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

Xun Yuan and Atif M Memon
Department of Computer Science, University of Maryland,
College Park, MD 20742, USA
{xyuan, atif}@cs.umd.edu

Abstract

Graphical user interfaces (GUIs) are the sole mode of interaction between end-users and back-end code for almost all of today's software applications. Because of this strategic role of GUIs, their quality has become important. During GUI testing, test cases—modeled as sequences of user events—sample the vast input space of all possible sequences with the goal of detecting faults; for effective testing, it is important to sample this space carefully. Existing techniques either sample manually or employ manually constructed abstract models—the abstraction and/or subsequent test-case generation algorithms enable sampling.

This paper presents an alternative approach to GUI testing – it's focus is on developing a *fully automatic* model-driven technique to generate GUI test cases. The technique is novel in that it uses feedback from the execution of a *"seed test suite"* on a GUI. The seed suite is generated automatically using an existing structural *event-interaction graph* (EIG) model of the GUI. During its execution, the run-time effect of each GUI event on all other events pinpoints new important *event-semantic interaction* (ESI) relationships between them, which are used to automatically create an *ESI graph* (ESIG) model and generate new test cases. Together with a reverse-engineering algorithm used to obtain the EIG, seed suite, ESIG, and new test cases, the feedback-based technique yields a fully automatic, end-to-end GUI testing process. Two independent 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.

1 Introduction

Automated test case generation (ATCG) has become increasingly popular due to its potential to reduce testing cost and improve software quality [22]. A typical approach used for ATCG is to create an abstract model (*e.g.*, state-machine model [8, 67, 73], event-flow model [44]) 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 the tasks of model creation and maintenance 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 [14,21,25–27,37,52,54,83,84,86]. 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 *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 [25–27, 37, 52, 54]. 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 [14], operational abstractions to cover increased program behaviors [21, 84], and partially generated non-exception-throwing method-call sequences to generate longer sequences [57].

This paper presents a new feedback-based technique for automated testing of graphical user interfaces (GUIs). The technique starts with an automatically reversed engineered *structural* model of the GUI, employs the model to automatically generate specific types of test cases (sequences of GUI events that exercise GUI widgets), executes them, and uses the execution results as feedback to automatically generate additional test cases. The feedback is an abstraction of the run-time state of GUI widgets. As noted by Mathur [41], there is a strong relationship between events in a

software and its states; GUIs are no exception. Because GUI events may drive the software into potentially new states, this feedback-based approach leverages run-time GUI state information to prune the state-space, thereby helping to reduce the number of test cases that need to be executed.

The nature of GUIs, their test cases, and the maturity of our existing model-based GUI test-case generation algorithms lend themselves to feedback-based techniques for a number of reasons. First, 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 [56]. A correct GUI is necessary for trouble-free execution of the application's underlying "business logic" [10, 67, 73]. Second, our existing fully automatic model-based GUI test-case generation algorithms produce test cases that exhaustively test *two-way interactions* between GUI events; these test cases are called smoke tests [45]. They are used as the basis for feedback collection, *i.e.*, they form the seed suite. Finally, our existing tools are easily adapted to monitor and store the run-time state of the GUI.

Our previous empirical studies showed that smoke test cases reveal a large number of GUI faults; we and other researchers have shown that additional faults may be detected by testing certain types of multi-way interactions [10, 49]. The challenge, of course, is how to systematically generate test cases for these interactions. Exhaustively testing them is impossible because the number of GUI test cases grows exponentially with number of events in the test case. A practical alternative is to identify small subsets of events that interact in interesting ways with one another and hence should be tested together, and generate test cases that test multi-way interactions among members of each subset. In this paper, we use the feedback-based technique to *automatically* identify such sets of events.

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. The EIG is a structural model of the GUI. More specifically, it represents *all possible sequences* of events that may be executed on the GUI. Note that an EIG has the flavor of the conventional control-flow model that represents all possible execution paths in a program [3], and a data-flow model that

represents all possible definitions and uses of a memory location [62]; except that an EIG is at the GUI event level of abstraction.

The EIG-generated 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 then 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.

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 [86], 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 our 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

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

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 [1, 11, 17–20, 23, 68, 74], grammars [5, 31, 42, 71, 72], AI planning [30, 39, 47, 66], genetic algorithms [35], probabilistic models [75, 77–79], architecture diagrams [65], and specifications [32, 33] to generate such sequences. All the above 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. The 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. In particular, we leverage our existing graph-traversal techniques based on an automatically obtained reverse engineered GUI model [46, 50] to develop a fully automatic model-based testing technique. Run-time feedback is used to enhance the model and generate new test cases. We feel that this type of 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 our previous work on automated, model-based, systematic GUI test-case generation,

- definition of new relationships among GUI events based on the GUI widgets that they use/influence,
- utilization of run-time state to explore a larger input space and improve fault-detection effectiveness,
- 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 the type of test suites, and
- demonstration that certain faults require well-crafted combinations of test cases and test 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. Models may be derived from a formal specification of the software or reverse engineered by observing the software's execution behavior. They may be described using various languages and mathematical objects.

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) [4, 6, 24, 28], *UML Diagram-based Models* [40] and *Markov Chains* [34, 76].

There have been numerous reports of success stories using FSM models for test automation. For example, Microsoft researchers [6] modeled the control flow of an object-oriented software under test as an FSM and described it using the *Abstract State Machine Language* (AsmL `research.microsoft.com/fse/asml`). A traversal engine (part of Spec Explorer, a tool for advanced model-based specification and conformance testing), used the resulting finite state machine to produce behavioral tests to cover all explored transitions. Hong *et al.* also used FSMs for unit testing of classes in object-oriented programs; they used FSMs to model interactions between class data members and member functions [28]. The FSMs, called *class state machines* (CSM), were then transformed into *class flow graphs* (CFG); test case generation was done by selecting test cases according to the locations of definitions and uses of variables in the CFG. In another reported research, Farchi *et al.* used FSM models to test implementations of the $POSIX$ standard and $Java$ exception handling [24]. Both state machine models were created from the software specifications and represented using the *GOTCHA Definition Language*. The GOTCHA-TCBean test generator was then used to automatically explore the state space from the model and generate an abstract test suite.

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 [4]; 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 [16,36,38,53]. 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 state machine models to test object-oriented reactive systems [16]. 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.

Table-based Models: Table-based models define software behavior in the form of tables relating model elements such as system modes, conditions, events and terms. These tables are then used as the basis for test case generation. The table-based modeling approach SCR, which is an abbreviation for *software cost reduction*, has been used for security functional testing [12] and *Mars Polar Lander* software to detect faults [13].

Grammars: Production grammars have been used to test large, complex and safety-critical software systems; a popular example is the Java Virtual Machine [69]. The grammars are collections of non-terminal to terminal mappings that resemble regular parsing grammars. A production grammar produces a program (*i.e.*, a set of terminals, or tokens) starting from a high-level description (*i.e.*, a set of non-terminals). The composition of the generated programs models the restrictions placed on the software by the production grammar.

Summary: The above model-based testing techniques rely heavily on the manual or semi-manual construction of the abstract model. Consequently, they are prone to errors. Moreover, any change to the software requires reconstruction of the model, which is typically cumbersome and time consuming.

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 and algorithms to generate test cases from the model.

State-Based Techniques: Finite state machines have been used to model GUIs [7, 73]. A GUI's state is represented in terms of its windows and widgets; each user event triggers a transition in the FSM. For the sake of test-case generation, a test case is a sequence of user events and corresponds to a path in the FSM. As is the case for conventional software (discussed in the previous section), 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 [7] converted a GUI FSM into simplified regular expressions. The regular expressions were used to generate event sequences. Shehady *et al.* [67] 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. Event transition may modify values of these variables.

AI planning has also been used to manage the state-space explosion by eliminating the need for explicit states. AI planning models the infinite state space of a GUI [48]. 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 state to the goal state. However, the quality of the test cases is determined by the choice of tasks. Moreover, the manual operator definition and task selection may be expensive for large GUIs.

Genetic Algorithm: Test cases have been generated using genetic algorithms to mimic novice users [35]. The approach uses an expert to generate an initial event sequence manually and then uses genetic algorithm 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. Although useful for generating multiple test cases, the technique relies on an expert to generate the initial test case. The final test suite depends largely on the paths taken by the expert user. The idea of using a task and generating an initial test case may be better handled by using planning, since multiple test cases may be generated automatically according to some predetermined coverage criterion.

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) [45] and *Event Interaction Graphs* (EIG). Because of their central role in this paper, we discuss these models in Section 3.

Summary: FSM models and genetic algorithm based GUI testing suffer from the problem of manual creation of the model, *i.e.*, state machine and fitness function. The primary problem with the GUI graph models is that the number of event sequences grows exponentially with length. Hence, the existing graph-model based GUI test-case generation algorithms have only been able to generate test cases that cover all edges in the graph models, *i.e.*, they test *two-way* interactions between GUI events.

2.3 Execution Feedback for Test Case Generation

In an earlier report of this research [86], we introduced the idea of employing *feedback* from the execution of a seed test suite (our smoke tests generated using the EIG) to generate additional multi-way interaction test cases. A study on four large fielded open-source software applications demonstrated the feasibility and usefulness of this new approach; we showed that feedback was able to significantly improve our existing techniques and help identify/report serious problems in the software applications.

We now discuss similar feedback-based approaches used by other researchers on non-GUI software to generate test cases. Execution feedback refers to information obtained during test execution, and used to guide automatic test case/test suite generation. This is called *dynamic test case generation* and, to the best of our knowledge, was originally proposed by Miller and

Spooner [54]. 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, which may be a test suite or a single test case. The execution feedback is collected and analyzed. The results are used to evaluate the "closeness," according to some criterion, of the previous execution to the desired outcome; the model used to generate test cases is then modified accordingly and a new test case is generated. This loop stops when the "closeness" evaluation is satisfied.

Since then, several researchers have used similar ideas of dynamic test generation.

Object Properties: The work of Xie *et al.* is most closely related to this research [83, 84]. They 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 for test case generation. 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. The test case generation process starts from an existing test suite. Through executions of these test cases, object states (values of variables and parameters, and return values) are recorded at the entry and exit of method executions. Based on the collected traces and a set of pre-defined axiom-pattern templates, equality patterns are searched to create operational abstractions. The discovered operational abstractions consist of object properties that hold for all the observed executions. By removing or relaxing inferred preconditions on parameter values in the operational abstractions, both legal and illegal test cases are generated. The newly generated test cases are executed. Because they were generated by relaxing inferred preconditions, some of these test cases may cause an uncaught runtime exception. The other, non-crashing test cases are used to obtain new operational abstractions, which are again used to generate additional test cases. Other researchers have also used operational abstractions, combined with symbolic execution, to guide the generation of test cases [21].

Method-call Sequences: Pacheco *et al.* [58] have improved random unit test generation by

incorporating feedback obtained from executing test inputs as they are created. They build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. The key idea of their work is that they build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.

Similarly, Boyapati *et al.* employ a feedback-based technique to obtain all non-isomorphic inputs (test cases) for a method [14]. A programmer develops (1) a "guided test generation engine" that outputs test cases to explore the method's input space and (2) a predicate from the method's preconditions to check the validity of the generated input. This technique prunes a large portion of the input space by monitoring the execution of the predicate on an initial test suite, guiding the engine and yielding a suite of all non-isomorphic inputs.

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.* [54] 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 [26, 27, 37]. 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. One disadvantage of these approaches is that they can get stuck in a local minima during test case generation.

The *chaining* approach [25] 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 [52], 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.

Summary: All the above execution feedback-based techniques have been used for a specific type of test case, that is, numerical data values. The feedback (in the form of branch predicate evaluations, condition-decision coverage, and object states) is used to tweak these numerical values in order to improve overall coverage. These techniques are not directly applicable to GUI testing because a GUI test case is a sequence of events. There is no clear notion of tweaking a GUI test case to improve coverage.

Although the techniques discussed in this section are not directly applicable to feedback-directed GUI test case generation, many of the underlying concepts have been used. 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.

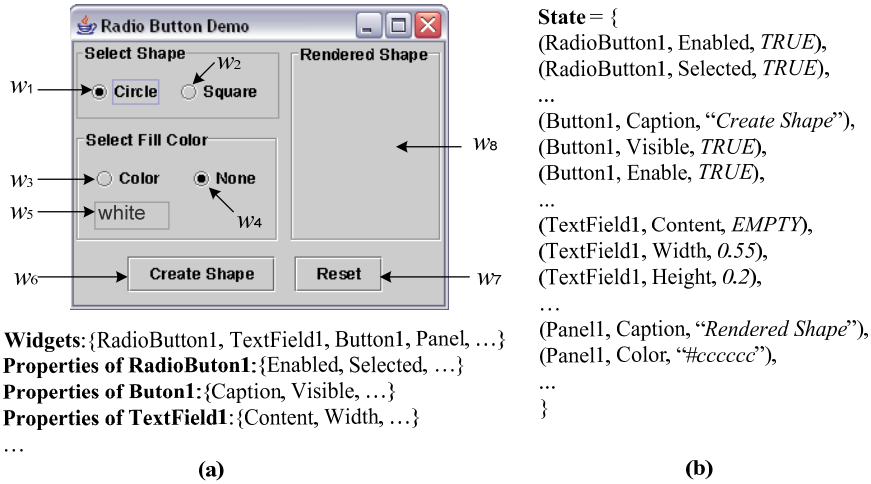


Figure 1: (a) "`Radio Button Demo`" GUI, (b) its Partial Observable State

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. Figure 1 shows the partial GUI state of a simple application's window called `Radio Button Demo`. The GUI contains eight widgets labeled w_1 through w_8; a user can perform events e_1 through e_7 on w_1 through w_7, respectively; no event can be performed on w_8.

