4 | 11 | 0 | 20 |

could not detect,
3) several of the undetected faults require even longer sequences,
4) the effect of several faults was masked by the event handler code even before our test oracle could detect it,
5) some faults crashed their corresponding fault-seeded version.

We show the numbers of these faults in Table 6. The large number of "Ignored Widget Properties" faults has prompted us to improve our test oracles for future work.

This controlled study showed that the automatically identified ESI relationships between events generate test suites that detect more faults than their code-, event-, and event-interaction-coverage equivalent counterparts. Moreover, we saw that several of our missed faults remained undetected because of limitations with our automated GUI-based test oracle, and others required even longer sequences.

## 7 Conclusions and Future Work

This paper presented a new fully automatic technique to test multi-way interactions among GUI events. The technique is based on analysis of feedback obtained from the run-time state of GUI widgets. A seed test suite is used for feedback collection. The technique was demonstrated via two independent studies on eight software applications. The results of the first study showed that the test cases generated using the feedback were useful at detecting serious and relevant faults in the applications. The second study compared the ESIG-based test suite to similar EIG-based suites. It showed that the added effectiveness is due to targeted testing of the identified ESI relationships, not an incidental side-effect of the size of the suite, nor the additional events and code that it covers.

As is the case with all research involving empirical studies, these results are subject to threats to validity. First is the selection of subject applications and their characteristics. The results may vary for applications that have a complex back-end, are not developed using the object-oriented paradigm, or have non-deterministic behavior. Second, in study 2, the test pool approach was used due to practical limitations. It is expected that the repetition of the same test case across multiple test suites will have an impact on some of the results. The algorithm used to create the test pool ensures that each event (the first event in the test case) is executed in a known initial state; the choice of this state may have an effect on the results. Third, the Java API allow the extraction of only 12 properties of each widget; only these properties were used to obtain the ESI relationship via GUI state; moreover, faults are reported for mismatches between these 12 properties. Fourth, we used one technique to generate test cases – based on event-interaction graphs. Other techniques, *e.g.*, using capture/replay tools and programming the test cases manually may produce different types of test cases, which may show different execution behavior. Fifth, in Study 2, a threat is related to our measurement of fault detection effectiveness; each fault was seeded and activated individually. Note that multiple faults present simultaneously can lead to more complex scenarios that include fault masking. Finally, several threats are related to fault seeding in Study 2. Threats from issues such as human decision-making are minimized by using an objective technique for uniformly distributing faults based on functional units.

This research has presented several exciting opportunities for future work. In the immediate future, the three contexts for the cases will be simplified and, if possible, combined. The current special treatment of termination events, which led to an additional two contexts, will be revised. One possibility is the revision of the EIG model; the elimination of all termination events from this model will be explored. This revision will also lead to the definition of new, fundamentally different cases for the ESI relationship.

The results showed that certain events in the GUI dominate the ESI relationship. These events will be studied and classified. In the future, additional GUI applications and software problems will be studied. The run-time state information was collected using the Java Swing API for standard Swing widgets. Future work involves incorporating customized API for application-specific widgets into feedback collection and analysis.

The analysis summarized in Section 5 led to a deeper understanding of the relationship between real GUI events and the underlying code in fielded GUI applications. This may lead to new techniques that combine dynamic analysis of the GUI and static analysis of the event handler code. For example, the code for related events may be given to a static-analysis engine that could examine the code for possible interactions that are only apparent at the code level, *e.g.*, data-flow relationships.

The feedback currently obtained at run time is in the form of GUI widgets. Mechanisms, such as reflection, in modern programming languages may be used to obtain additional feedback from non-GUI objects. The definition of state, in terms of a set of objects with properties and values, is general; it may be applied to any executing object. Some of the six cases may be adapted for non-GUI objects. Another straightforward way to enhance the feedback is to instrument the software for code coverage and run-time invariant collection. This feedback may be used to generate new types of test cases. Another logical extension of this work is to examine the redundancy in our ESIG suites via existing test minimization techniques developed for user interfaces [48].

Some of the challenges of GUI testing are also relevant to testing of event-driven software, *e.g.*, web applications and object-oriented software. One way to test these classes of software is to generate test cases that are sequences of events (*e.g.*, web user actions or method calls). Some of the

techniques developed in this research have already been used by other researchers to prune the space of all possible event interactions to be tested for web applications [49]; similar extensions will be explored for object-oriented software.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers whose comments and suggestions helped to extend the second 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

[1] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[2] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *FTCS '97: Proceedings of the 27th International Symposium on Fault -Tolerant Computing (FTCS '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 80.

[3] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *ISSRE '00: Proceedings of the 11th International Symposium on Soft ware Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000, p. 110.

[4] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 884–896, 2005.

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

[6] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[7] T. Xie and D. Notkin, "Tool-assisted unit-test generation and selection based on operational abstractions." *Autom. Softw. Eng.*, vol. 13, no. 3, pp. 345–371, 2006.

[8] R. Ferguson and B. Korel, "The chaining approach for software test data generation." *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[9] M. J. Gallagher and V. L. Narasimhan, "Adtest: A test data generation suite for Ada software systems." *IEEE Trans. Software Eng.*, vol. 23, no. 8, pp. 473–484, 1997.

[10] B. Korel, "Automated software test data generation." *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[11] C. C. Michael, G. McGraw, and M. Schatz, "Generating software test data by evolution." *IEEE Trans. Software Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.

[12] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates." in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002, pp. 123–133.

[13] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method." in *SIGSOFT FSE*, 1998, pp. 231–244.

[14] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data." *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 223–226, 1976.

[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, May 23–25, 2007, pp. 396–405.

[16] B. A. Myers and M. B. Rosson, "Survey on user interface programming." in *CHI*, 1992, pp. 195–202.