A set of states S_I is called the *valid initial state set* for a particular GUI if the GUI may be in

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

any state $S_i \in S_I$ when it is first invoked. The single *initial state* for `Radio Button Demo` has `Circle` and `None` selected; the text-field corresponding to w_5 has a default color "white"; and the `Rendered Shape` area (widget w_8) is empty.

The state of a GUI is not static; events e_1, e_2, ..., e_n performed on the GUI change its state and hence are modeled as functions that transform one state of the GUI to another. For `Radio Button Demo`, event e_1 sets the shape to a circle; if there is already a square in the `Rendered Shape` area, then it is immediately changed to a circle. Event e_2 is similar to e_1, except that it changes the shape to a square. Event e_3 enables the text-field w_5, allowing the user to enter a custom fill-color, which is immediately reflected in the shape being displayed (if there is a shape there). Event e_4 reverts back to the "no fill color" state. Event e_5 is used to fill a custom color in the text-field w_5. Event e_6 creates a shape in the `Rendered Shape` area according to current settings of $w_1 \ldots w_5$; event e_7 resets the entire software to its initial state.

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.

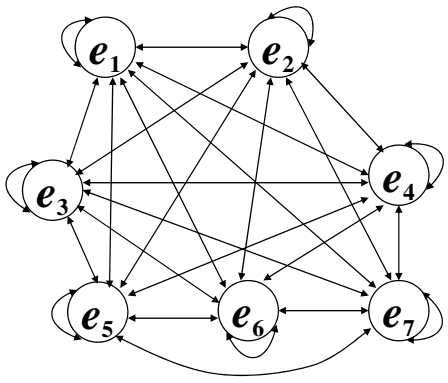


Figure 2: EIG of "`Radio Button Demo`" GUI

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

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 [49]. 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 [49]). 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 EIG for the `Radio Button Demo` is shown in Figure 2. Because this is a single-window GUI with no menus, it has no structural events; it contains only system-interaction events. There is one node for each of the widgets on which a user can perform an event.

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 [49], 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. For the EIG of Figure 2, there are a total of 49 test cases of length 2, corresponding to the 49 edges in the EIG. This research will generate test cases for *multi-way interactions*, *i.e.*, longer paths in an EIG. For example, a 3-way test case is $< e_1; e_2; e_3 >$; a 4-way test case is $< e_1; e_2; e_7; e_2 >$. 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 [49]. To allow a clean application exit, a 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 our ability to identify sets of events that need to be tested together in multi-way interactions. Because each event is executed using its corresponding event handler, one could hypothesize that an event e_x whose event handler updates code element(s) (*e.g.*, variables, object fields) that are used (directly or indirectly) by another event e_y's handler could potentially influence it's execution; e_x and e_y are good candidates to be tested together. For example, consider the event handlers for the events e_1 and e_2 shown in Figure 3. As these event handlers interact via the variable *currentTool*, the events e_1 and e_2 should be tested together. Similarly, events e_3 and e_4 interact via *curZoom* and should be tested together. However, because the handlers for e_1 and e_3 do not interact, these events need not be tested together.

e_1:: select ellipse tool

```
public void ellipsePerformed (java.awt.event.ActionEvent evt){
...;      currentTool = toolEllipse; ... }
```

e_2:: drag mouse on canvas

```
public void mouseDragged(java.awt.event.MouseEvent evt) {
...;      currentTool.dragAction(newEvt, center); ... }
```

e_3:: set zoom factor to double

```
public void zoom1Performed(java.awt.event.ActionEvent evt) {
...;       curZoom = zoom1; ... }
```

e_4:: click left mouse button on canvas

```
public void mousePressed(java.awt.event.MouseEvent evt) { ...
if (currentTool == toolZoom){ // if the zoom tool is being used
  int temp = toolZoom.getZoom(); // current zoom level
  if (SwingUtilities.isLeftMouseButton(evt){ switch (temp){
    case 1: zoom2.setBG(pColor); curZoom = zoom2;
    case 2: zoom3.setBG(pColor); curZoom = zoom3;... } }}
theCanvas.repaint();... }
```

Figure 3: Example Event Handlers

One may employ a variety of static program-analysis techniques to identify such interactions [64]. They can certainly be used in this example. However, the limitations of static analysis in the presence of multi-language GUI implementations, callbacks for event handlers, virtual function calls, reflection, and multi-threading are well known [43, 64]; these limitations complicate the application of static analysis. Moreover, because most GUI applications employ a large number of library elements (***e.g.***, Java Swing), source code (an essential component for most static analysis techniques) may not be available for parts of the GUI.

This research avoids static analysis; instead it approximates the identification of interactions between event handlers by analyzing feedback from the run-time state of the GUI on an initial test suite. Recall that testing all two-way interactions between events is already quite practical with the smoke test suite; we treat this suite as a starting point to collect the feedback. The remaining question, addressed in this section, is: *What dynamic behavior constitutes event interaction?*

Consider the example shown in Figure 4. The top-left shows the *initial state* (S_0) of an application. After an event e_1 ($Select$ `Ellipse tool`; event handler shown in Figure 3) is executed, the GUI changes its state to the one shown in the top-right ($e_1(S_0)$). In this state, the "ellipse tool" remains selected. Starting from S_0, one can execute another event e_2 (*Drag mouse on canvas*) and obtain the state shown in the bottom-left ($e_2(S_0)$); an area of the canvas has been selected. If, however, the sequence $< e_1; e_2 >$ is executed in S_0, a new state ($e_2(e_1(S_0))$), shown in the bottom-right is obtained; an ellipse has been created. This execution is equivalent to the execution of event e_2 in the state $e_1(S_0)$. The sequence $< e_1; e_2 >$ produces a GUI state that shows the influence of e_1 on e_2. Hence, e_1 and e_2 are good candidates to be tested together in longer sequences to check for interaction problems. The code for e_1 and e_2 (previously seen in Figure 3) shows that they do in fact interact.

The remaining problem is to automatically compute the run-time relationship between event e_1 and e_2 of Figure 4. We use four state descriptions for this computation: S_0, $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$. More specifically, to compute this relationship, we need to find at least one *new* widget w with property p and value v in state $e_2(e_1(S_0))$, ***i.e.***, it is created by event sequence $< e_1; e_2 >$;

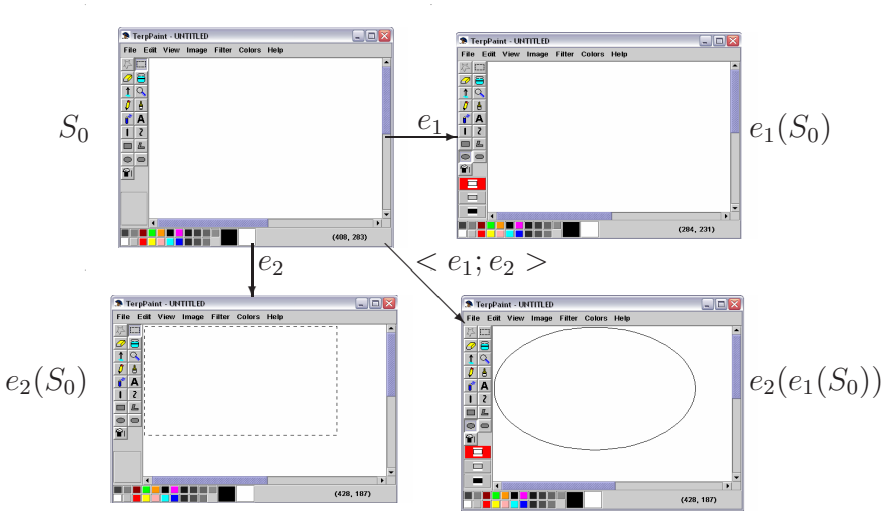


Figure 4: Execution of Events e_1 ($Select$ `Ellipse tool`) and e_2 (*Drag mouse on canvas*)

but it does not exist in state S_0 and could not be created by either e_1 or e_2 individually, *i.e.*, no triple involving widget w exists in any of the states S_0, $e_1(S_0)$ and $e_2(S_0)$. This statement can be formally described as a predicate: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t.$[4] $(((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, \bar{p}, \bar{v}) \notin e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v) \in e_2(e_1(S_0))))$. If this predicate evaluates to TRUE for a widget (in our case the ellipse), then we have discovered a case where e_1 influences e_2.

It is quite straightforward to encode such a predicate in a high-level programming language, such as Java. The implementation would loop through the state triples and stop when one widget satisfying the predicate is detected.

The reader should note that the above predicate is necessary for the computation of the relationship between e_1 and e_2. At first glance, the reader might be tempted to think that checking state non-equivalence would be sufficient to identify interacting events, *i.e.*, by using a predicate $\mathcal{P}$ such as $(e_1(S_0) \neq e_2(e_1(S_0))) \vee (e_2(S_0) \neq e_2(e_1(S_0)))$. However, this is not the case. Consider an example of two non-interacting events, e_x and e_y, which toggle the states of two independent check-box widgets $\square_x$ and $\square_y$, respectively. Starting in a state $S_0 = \{\square_x, \square_y\}$, *i.e.*, both boxes unchecked, each event would "check" its corresponding check-box, *i.e.*, $e_x(S_0) = \{\boxtimes_x, \square_y\}$, $e_y(S_0) = \{\square_x, \boxtimes_y\}$, and $e_y(e_x(S_0)) = \{\boxtimes_x, \boxtimes_y\}$. Even though $\mathcal{P}$ would evaluate to TRUE for this

[4]Notation for "such that"

example, events e_x and e_y are non-interacting and need not be tested together.

Additional complexity arises from the nature of modal windows in GUIs. Modal windows create special situations due to the presence of termination events. For example, instead of being in the main window, had e_1 and e_2 in Figure 4 been associated with widgets contained in a modal window with termination event `TERM`, states $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$ might not contain the necessary information needed to compute the predicate. This is because user actions in modal windows do not cause immediate state changes; they typically take effect after a termination event 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.

It turns out that the example illustrated in Figure 4 is just one *case* of how the GUI state may be used to pinpoint interactions between event handlers – there are several more cases, each requiring a different predicate. Because of the need to define precise predicates for all these cases and for special handling of modal windows, we take a two dimensional approach. We first define six[5] 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.

Case 1: $\mathcal{P}_{1(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.\ ((v \neq v') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; 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'.

Figure 5 gives an example of **Case 1**. This is a "`GUI Demo`" application with several widgets. The `Fill with color` checkbox fills the currently selected shape (highlighted with a deep

[5]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.

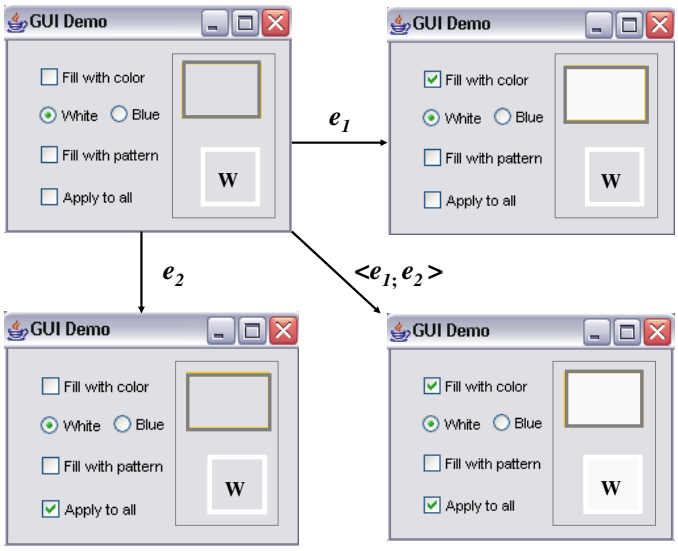


Figure 5: **Case 1**: e_1: *Check* `Fill with color`; e_2: *Check* `Apply to all`

grey border) with the chosen color determined by the radio buttons `White` and `Blue`. Checkbox `Fill with pattern` determines whether to fill the selected shape with a pattern. Checking `Apply to all` sets all shapes in the right panel with the same color and pattern.

For the purpose of **Case 1**, e_1 is *Check* `Fill with color` and e_2 is *Check* `Apply to all`. The initial state has the rectangle widget selected and color is set to white. The square widget (marked with **W**) is not modified by e_1 or e_2 individually; however, the event sequence $< e_1; e_2 >$ fills the square with the white color. Hence $\mathcal{P}_{1(1)}(e_1, e_2)$ evaluates to `TRUE` – **Case 1** is applicable here and e_1 is ESI related to e_2 because e_1 influences e_2 and their combination modifies the previously unmodified widget **W**.

Case 2: $\mathcal{P}_{2(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, s.t.\ ((v \neq v') \wedge (v' \neq v'') \wedge ((w, p, v) \in \{S_0 \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$ 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 $< e_1; e_2 >$.

An example of **Case 2** using the "`GUI Demo`" application is given in Figure 6, where e_1 now represents *Check* `Fill with color` and e_2 is *Click radio button* `Blue`. The initial state has the rectangle selected and color is set to white. Individually, in this initial state, event e_1 fills

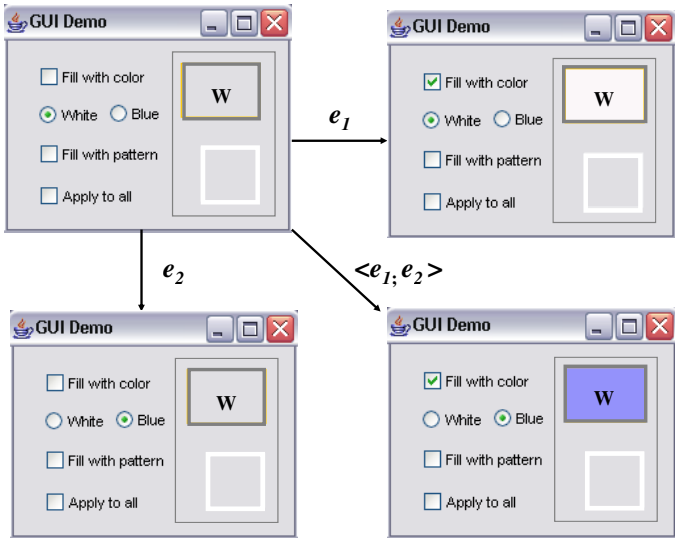


Figure 6: **Case 2**: e_1: *Check* `Fill with color`; e_2: *Click radio button* `Blue`

the rectangle with the white color; event e_2 sets the current color to blue. However, executing $< e_1; e_2 >$ now fills the rectangle with the color blue. Hence $\mathcal{P}_{2(1)}(e_1, e_2)$ evaluates to TRUE – **Case 2** applies here as e_1 influences e_2 execution; the widget (marked with **W**) is modified by e_1; it is not modified by e_2; however, it is modified differently by the sequence $< e_1; e_2 >$.

Case 3: $\mathcal{P}_{3(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, s.t.\ ((v \neq v') \wedge (v' \neq v'') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0)\}) \wedge ((w, p, v') \in e_2(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$ 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.

An example of **Case 3** using the "`GUI Demo`" application is given in Figure 7, where e_1 now represents *Click radio button* `Blue` and e_2 is *Check* `Fill with color`. The initial state has the rectangle selected and color is set to white. Individually, in this initial state, event e_1 sets the current color to blue; event e_2 fills the rectangle with the white color. However, executing $< e_1; e_2 >$ now fills the rectangle with the color blue. Hence $\mathcal{P}_{3(1)}(e_1, e_2)$ evaluates to TRUE – **Case 3** applies here as e_1 influences e_2 execution; the widget (marked with **W**) is not modified by

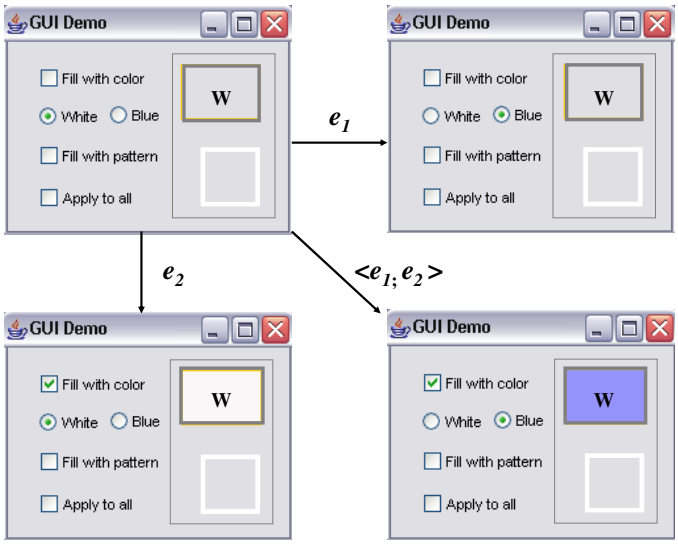


Figure 7: **Case 3**: e_1: *Click radio button* `Blue`; e_2: *Check* `Fill with color`

e_1; it is modified by e_2; however, it is modified differently by the sequence $< e_1; e_2 >$.

Case 4: $\mathcal{P}_{4(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \bar{v} \in V_p, s.t.\ ((v \neq v') \wedge (v \neq v'') \wedge (v'' \neq \bar{v}) \wedge ((w, p, v) \in S_0) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(S_0)) \wedge ((w, p, \bar{v}) \in e_2(e_1(S_0))))$; 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 $< e_1; e_2 >$.

Figure 8 shows one example of this case using the "`GUI Demo`" application. In this example, the initial state has `Fill with color` checked, white is set to be the current color and the rectangle is selected. Event e_1 here is *Click radio button* `Blue` and e_2 is *Check* `Fill with pattern` that fills the current shape with a pattern. Events e_1 and e_2 modify the rectangle individually; however, executing $< e_1; e_2 >$ now modifies the rectangle differently. Therefore, e_1 influences e_2, *i.e.*, resulting in different modification of the existing widget (marked with **W**), and **Case 4** applies because $\mathcal{P}_{4(1)}(e_1, e_2)$ evaluates to `TRUE`.

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: $\mathcal{P}_{5(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.\ ((v \neq v') \wedge ((w, p, v) \in e_x(S_0)) \wedge ((w, p, v) \notin S_0) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; 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.

Figure 9 shows an example for **Case 5** using the "`Radio Button Demo`" application introduced earlier. The initial state has the `Circle` and `None` radio button selected and an empty `Rendered Shape` panel. In this initial state, event e_2 selectes `Square`; event e_6 clicks the button `Create Shape` and creates a circle in the `Rendered Shape` panel. However, when executing $< e_2; e_6 >$ in the initial state, a square is created in the `Rendered Shape` panel. Therefore, e_2 influences e_6; $\mathcal{P}_{5(1)}(e_2, e_6)$ evaluates to `TRUE` – **Case 5** is applicable here.

It turns out that in this simple application, **Case 5** also applies to $< e_6; e_2 >$, $< e_3; e_6 >$ and $< e_6; e_3 >$.

A common occurrence of event interaction in GUIs is enabling/disabling widgets, which may be modeled as the widget's `ENABLED` property being set to `TRUE` or `FALSE`.

Case 6: $\mathcal{P}_{6(1)}(e_1, e_2) = \exists w \in W, \texttt{ENABLED} \in P_w, \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}}, s.t.\ (((w, \texttt{ENABLED}, \texttt{FALSE}) S_0) \wedge ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_1(S_0)) \wedge \texttt{EXEC}(e_2, w))$; 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, represented by a predicate $\texttt{EXEC}(e_2, w)$.

Using the `Radio Button Demo` application, we see in Figure 10 that e_3 enables e_5. Hence $\mathcal{P}_{6(1)}(e_3, e_5)$ evaluates to `TRUE` – **Case 6** applies.

As mentioned earlier, the second dimension of our definitions are contexts. This is because modal windows create special situations for Cases 1 through 6 due to the presence of termination events. User actions in these windows do not cause immediate state changes; they typically take effect after a termination event has been executed, leading to *contexts 2 and 3*.

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

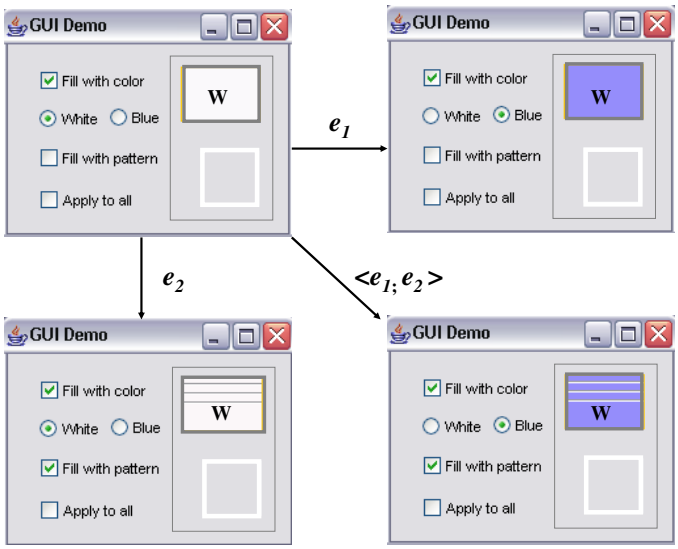


Figure 8: **Case 4**: e_1: *Click radio button* `Blue`; e_2: *Check* `Fill with pattern`

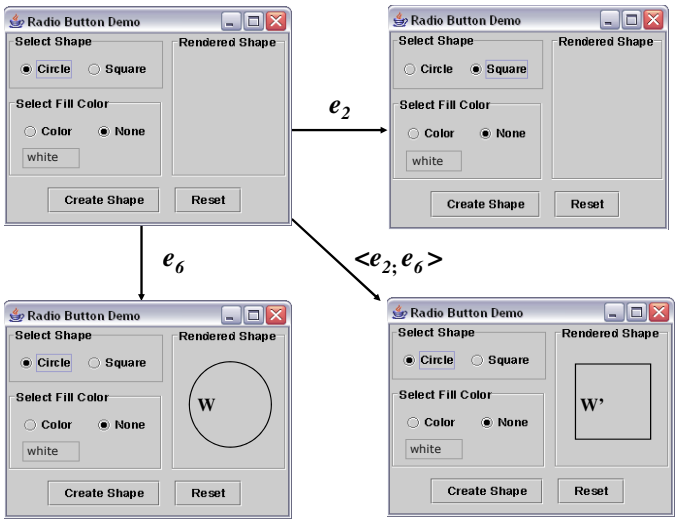


Figure 9: **Case 5**: e_2: *Select* `Square`; e_6: *Click button* `Create Shape`

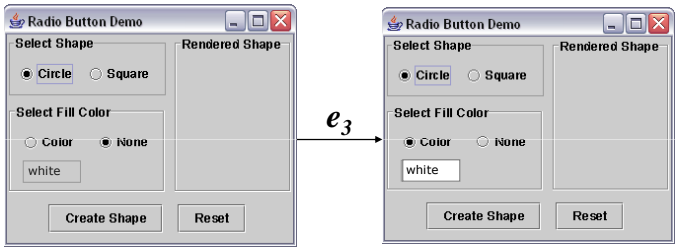


Figure 10: **Case 6**: e_3: *Select* `Color`; e_5: *Input color*

$e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $< e_1; e_2; \texttt{TERM} >$. All the predicates defined in Cases 1 through 6 apply, using these modified definitions, for e_1 and e_2 in the same modal window. The notation used for these 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 $< e_1; \texttt{TERM}; e_2 >$. All the predicates defined in Cases 1 through 6 apply. The notation used for these predicates when applied in Context 3 is $\mathcal{P}_{n(3)}(e_1, e_2)$, where n is the case number.

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. As demonstrated earlier, for our `Radio Button Demo` application, $e_2 \xrightarrow{5(1)} e_6$, $e_6 \xrightarrow{5(1)} e_2$, $e_3 \xrightarrow{5(1)} e_6$, $e_6 \xrightarrow{5(1)} e_3$, and $e_3 \xrightarrow{6(1)} e_5$.

Once all of the cases have been implemented, the feedback-based process execution is straight-

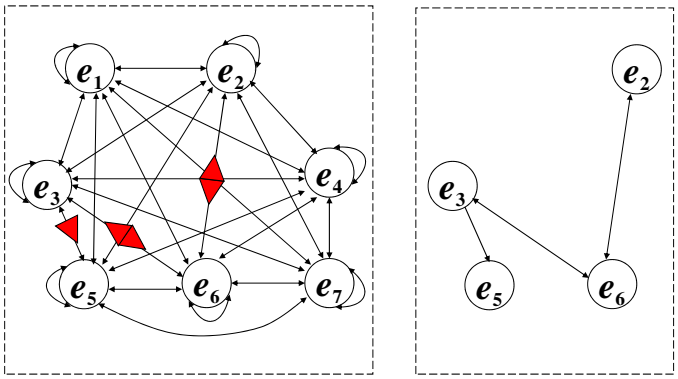


Figure 11: EIG with marked ESI relationships and ESIG for "`Radio Button Demo`" GUI

forward. 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. The EIG annotated with the five ESI relationships found in `Radio Button Demo` are shown in Figure 11; the ESIG (shown on the right) is a subgraph of the EIG.