[17] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles," *Softw. Test. Verif. Reliab.*, vol. 16, no. 1, pp. 3–32, 2006.

[18] J. M. Clarke, "Automated test generation from a behavioral model," in *Proceedings of Pacific Northwest Software Quality Conference*. Portland, OR: Pnsqc/Pacific Agenda, May 1998.

[19] S. Esmelioglu and L. Apfelbaum, "Automated test generation, execution, and reporting," in *Proceedings of Pacific Northwest Software Quality Conference*. Portland, OR: Pnsqc/Pacific Agenda, Oct 1997, pp. 127–142.

[20] P. J. Bernhard, "A reduced test suite for protocol conformance testing," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 3, pp. 201–220, Jul. 1994.

[21] W.-H. Chen, C.-S. Lu, E. R. Brozovsky, and J.-T. Wang, "An optimization technique for protocol conformance testing using multiple uio sequences," *Inf. Process. Lett.*, vol. 36, no. 1, pp. 7–11, 1990.

[22] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178–187, 1978.

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

[24] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, Jul. 1990.

[25] M. Auguston, J. B. Michael, and M.-T. Shing, "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*. New York, NY, USA: ACM Press, 2005, pp. 1–6.

[26] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, 2001.

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

[28] D. J. Kasik and H. G. George, "Toward automatic generation of novice user test scripts," in *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*. New York: ACM Press, 13–18 Apr. 1996, pp. 244–251.

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

[30] B. Sarikaya, "Conformance testing: architectures and test sequences," *Comput. Netw. ISDN Syst.*, vol. 17, no. 2, pp. 111–126, 1989.

[31] F. Ipate and M. Holcombe, "Complete testing from a stream x-machine specification," *Fundam. Inf.*, vol. 64, no. 1-4, pp. 205–216, 2004.

[32] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Towards a tool environment for model-based testing with AsmL." in *FATES*, 2003, pp. 252–266.

[33] E. Farchi, A. Hartman, and S. S. Pinter, "Using a model-based test generator to test for standard conformance." *IBM Systems Journal*, vol. 41, no. 1, pp. 89–110, 2002.

[34] H. S. Hong, Y. R. Kwon, and S. D. Cha, "Testing of object-oriented programs based on finite state machines." in *APSEC*. IEEE Computer Society, 1995, pp. 234–.

[35] L. Apfelbaum, "Automated functional test generation." in *Autotestcon '95 Conference*. IEEE, 1995.

[36] L. Lucio, L. Pedro, and D. Buchs, "A methodology and a framework for model-based testing." in *RISE*, ser. Lecture Notes in Computer Science, N. Guelfi, Ed., vol. 3475. Springer, 2004, pp. 57–70.

[37] J. A. Whittaker, "Stochastic software testing." *Ann. Software Eng.*, vol. 4, pp. 115–131, 1997.

[38] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-based testing of object-oriented reactive systems with spec explorer." May 2005.

[39] P. W. M. Koopman, R. Plasmeijer, and P. Achten, "Model-based testing of thin-client web applications." in *FATES/RV*, 2006, pp. 115–132.

[40] N. H. Lee and S. D. Cha, "Generating test sequences from a set of mscs." *Computer Networks*, vol. 42, no. 3, pp. 405–417, 2003.

[41] F. Belli, "Finite-state testing and analysis of graphical user interfaces," in *ISSRE*. IEEE Computer Society, 2001, pp. 34–43.

[42] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[43] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback for test automation," University of Maryland, College Park, MD USA, Technical Report, Aug. 2009, http://hdl.handle.net/1903/9416. [Online]. Available: http://hdl.handle.net/1903/9416

[44] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 4, 2007.

[45] L. C. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in *ICSE '04: Pro-*

*ceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 86–95.

[46] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 3, pp. 277–331, 2004.

[47] D. J. Richardson and M. C. Thompson, "An analysis of test data selection criteria using the relay model of fault detection," *IEEE Trans. Softw. Eng.*, vol. 19, no. 6, pp. 533–553, 1993.

[48] F. Belli and C. J. Budnik, "Test minimization for human-computer interaction," *Applied Intelligence*, vol. 26, no. 2, pp. 161–174, 2007.

[49] P. T. Alessandro Marchetto and F. Ricca, "State-based testing of Ajax web applications," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Valication*, April 9–11, 2008, pp. 121–130.

**Xun Yuan** is a Software Engineer in Test (SET) at Google Kirkland where she is in charge of ensuring the quality of a web-based software product called Website Optimizer. She completed her PhD from the Department of Computer Science at the University of Maryland in 2008 and MS in Computer Science from the Institute of Software Chinese Academy of Sciences in 2001. Her research interests include software testing, quality assurance, web application design, and model-based design. In addition to her interests in Computer Science, she also likes mathematics and literature.

**Atif M Memon** is an Associate Professor at the Department of Computer Science, University of Maryland. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is the inventor of the GUITAR system (http://guitar.sourceforge.net/) for automated model-based GUI testing. He is the founder of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS). He serves on various editorial boards, including that of the Journal of Software Testing, Verification, and Reliability. He has served on numerous National Science Foundation panels and program committees, including the International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Software Testing Verification and Validation (ICST), Web Engineering Track of The International World Wide Web Conference (WWW), the Working Conference on Reverse Engineering (WCRE), International Conference on Automated Software Engineering (ASE), and the International Conference on Software Maintenance (ICSM). He is currently serving on a National Academy of Sciences panel as an expert in the area of Computer Science and Information Technology, for the Pakistan-U.S. Science and Technology Cooperative Program, sponsored by United States Agency for International Development (USAID). In addition to his research and academic interests, he handcrafts fine wood furniture.