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 instead concatenate pairs of paths together and union all of them (respectively) to eliminate duplicates. The final matrix entries are paths of specific lengths, *i.e.*, test cases of specific lengths.

Much of the functionality needed for this test-case generation approach is implemented in the *Mathematica* package. The function `MatrixPower[mat, n]`, returns the `n`th matrix power of matrix `mat`. If the matrix is encoded as a 0/1 adjacency list, `MatrixPower` returns a matrix in which each entry (i, j) is a count of the number of test cases of a specific length from node i to node j. We used simple rewriting rules (provided as a `Replace` function denoted by the operator "`/.`") built into *Mathematica* to alter `MatrixPower` – we replaced the `Times` (*i.e.*, multiply) operator with `Join` (*i.e.*, list concatenation) and the `Plus` operator with `Union`. We thus generate all length `N` -1 test cases by using the *Mathematica* command:

```
MatrixPower[matrix, N]/.{Power[x_, 2]->Join[x, x], Plus->Union,
Times->Join}
```

with an appropriate representation of the matrix.

Because our ESIG contains very few edges and nodes, the adjacency matrix tends to be sparse, leading to fast computations. For increased efficiency, *Mathematica*'s `MatrixPower` uses the built-in `SparseArray` structure.

For our example ESIG of Figure 11, the test cases of length 3 are $< e_2; e_6; e_2 >$, $< e_2; e_6; e_3 >$, $< e_3; e_6; e_3 >$, $< e_3; e_6; e_2 >$, $< e_6; e_3; e_6 >$, and $< e_6; e_2; e_6 >$. Note that because there is no non-determinism in the test-case generation algorithm, there is a unique ESIG-based test suite of specific-length test cases for an application.

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.

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: How many test cases are required to test two-way interactions in an EIG? How does this number grow for 3-, 4-, ..., n-way interactions?

S1Q2: 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?

S1Q3: 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?

More specifically, the following process was used for this study:

1. Select software subjects with GUI front-ends.
2. Generate a seed test suite using EIGs.
3. Execute the seed test suite. Report crashes.
4. During execution, construct the ESIG. Use the ESIG to generate additional test cases.
5. Execute the new test cases. Report crashes.

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 [80, 86]; 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* [29] was executed on all the applications; after the study, we

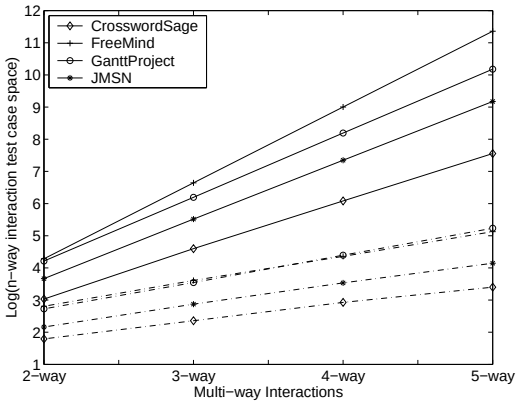


Figure 12: Test Case Space Growth

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. To address **S1Q1** above, the number of test cases required to test 2-, 3-, 4-, and 5-way interactions was computed. The result for each application is shown as a solid line in Figure 12 (the y-axis in all these plots is a logarithmic scale). The plot shows that the number of test cases grows exponentially with the number of interactions. The number quickly becomes unmanageable for more than 2- and 3-way interactions. 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.

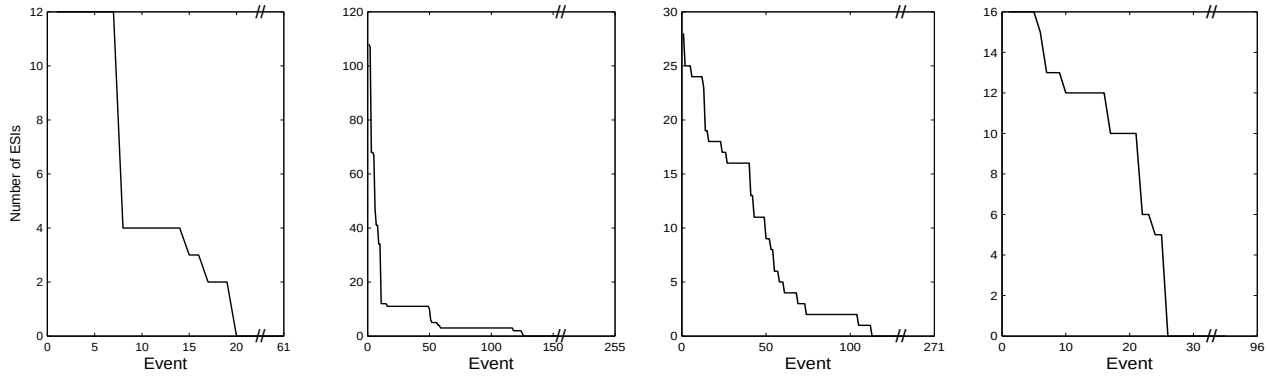
Note that debugging and fault-fixing was necessary due to two reasons. First, had we not done so, the longer test cases that we will generate in the next few steps may contain these short test cases as subsequences; the longer tests may hence also crash due to the faults previously detected by the seed 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 **S1Q2**, the number of ESI relationships in which each event participates is shown in Figure 13. 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 required to test 2-, 3-, 4-, and 5-way interactions using an ESIG is shown, for each application, as a dotted line in Figure 12. This result shows that the growth of the ESIG-generated test cases appears manageable for 3-, 4-, and (given sufficient resources) 5-way interactions. They are in fact reduced from the EIG by 99.78%, 99.97%, and 99.99% for 3-, 4-, and 5-way interactions, respectively. In this study, test cases for 3-, 4-, and 5-way interactions were generated. The total number of test cases for these interactions was 3592, 160,629, 199,127, and 18,144 for CrosswordSage, FreeMind, GanttProject, and JMSN, respectively.



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

Figure 13: ESI Distribution in OSS

STEP 5: Execution of the test cases. To address **S1Q3**, 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 additional faults.

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.

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

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

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

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

Table 1: TerpOffice Applications

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 the development of a formal specification and 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 [51, 82]. 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.

	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

Table 2: ESIG vs. EIG Sizes

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 quite impractical to generate 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 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

	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

Table 3: ESIG vs. EIG Fault Detection

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 [81]. 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 [86]. 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 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 – an impractical task. 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 [15]. 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. Their execution 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.* [63] 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. 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,

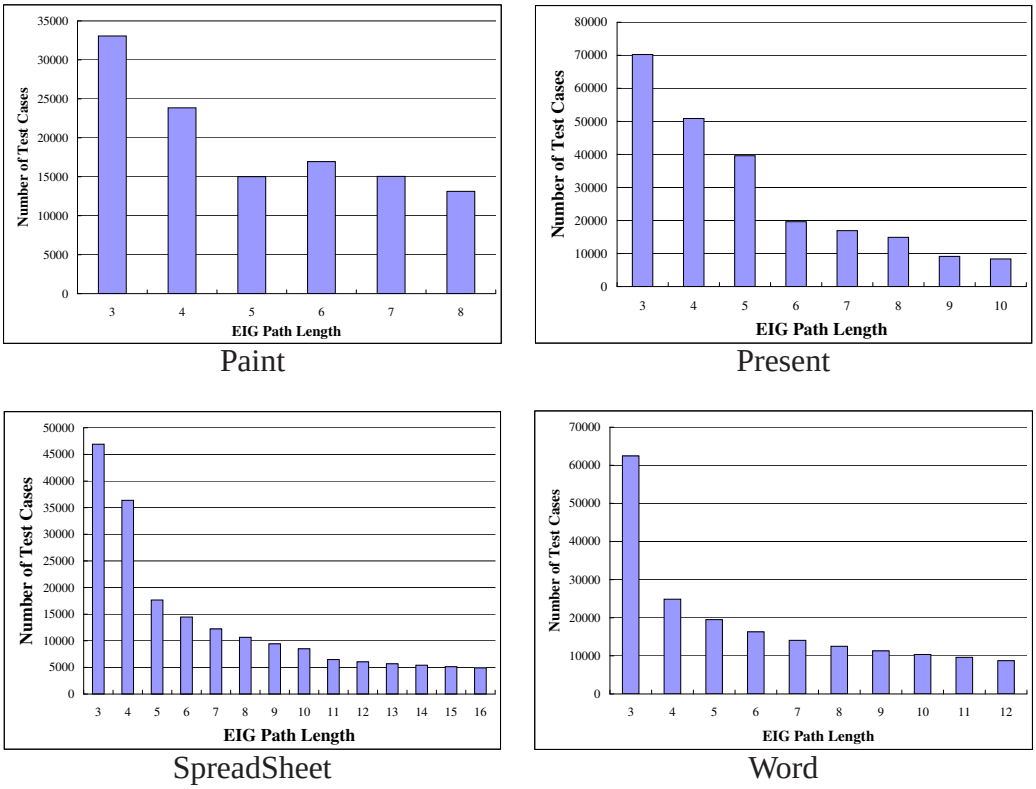


Figure 14: Histograms of Test Case Lengths in Pool

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 the test cases is shown as a histogram in Figure 14. 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

	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

Table 4: Test Pool and Average-Suite Sizes

coverage. The process was repeated 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 15 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

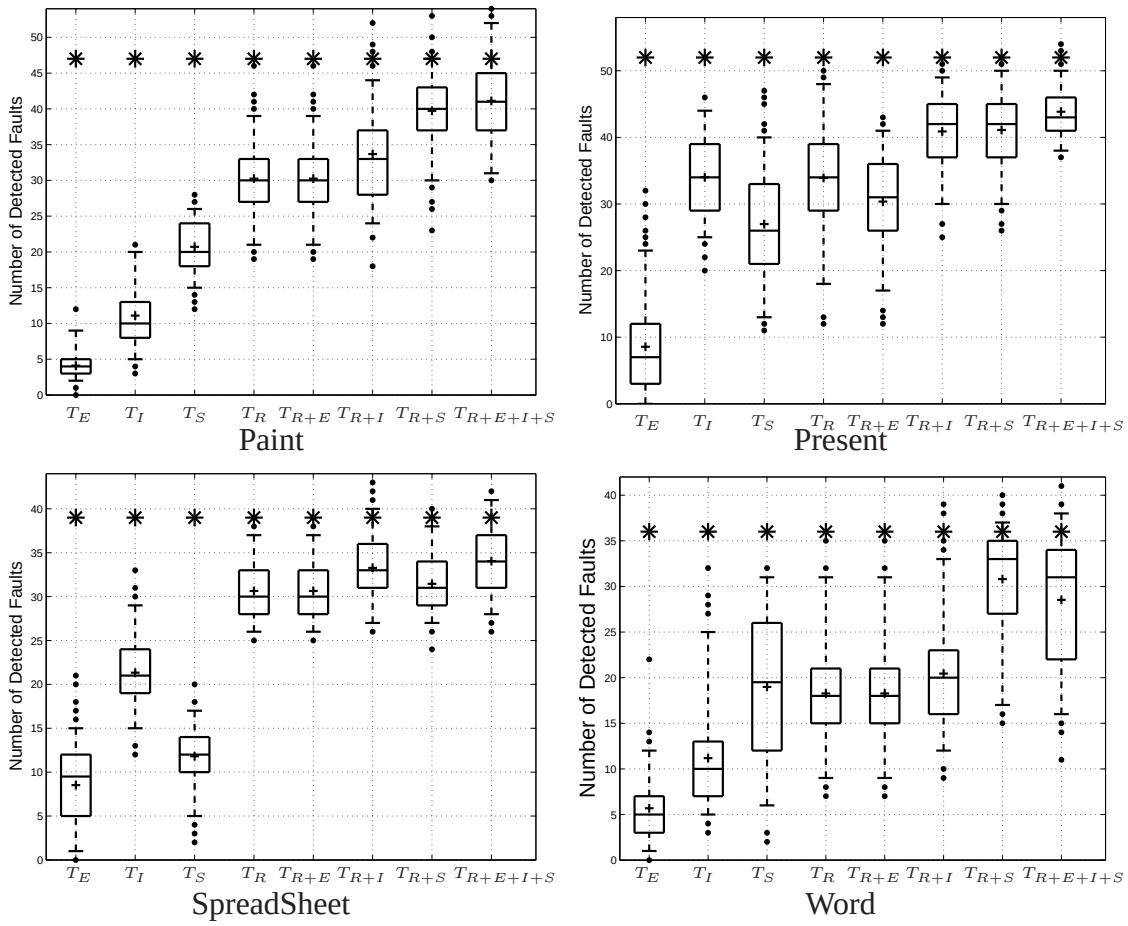


Figure 15: Fault Detection Distribution

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 17 shows eight histograms for TerpPresent, 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. Similar plots are shown in Figures 17–19 for Present, SpreadSheet, and Word, respectively.

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 test suites, T_{R+I} and T_{R+S} did better than the ESIG suites. In particular,

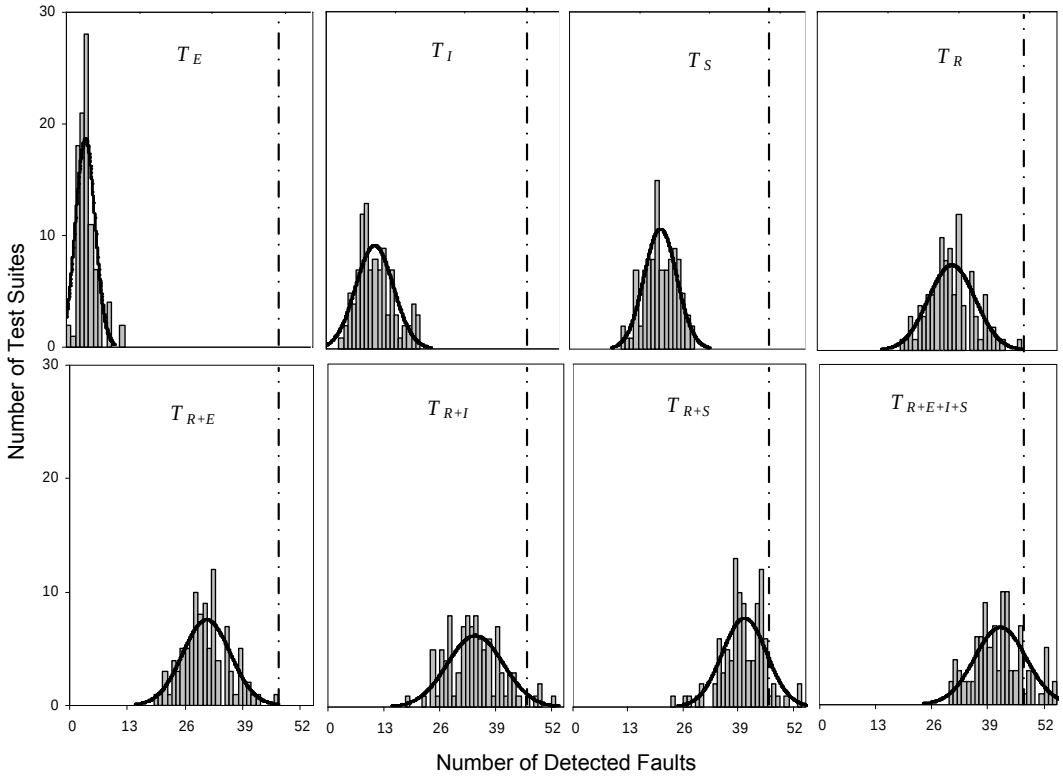


Figure 16: Histogram for Paint

4%, 0%, 6%, and 2% of the test suites for T_{R+I} did better than the ESIG suites, for Paint, Present, SpreadSheet, and Word, respectively; 4%, 0%, 2%, and 11% of the test suites for T_{R+S} did better than the ESIG suites, for Paint, Present, SpreadSheet, and Word, respectively. $T_{R+E+I+S}$ outperformed all the other suites in most cases; Specifically, 15%, 2%, 8%, and 9% of the test suites for $T_{R+E+I+S}$ did better than the ESIG suites, for Paint, Present, SpreadSheet, and Word, respectively. 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

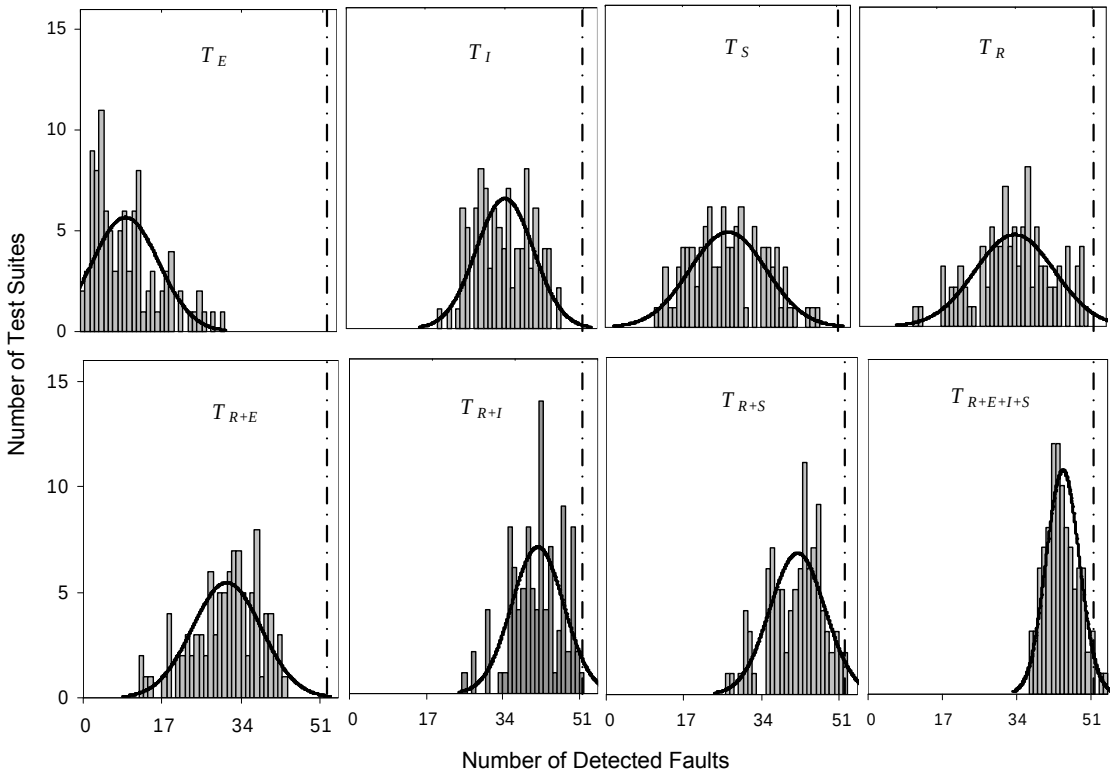


Figure 17: Histogram for Present

"number of faults detected") and all the other "similar" suites (100 values per distribution). We may 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 if certain conditions are met.

Many statistical tests are based upon the assumption that the data is normally distributed. These tests are called *parametric* tests; a common example includes the t-test [55]. Since all the computations in parametric tests are based on the data normality assumption, the significance level (p-value) cannot be considered reliable when the data distribution is not normal. Tests that do not make assumptions about distribution are referred to as *non-parametric* tests; these tests rank the dependent variable from low to high and then analyze the ranks [55].

Another factor that plays a role in choice of statistical tests is sample size. If the sample size is small, choosing a parametric test with non-normally distributed data, or choosing a non-parametric test with normally distributed data, will yield an inaccurate p-value. On the other hand, if the sample size is large, the distribution can be considered normal based upon the central limit

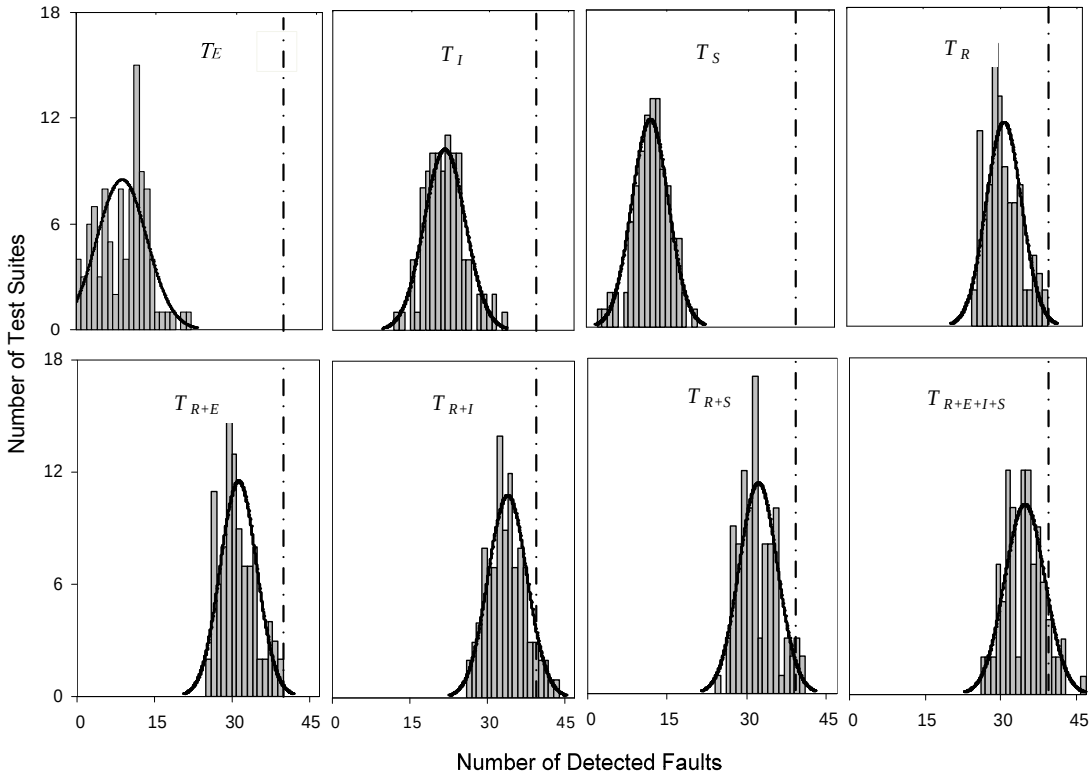


Figure 18: Histogram for SpreadSheet

theorem [55]. Consequently, in practice, with large sample size, the choice of parametric vs. non-parametric does not matter, although the p-value tends to be large for non-parametric tests; however, the discrepancy is small.

For illustration, the solid line superimposed on the histograms (Figures 17-19) shows the normal distribution approximation; informally, the data seems to follow the normal distribution quite well. There are several ways to formally determine normality of data. A popular way is to use a quantile-quantile (QQ) plot. A quantile is the fraction (or percentage) of points below a given value. For example, 70% of the data fall below the "0.7 quantile" point. A QQ-plot shows the quantiles of a data set (in our case the number of faults using different selection criteria) against the quantiles of a second data set (normal distribution). A mark (x,y) on the plot means that $f_2(x)$ and $f_1(y)$ share the same quantile, where f_1 and f_2 are the functions for the first and second data set. If the two sets come from a population with the same distribution, the plot will be along a $(x = y)$ straight line. The greater the departure from this reference line, the greater the evidence for

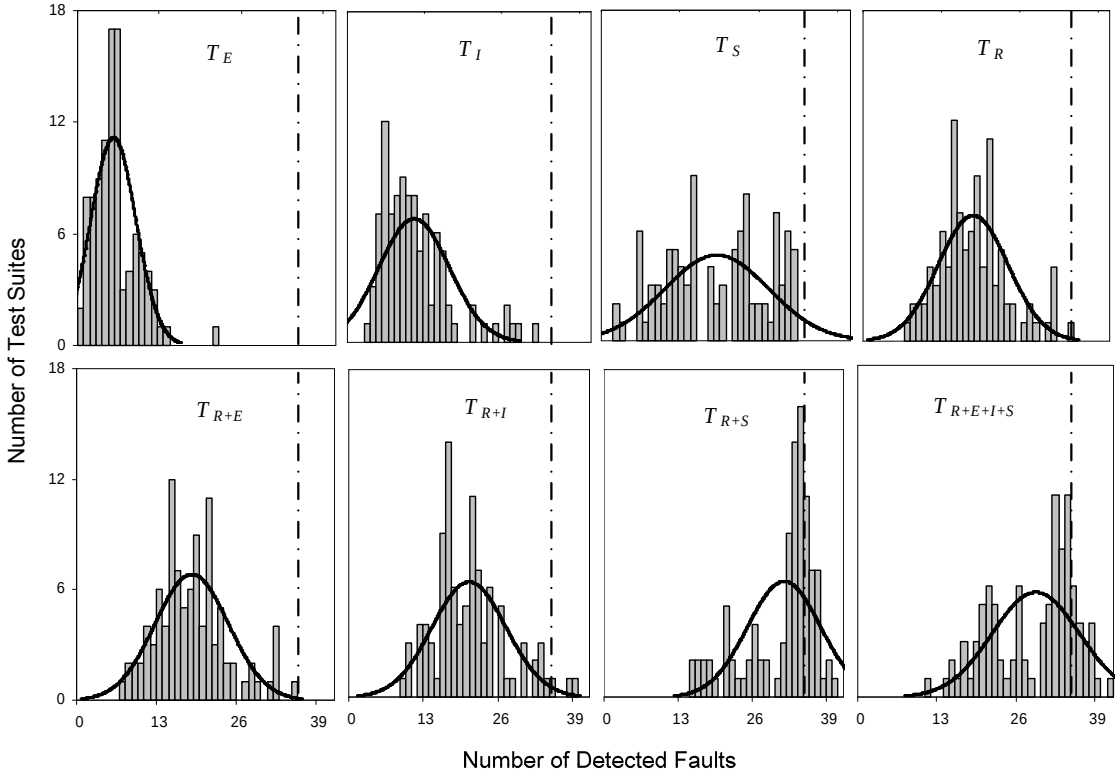


Figure 19: Histogram for Word

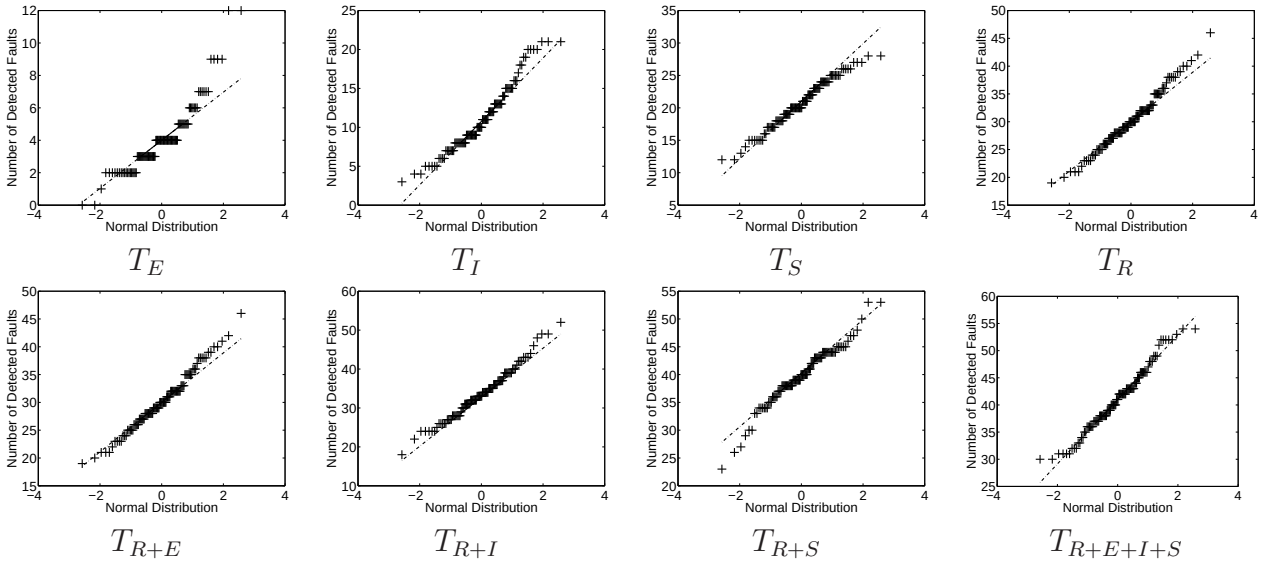


Figure 20: Quantile-Quantile plot for Paint

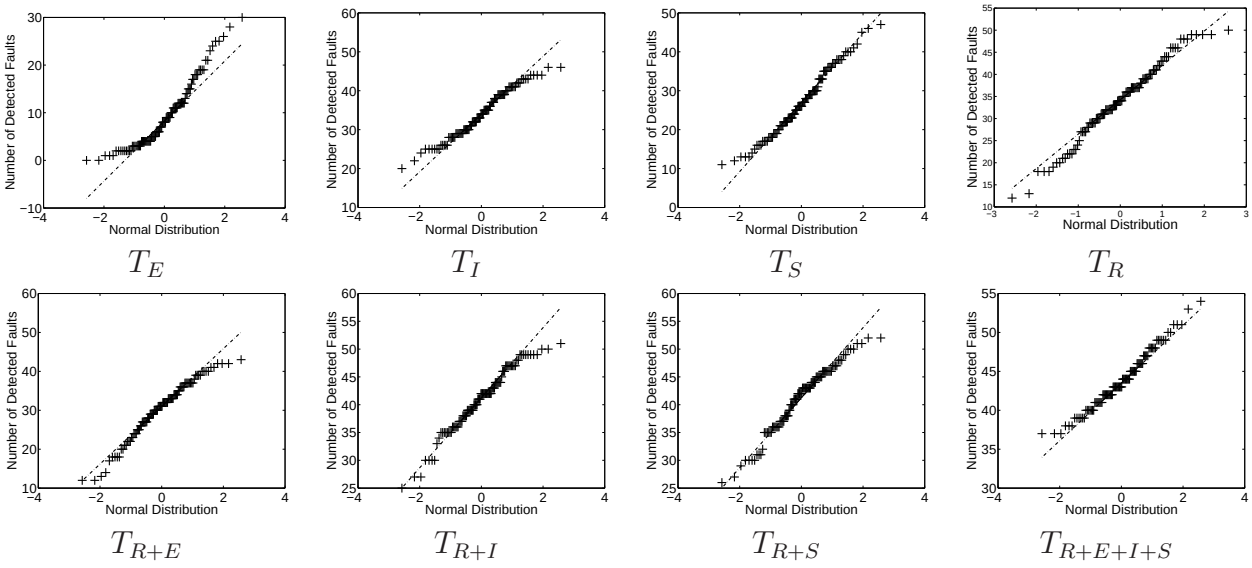


Figure 21: Quantile-Quantile plot for Present

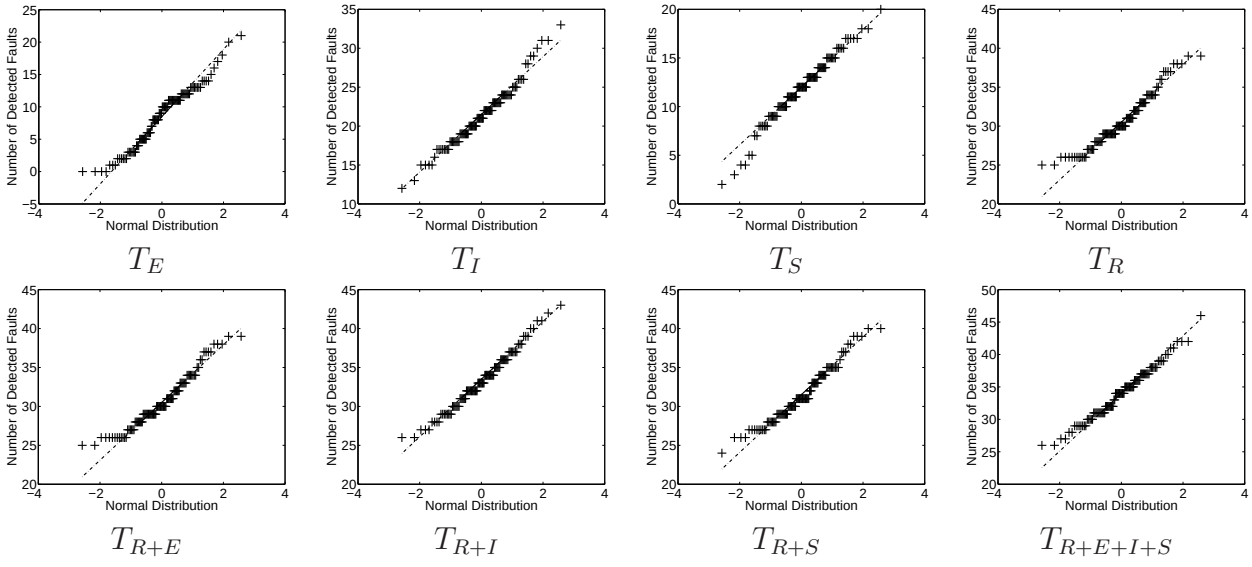


Figure 22: Quantile-Quantile plot for SpreadSheet

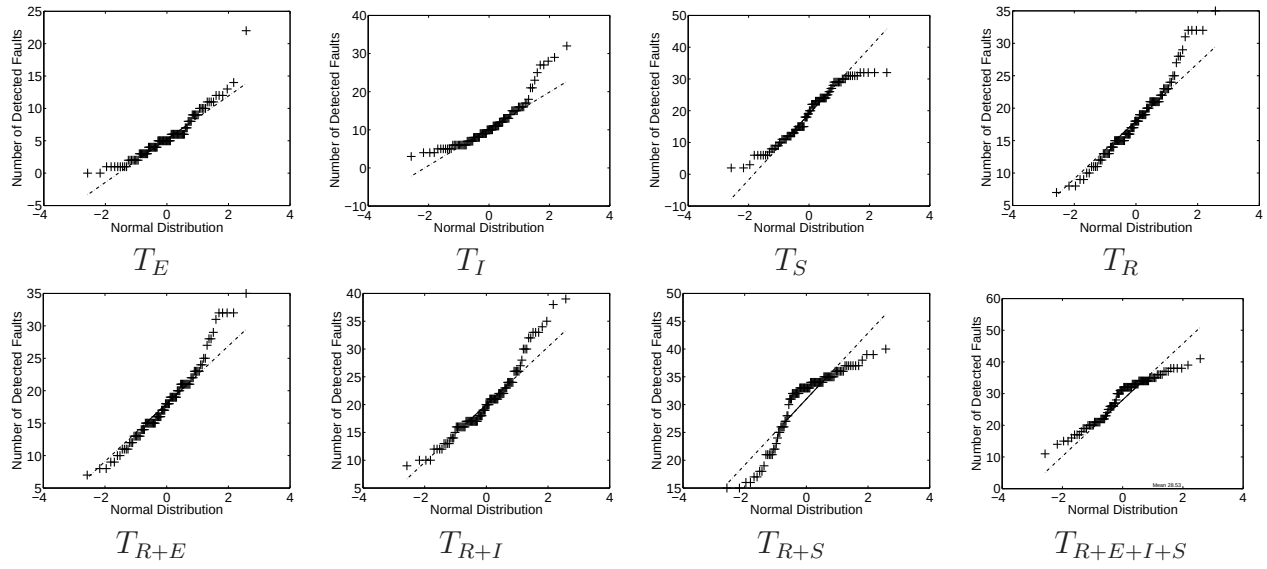


Figure 23: Quantile-Quantile plot for Word

the conclusion that the two data sets have come from populations with different distributions [60]. Figure 21 through Figure 22 show the quantile-quantile plots for each application. The x-axis is the normal distribution; the y-axis is the "number of faults" distribution. The QQ-plots more or less show a straight line, implying normality.

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 related to the RELAY model [61, 70] 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 [61], both are necessary conditions for fault detection. The data in Figure 15 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 24 shows data of the number of test cases that cover a fault-seeded statement, and the number of test cases in the pool that detect it. The figure is divided into four parts, one for each subject application. Each part is a 3-column table (long tables are wrapped); Column 1 is the seeded fault number, corresponding to a statement in which the fault was seeded; Column 2 is the number of test cases that detected the fault; Column 3 is the number of test cases that did not detect the fault even though they executed the fault-seeded statement; the fault number entry 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 test cases, 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

Paint					
158	1	1247	248	3	115
159	1	1173	255	3	115
65	3	721	256	3	115
36	4	869	261	3	115
52	5	830	262	3	115
64	5	772	42	10	381
33	6	856	47	21	762
34	7	843	46	10	359
90	6	722	93	21	748
179	1	120	98	21	730
180	1	118	68	21	729
246	1	117	54	11	379
249	1	117	45	12	383
191	1	116	30	4	127
192	1	116	31	4	127
89	7	773	43	12	373
211	1	110	67	23	707
212	1	110	124	3	89
50	10	934	97	27	771
101	24	2206	41	15	402
102	23	2093	87	5	127
85	8	720	73	28	703
22	9	796	72	31	744
77	9	735	44	15	339
57	4	311	86	11	180
21	11	845	14	1	15
51	13	874	120	15	220
56	5	331	32	10	122
19	13	850	253	9	109
20	14	891	119	19	216
177	2	125	259	11	107
96	12	711	59	33	314
55	6	354	58	36	337
100	12	704	123	25	207
247	2	116	128	39	270
250	2	116	127	42	267
260	2	116	134	853	3273
76	14	781	178	28	99
26	2	108	229	28	98
18	18	920	200	357	372
66	15	764	136	747	446
69	14	688	137	557	260
37	17	834			
74	16	755			
75	15	707			
92	15	698			
71	15	696			
99	17	769			
125	2	90			
126	2	90			
84	17	762			
48	16	717			
80	16	714			
94	16	705			
83	16	704			
82	17	743			
79	17	719			
88	1	42			
81	16	664			
70	18	746			
91	18	738			
251	2	82			
252	2	82			
17	24	978			
78	19	762			
95	19	761			
245	3	115			

Present					
214	1	262	94	45	479
215	21	4242	157	46	484
212	4	520	222	46	484
33	2	259	218	91	912
73	4	257	127	48	476
61	6	255	177	49	481
34	6	250	102	44	431
30	7	254	151	19	169
62	7	254	106	50	425
71	7	250	57	26	207
72	7	250	58	30	230
80	7	249	59	30	230
32	8	253	105	54	411
36	8	248	154	133	862
38	8	248	198	107	601
162	8	247	197	115	593
192	9	258	199	118	590
216	10	282	182	45	217
56	8	225	196	124	584
29	9	252	76	46	215
161	131	3119	115	96	428
40	25	580	89	103	421
8	177	4086	48	53	210
167	121	2780	47	54	209
66	11	250	99	98	377
75	11	250	159	682	2479
166	138	3112	55	58	205
194	23	517	152	715	2446
165	139	3111	123	106	357
178	12	260	118	60	202
185	12	255	122	63	199
43	27	568	211	169	355
6	198	4065	224	168	209
5	177	3630	93	301	223
9	206	4057	90	310	214
7	208	4055	202	313	211
41	30	575	117	314	210
85	220	4043	201	322	202
12	223	4040	203	332	192
16	32	573	200	333	191
78	228	4035	147	933	325
187	29	508	148	936	322
104	26	439			
17	34	571			
229	75	1216			
221	32	498			
96	32	492			
137	15	229			
153	267	3996			
95	33	491			
184	35	495			
52	18	245			
227	37	490			
111	37	487			
186	37	487			
150	13	168			
82	39	491			
158	39	491			
226	39	491			
225	41	489			
128	41	483			
156	57	668			
220	42	488			
108	38	437			
176	44	486			
219	84	919			
79	45	485			

SpreadSheet		
226	12	1130
232	13	1048
33	172	13445
34	179	13438
163	129	9679
233	14	1037
164	133	9675
218	35	2429
180	67	4512
15	25	1681
200	25	1681
214	37	2427
220	37	2427
160	69	4510
161	70	4509
133	12	772
182	71	4508
230	17	1053
47	2	123
48	2	123
17	28	1678
28	311	18223
231	19	1051
8	57	3150
208	77	4224
192	83	4496
234	19	1025
32	19	1010
39	87	4481
210	82	4219
2	206	10485
54	393	19389
209	89	4212
221	67	3140
20	36	1670
204	69	3138
211	93	4208
41	34	1512
38	105	4463
191	74	3100
45	3	122
130	20	786
5	84	3123
53	53	1916
224	10	300
37	20	576
55	68	1074
225	201	3006
56	76	1066
36	44	569
44	17	164
18	237	1469
229	738	404
30	745	397
228	746	396
42	1030	516
43	1030	516
31	746	362
132	615	191
131	618	188
78	546	157
79	546	157
75	1155	330
136	1161	324
135	614	170
134	615	169
76	1088	296
77	544	137

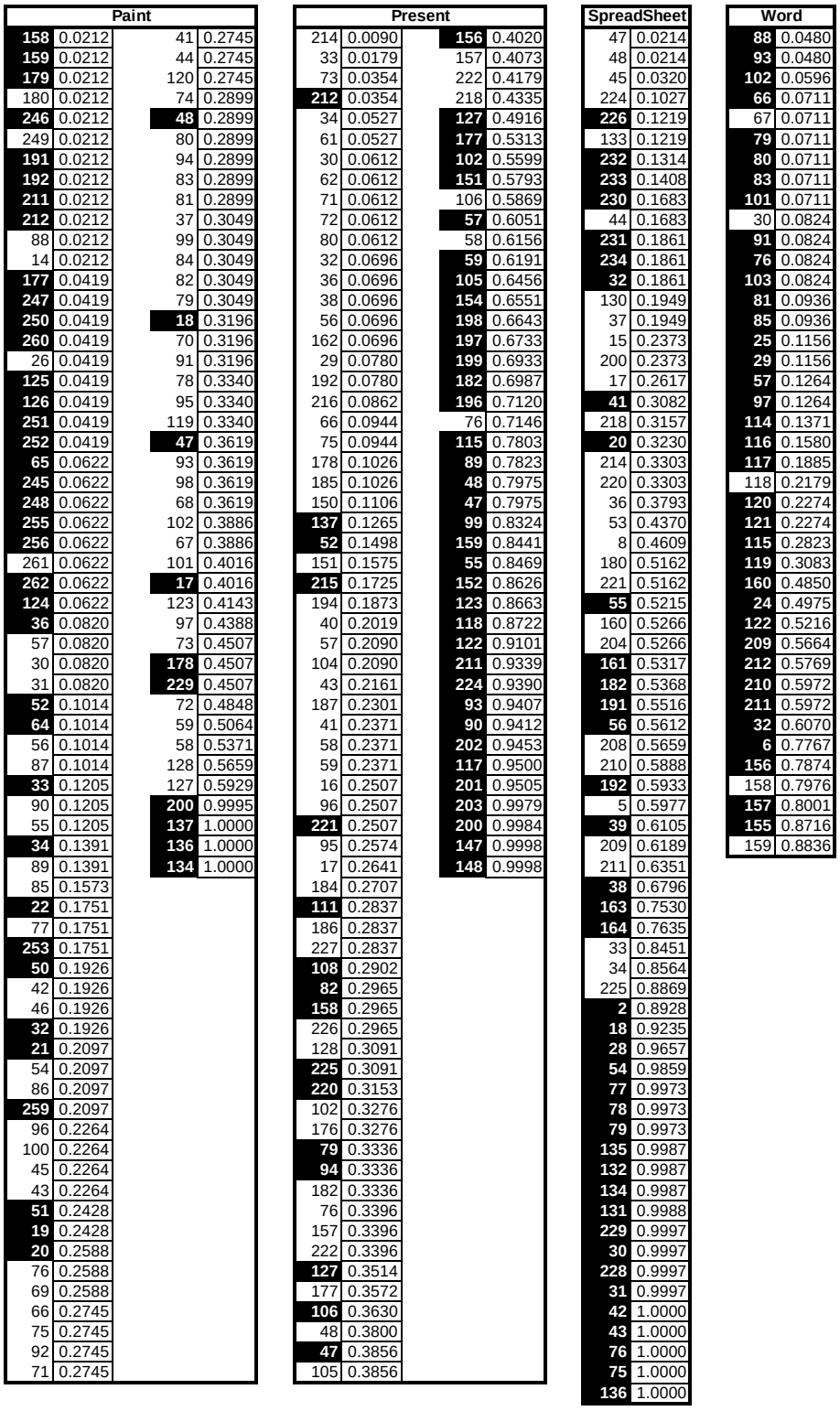
Word		
156	126	2336
209	68	1238
157	131	2331
158	130	2146
88	4	65
93	4	65
210	74	1185
211	74	1185
212	70	1072
155	167	2295
159	175	2101
102	5	57
30	7	75
66	6	63
67	6	63
79	6	63
80	6	63
83	6	63
101	6	56
91	7	62
76	7	55
103	7	55
81	8	61
85	8	61
25	10	72
29	10	72
114	12	72
57	11	58
97	11	58
116	14	70
117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

Figure 24: Test Cases Covered Faulty Statements and Their Fault Detection.

are independent). The data in Figure 24 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 15. 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). 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 by the user. Because there are many permutations 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.

Forth, several faults were detected by very few test cases. 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 (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



Paint			
158	0.0212	41	0.2745
159	0.0212	44	0.2745
179	0.0212	120	0.2745
180	0.0212	74	0.2899
246	0.0212	48	0.2899
249	0.0212	80	0.2899
191	0.0212	94	0.2899
192	0.0212	83	0.2899
211	0.0212	81	0.2899
212	0.0212	37	0.3049
88	0.0212	99	0.3049
14	0.0212	84	0.3049
177	0.0419	82	0.3049
247	0.0419	79	0.3049
250	0.0419	18	0.3196
260	0.0419	70	0.3196
26	0.0419	91	0.3196
125	0.0419	78	0.3340
126	0.0419	95	0.3340
251	0.0419	119	0.3340
252	0.0419	47	0.3619
65	0.0622	93	0.3619
245	0.0622	98	0.3619
248	0.0622	68	0.3619
255	0.0622	102	0.3886
256	0.0622	67	0.3886
261	0.0622	101	0.4016
262	0.0622	17	0.4016
124	0.0622	123	0.4143
36	0.0820	97	0.4388
57	0.0820	73	0.4507
30	0.0820	178	0.4507
31	0.0820	229	0.4507
52	0.1014	72	0.4848
64	0.1014	59	0.5064
56	0.1014	58	0.5371
87	0.1014	128	0.5659
33	0.1205	127	0.5929
90	0.1205	200	0.9995
55	0.1205	137	1.0000
34	0.1391	136	1.0000
89	0.1391	134	1.0000
85	0.1573		
22	0.1751		
77	0.1751		
253	0.1751		
50	0.1926		
42	0.1926		
46	0.1926		
32	0.1926		
21	0.2097		
54	0.2097		
86	0.2097		
259	0.2097		
96	0.2264		
100	0.2264		
45	0.2264		
43	0.2264		
51	0.2428		
19	0.2428		
20	0.2588		
76	0.2588		
69	0.2588		
66	0.2745		
75	0.2745		
92	0.2745		
71	0.2745		

Present			
214	0.0090	156	0.4020
33	0.0179	157	0.4073
73	0.0354	222	0.4179
212	0.0354	218	0.4335
34	0.0527	127	0.4916
61	0.0527	177	0.5313
30	0.0612	102	0.5599
62	0.0612	151	0.5793
71	0.0612	106	0.5869
72	0.0612	57	0.6051
80	0.0612	58	0.6156
32	0.0696	59	0.6191
36	0.0696	105	0.6456
38	0.0696	154	0.6551
56	0.0696	198	0.6643
162	0.0696	197	0.6733
29	0.0780	199	0.6933
192	0.0780	182	0.6987
216	0.0862	196	0.7120
66	0.0944	76	0.7146
75	0.0944	115	0.7803
178	0.1026	89	0.7823
185	0.1026	48	0.7975
150	0.1106	47	0.7975
137	0.1265	99	0.8324
52	0.1498	159	0.8441
151	0.1575	55	0.8469
215	0.1725	152	0.8626
194	0.1873	123	0.8663
40	0.2019	118	0.8722
57	0.2090	122	0.9101
104	0.2090	211	0.9339
43	0.2161	224	0.9390
187	0.2301	93	0.9407
41	0.2371	90	0.9412
58	0.2371	202	0.9453
59	0.2371	117	0.9500
16	0.2507	201	0.9505
96	0.2507	203	0.9979
221	0.2507	200	0.9984
95	0.2574	147	0.9998
17	0.2641	148	0.9998
184	0.2707		
111	0.2837		
186	0.2837		
227	0.2837		
108	0.2902		
82	0.2965		
158	0.2965		
226	0.2965		
128	0.3091		
225	0.3091		
220	0.3153		
102	0.3276		
176	0.3276		
79	0.3336		
94	0.3336		
182	0.3336		
76	0.3396		
157	0.3396		
222	0.3396		
127	0.3514		
177	0.3572		
106	0.3630		
48	0.3800		
47	0.3856		
105	0.3856		

SpreadSheet	
47	0.0214
48	0.0214
45	0.0320
224	0.1027
226	0.1219
133	0.1219
232	0.1314
233	0.1408
230	0.1683
44	0.1683
231	0.1861
234	0.1861
32	0.1861
130	0.1949
37	0.1949
15	0.2373
200	0.2373
17	0.2617
41	0.3082
218	0.3157
20	0.3230
214	0.3303
220	0.3303
36	0.3793
53	0.4370
8	0.4609
180	0.5162
221	0.5162
55	0.5215
160	0.5266
204	0.5266
161	0.5317
182	0.5368
191	0.5516
56	0.5612
208	0.5659
210	0.5888
192	0.5933
5	0.5977
39	0.6105
209	0.6189
211	0.6351
38	0.6796
163	0.7530
164	0.7635
33	0.8451
34	0.8564
225	0.8869
2	0.8928
18	0.9235
28	0.9657
54	0.9859
77	0.9973
78	0.9973
79	0.9973
135	0.9987
132	0.9987
134	0.9987
131	0.9988
229	0.9997
30	0.9997
228	0.9997
31	0.9997
42	1.0000
43	1.0000
76	1.0000
75	1.0000
136	1.0000

Word	
88	0.0480
93	0.0480
102	0.0596
66	0.0711
67	0.0711
79	0.0711
80	0.0711
83	0.0711
101	0.0711
30	0.0824
91	0.0824
76	0.0824
103	0.0824
81	0.0936
85	0.0936
25	0.1156
29	0.1156
57	0.1264
97	0.1264
114	0.1371
116	0.1580
117	0.1885
118	0.2179
120	0.2274
121	0.2274
115	0.2823
119	0.3083
160	0.4850
24	0.4975
122	0.5216
209	0.5664
212	0.5769
210	0.5972
211	0.5972
32	0.6070
6	0.7767
156	0.7874
158	0.7976
157	0.8001
155	0.8716
159	0.8836

Figure 25: Probability of Detecting Faults by Random Test Cases.

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 detected by more than 84 test cases in the pool, is detected by at least one test case in T_S is 0.49 – this is quite high. In Figure 25, we show the probability that a random suite of its corresponding ESI-suite size would detect a particular fault. 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 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 test oracle was not capable of examining these parts of the GUI,
2. very few faults caused failures in non-GUI output,
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 5. The large number of "Ignored Widget Properties" 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

	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

Table 5: Undetected Faults Classification

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 studies are subject to threats to validity. These threats need to be considered in order to assess their impact on the results. 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 for obtaining 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 – using 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 current test-case generation algorithms output a set of all possible event sequences bounded by a pre-determined length. As the goal of this work is to generate multi-way interactions among GUI events, other techniques (such as *covering arrays* [85]) designed to minimize the number of test cases while retaining high interaction coverage will be explored.

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 [9].

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 [2]; 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] AHO, A. V., DAHBURA, A. T., LEE, D., AND UYAR, M. U. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *Conformance testing methodologies and architectures for OSI protocols* (1995), 427–438.

[2] ALESSANDRO MARCHETTO, P. T., AND RICCA, F. 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.

[3] ALLEN, F. E. Control flow analysis. In *Proceedings of a symposium on Compiler optimization* (1970), pp. 1–19.

[4] APFELBAUM, L. Automated functional test generation. In *Autotestcon '95 Conference* (1995), IEEE.

[5] AUGUSTON, M., MICHAEL, J. B., AND SHING, M.-T. 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, 2005), ACM Press, pp. 1–6.

[6] BARNETT, M., GRIESKAMP, W., NACHMANSON, L., SCHULTE, W., TILLMANN, N., AND VEANES, M. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich [59], pp. 252–266.

[7] BELLI, F. Finite-state testing and analysis of graphical user interfaces. In *ISSRE* (2001), IEEE Computer Society, pp. 34–43.

[8] BELLI, F. Finite-state testing and analysis of GUIs. In *Proceedings of the 12th International Symposium on Software Reliability Engineering* (2001), pp. 34–43.

[9] BELLI, F., AND BUDNIK, C. J. Test minimization for human-computer interaction. *Applied Intelligence 26*, 2 (2007), 161–174.

[10] BELLI, F., BUDNIK, C. J., AND WHITE, L. Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles. *Softw. Test. Verif. Reliab. 16*, 1 (2006), 3–32.

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

[12] BLACKBURN, M. R., BUSSER, R., NAUMAN, A., AND CHANDRAMOULI, R. Model-based approach to security test automation. In *13th International Symposium on Software Reliability Engineering (ISSRE)* (Annapolis, MD, 2002).

[13] BLACKBURN, M. R., BUSSER, R., NAUMAN, A., KNICKERBOCKER, R., AND KASUDA, R. Mars polar lander fault identification using model-based testing. In *ICECCS* (2002), IEEE Computer Society, pp. 163–.

[14] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. 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.

[15] BRIAND, L. C., LABICHE, Y., AND WANG, Y. Using simulation to empirically investigate test coverage criteria based on statechart. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE Computer Society, pp. 86–95.

[16] CAMPBELL, C., GRIESKAMP, W., NACHMANSON, L., SCHULTE, W., TILLMANN, N., AND VEANES, M. Model-based testing of object-oriented reactive systems with spec explorer., May 2005.

[17] CHEN, W.-H., LU, C.-S., BROZOVSKY, E. R., AND WANG, J.-T. An optimization technique for protocol conformance testing using multiple uio sequences. *Inf. Process. Lett. 36*, 1 (1990), 7–11.

[18] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE trans. on Software Engineering SE-4, 3* (1978), 178–187.

[19] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng. 4*, 3 (1978), 178–187.

[20] CLARKE, J. M. Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference* (Portland, OR, May 1998), Pnsqc/Pacific Agenda.

[21] D'AMORIM, M., PACHECO, C., XIE, T., MARINOV, D., AND ERNST, M. D. 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).

[22] DUSTIN, E., RASHKA, J., AND PAUL, J. *Automated software testing: introduction, management, and performance*. 1999.

[23] ESMELIOGLU, S., AND APFELBAUM, L. Automated test generation, execution, and reporting. In *Proceedings of Pacific Northwest Software Quality Conference* (Portland, OR, Oct 1997), Pnsqc/Pacific Agenda, pp. 127–142.

[24] FARCHI, E., HARTMAN, A., AND PINTER, S. S. Using a model-based test generator to test for standard conformance. *IBM Systems Journal 41*, 1 (2002), 89–110.

[25] FERGUSON, R., AND KOREL, B. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5*, 1 (1996), 63–86.

[26] GALLAGHER, M. J., AND NARASIMHAN, V. L. Adtest: A test data generation suite for Ada software systems. *IEEE Trans. Software Eng. 23*, 8 (1997), 473–484.

[27] GUPTA, N., MATHUR, A. P., AND SOFFA, M. L. Automated test data generation using an iterative relaxation method. In *SIGSOFT FSE* (1998), pp. 231–244.

[28] HONG, H. S., KWON, Y. R., AND CHA, S. D. Testing of object-oriented programs based on finite state machines. In *APSEC* (1995), IEEE Computer Society, pp. 234–.

[29] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[30] HOWE, A., VON MAYRHAUSER, A., AND MRAZ, R. T. Test case generation as an AI planning problem. *Automated Software Engineering 4* (1997), 77–106.

[31] IMANIAN, J. A. *Automatic Test Case Generation for Reactive Software Systems Based on Environment Models*. PhD thesis, Naval Postgraduate School, Monterey, CA, June 2005.

[32] IPATE, F., AND HOLCOMBE, M. Specification and testing using generalized machines: a presentation and a case study. *Software Testing, Verification and Reliability* (1998), 61–81.

[33] IPATE, F., AND HOLCOMBE, M. Complete testing from a stream x-machine specification. *Fundam. Inf. 64*, 1-4 (2004), 205–216.

[34] JORGENSEN, A. A., AND WHITTAKER, J. A. An application program interface (API) testing method. In *STAREast Conference* (May 2000).

[35] KASIK, D. J., AND GEORGE, H. G. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground* (New York, 13–18 Apr. 1996), ACM Press, pp. 244–251.

[36] KOOPMAN, P. W. M., PLASMEIJER, R., AND ACHTEN, P. Model-based testing of thin-client web applications. In *FATES/RV* (2006), K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds., vol. 4262 of *Lecture Notes in Computer Science*, Springer, pp. 115–132.

[37] KOREL, B. Automated software test data generation. *IEEE Trans. Software Eng. 16*, 8 (1990), 870–879.

[38] LEE, N. H., AND CHA, S. D. Generating test sequences from a set of mscs. *Computer Networks 42*, 3 (2003), 405–417.

[39] LEOW, W. K., KHOO, S. C., AND SUN, Y. Automated generation of test programs from closed specifications of classes and test cases. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 96–105.

[40] LUCIO, L., PEDRO, L., AND BUCHS, D. A methodology and a framework for model-based testing. In *RISE* (2004), N. Guelfi, Ed., vol. 3475 of *Lecture Notes in Computer Science*, Springer, pp. 57–70.

[41] MATHUR, A. P. *Foundations of Software Testing: Fundamental Algorithms and Techniques*. Pearson Education., 2008.

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

[43] MCMASTER, S., AND MEMON, A. M. Call-stack coverage for GUI test-suite reduction. *IEEE Trans. Softw. Eng.* (2008).

[44] MEMON, A. M. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability 17*, 3 (2007), 137–157.

[45] MEMON, A. M., NAGARAJAN, A., AND XIE, Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution 17*, 1 (Jan. 2005), 27–64.

[46] MEMON, A. M., NAGARAJAN, A., AND XIE, Q. Automating regression testing for evolving gui software. *Journal of Software Maintenance 17*, 1 (2005), 27–64.

[47] MEMON, A. M., POLLACK, M. E., AND SOFFA, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering 27*, 2 (Feb. 2001), 144–155.

[48] MEMON, A. M., POLLACK, M. E., AND SOFFA, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng. 27*, 2 (2001), 144–155.

[49] MEMON, A. M., AND XIE, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng. 31*, 10 (2005), 884–896.

[50] MEMON, A. M., AND XIE, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering 31*, 10 (Oct. 2005), 884–896.

[51] MEMON, A. M., AND XIE, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng. 31*, 10 (2005), 884–896.

[52] MICHAEL, C. C., MCGRAW, G., AND SCHATZ, M. Generating software test data by evolution. *IEEE Trans. Software Eng. 27*, 12 (2001), 1085–1110.

[53] MICSKEI, Z., AND MAJZIK, I. Model-based automatic test generation for event-driven embedded systems using model checkers. In *DepCoS-RELCOMEX* (2006), IEEE Computer Society, pp. 191–198.

[54] MILLER, W., AND SPOONER, D. L. Automatic generation of floating-point test data. *IEEE Trans. Software Eng. 2*, 3 (1976), 223–226.

[55] MOTULSKY, H. *Intuitive Biostatistics*. Oxford University Press, 1995.

[56] MYERS, B. A., AND ROSSON, M. B. Survey on user interface programming. In *CHI* (1992), pp. 195–202.

[57] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (Minneapolis, MN, USA, May 23–25, 2007), IEEE Computer Society, pp. 396–405.

[58] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering* (Minneapolis, MN, USA, May 23–25, 2007).

[59] PETRENKO, A., AND ULRICH, A., Eds. *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003* (2004), vol. 2931 of *Lecture Notes in Computer Science*, Springer.

[60] Quantile-Quantile Plot. http://www.itl.nist.gov/div898/handbook/eda/section3/qqplot.htm .

[61] RICHARDSON, D. J., AND THOMPSON, M. C. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Trans. Softw. Eng. 19*, 6 (1993), 533–553.

[62] ROSEN, B. K. Data flow analysis for procedural languages. *J. ACM 26*, 2 (1979), 322–344.

[63] ROTHERMEL, G., ELBAUM, S., MALISHEVSKY, A. G., KALLAKURI, P., AND QIU, X. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol. 13*, 3 (2004), 277–331.

[64] ROUNTEV, A., KAGAN, S., AND GIBAS, M. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2004), pp. 14–16.

[65] SARIKAYA, B. Conformance testing: architectures and test sequences. *Comput. Netw. ISDN Syst. 17*, 2 (1989), 111–126.

[66] SCHEETZ, M., VON MAYRHAUSER, A., FRANCE, R., DAHLMAN, E., AND HOWE, A. E. Generating test cases from an OO model with an ai planning system. In *Proceedings of The 10th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 250–259.

[67] SHEHADY, R. K., AND SIEWIOREK, D. P. 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, 1997), IEEE Computer Society, p. 80.

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

[69] SIRER, E. G., AND BERSHAD, B. N. Using production grammars in software testing. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages* (New York, NY, USA, 1999), ACM Press, pp. 1–13.

[70] THOMPSON, M. C., RICHARDSON, D. J., AND CLARKE, L. A. An information flow model of fault detection. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1993), ACM, pp. 182–192.

[71] VON MAYRHAUSER, A., AND CRAWFORD-HINES, S. Automated testing support for a robot tape library. In *Proceedings of The Fourth International Software Reliability Engineering Conference* (Washington, DC, USA, Nov. 1993), IEEE Computer Society, pp. 6–14.

[72] VON MAYRHAUSER, A., MRAZ, R. T., AND WALLS, J. Domain based regression testing. In *Proceedings of The International Conference on Software Maintenance* (Washington, DC, USA, 1994), IEEE Computer Society, pp. 26–35.

[73] WHITE, L., AND ALMEZEN, H. 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, 2000), IEEE Computer Society, p. 110.

[74] WHITE, L., AND ALMEZEN, H. Generating test cases for gui responsibilities using complete interaction sequences. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 110.

[75] WHITTAKER, J. A. *Markov chain techniques for software testing and reliability analysis*. PhD thesis, University of Tennessee, Knoxville, TN, USA, 1992.

[76] WHITTAKER, J. A. Stochastic software testing. *Ann. Software Eng. 4* (1997), 115–131.

[77] WHITTAKER, J. A., AND THOMASON, M. G. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng. 20*, 10 (1994), 812–824.

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

[79] WOIT, D. M. 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, 1993), ACM Press, pp. 2–10.

[80] XIE, Q., AND MEMON, A. M. Automated model-based testing of community-driven open source GUI applications. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 145–154.

[81] XIE, Q., AND MEMON, A. M. Studying the characteristics of a 'good' GUI test suite. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)* (Nov. 2006), IEEE Computer Society Press.

[82] XIE, Q., AND MEMON, A. M. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology 16*, 1 (2007), 4.

[83] XIE, T., AND NOTKIN, D. Mutually enhancing test generation and specification inference. In Petrenko and Ulrich [59], pp. 60–69.

[84] XIE, T., AND NOTKIN, D. Tool-assisted unit-test generation and selection based on operational abstractions. *Autom. Softw. Eng. 13*, 3 (2006), 345–371.

[85] YILMAZ, C., COHEN, M. B., AND PORTER, A. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of the 2004 international symposium on Software testing and analysis* (2004), pp. 45–54.

[86] YUAN, X., AND MEMON, A. M. 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, May 23–25, 2007), IEEE Computer Society, pp. 396–405.