Using GUI Run-Time State as Feedback to Generate Test Cases

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

Abstract

This paper presents a new automated model-driven technique to generate test cases by using feedback from the execution of a "seed test suite" on an application under test (AUT). The test cases in the seed suite are designed to be generated automatically and executed very quickly. During their execution, feedback obtained from the AUT's run-time state is used to generate new, "improved" test cases. The new test cases subsequently become part of the seed suite. This "anytime technique" continues iteratively, generating and executing additional test cases until resources are exhausted or testing goals have been met.

The feedback-based technique is demonstrated for automated testing of graphical user interfaces (GUIs). An existing abstract model of the GUI is used to automatically generate the seed test suite. It is executed; during its execution, state changes in the GUI pinpoint important relationships between GUI events, which evolve the model and help to generate new test cases. Together with a reverse-engineering algorithm used to obtain the initial model and seed suite, the feedback-based technique yields a fully automatic, end-to-end GUI testing process. A feasibility study on four large fielded open-source software (OSS) applications demonstrates that this process is able to significantly improve existing techniques and help identify/report serious problems in the OSS. In response, these problems have been fixed by the developers of the OSS in subsequent versions.

1 Introduction

Automated test case generation (ATCG) has become increasingly popular due to its potential to reduce testing cost and increase software quality [4]. A typical approach used for ATCG is to create an abstract model of the software (*e.g.*, state-machine model [1, 17, 18]) 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. Consequently, a few researchers have developed automated feedback-based techniques to augment the models [2, 3, 5–7, 9, 13, 14, 20]. 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 [5–7, 9, 13, 14]. Few techniques use feedback from the AUT's *run-time state* to generate additional test cases, *e.g.*, in the form of outcomes of programmer-supplied predicates in the code to cover all non-isomorphic inputs [2], and in the form of operational abstractions to cover increased program behaviors [3, 20].

This paper presents a new feedback-based technique for automated testing of graphical user interfaces (GUIs). The feedback is an abstraction of the run-time state of GUI widgets; the goal is to test multi-way interactions between GUI events. The nature of GUIs, their test cases (sequences of GUI events that exercise GUI widgets), 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 an extremely important problem because GUIs are used as front-ends to most software applications and GUIs constitute as much as half of software's code [15]. A functionally correct GUI is necessary for trouble-free execution of the application's underlying "business logic." The event-driven nature of GUIs creates new challenges for testing that require the development of new solutions [17, 18].

Second, our existing 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 [10]. Our previous empirical studies showed that although these test cases reveal a large number of GUI faults, additional faults may be detected by testing certain types of multi-way interactions [12]. 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 test case length (number of events). A practical alternative is to identify sets 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 set. We will use the feedback-based technique to *automatically* identify such sets of events. Third, our automatically generated smoke test cases may be used as the basis for feedback collection. These smoke test cases form the seed suite. Finally, our existing tools can be easily adapted to monitor and store the run-time state of the GUI.

The new feedback-based technique has been used in a fully automatic end-to-end process for a specific type of

GUI testing. A seed test suite is generated automatically using an existing *event-interaction graph* (EIG) model of the GUI. The seed 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. A new model of the GUI, called the *Event Semantic Interaction Graph* (ESIG) is constructed automatically and used to generate additional test cases. This entire process, including the scripts required to set up, execute, and tear down test cases, has been implemented and executes without human intervention. A feasibility study has been conducted on four fielded, well-tested, and popular GUI-based Java applications downloaded from SourceForge. The results of this study show that the feedback-based technique improves our existing techniques with little additional cost. The ESI relationship is successful at identifying complex interactions among GUI event handlers that lead to serious failures. The failures were reported on the SourceForge bug reporting site; in response, the developers fixed some of the bugs. The developers had never detected our reported failures before because their own tools and testing processes were unable to comprehensively and automatically test the applications.

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, and
- immersion of the feedback-based technique into a fully automatic end-to-end GUI testing process and demonstration of its effectiveness on fielded applications.

The next section introduces basic GUI concepts and reviews the EIG model that forms the basis of the new ESIG model. Section 3 gives an overview of existing GUI-testing techniques. 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 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.[1] 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); hence 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 $v \in V_p$. A set of states S_I is called the *valid initial state set* for a particular GUI if the GUI may be in any state $S_i \in S_I$ when it is first invoked. The state of a GUI is not static; events 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. 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.

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

The seed test suite is generated using an *event-interaction graph* (EIG) model of the GUI, which is obtained automatically using a standard GUI-reverse-engineering algorithm [12]. 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 [12]). The remaining events, called system-interaction events, do not manipulate the structure of the GUI. Directed edges between nodes encode *execution paths*, *i.e.*, sequences of events, in the GUI. For example, an edge (e_x, e_y) shows that e_y may be executed after e_x along some *execution path*.

The basic motivation behind using a graph model to represent a GUI is that various types of existing graph-traversal algorithms (with well-known run-time complexities) may be used to "walk" the graph, enumerating the events along the visited nodes, thereby generating test cases. In earlier research [12], an algorithm called `GenTestCases` was implemented that returned all possible paths (sequences of events) in the graph bounded to a specific length of 2. These length-2 sequences are said to test all *two-way interactions* between the EIG events. This research will generate test cases for *multi-way interactions*, *i.e.*, longer paths in an

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

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

EIG. Because EIG nodes do not represent events to open or close menus, or open windows, the sequences obtained from the EIG may not be executable. At execution time, other events needed to reach the EIG events are automatically generated, yielding an executable test case [12]. 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.

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* $< e_1; e_2 >$.

3 Related Work

To the best of our knowledge, there is no prior work on the use of run-time state as feedback for GUI testing. However, several researchers have used feedback from test-execution results to automatically generate additional test cases for non-GUI software. This section summarizes some of their work, followed by a discussion of model-based GUI-testing techniques.

Feedback-based Test Case Generation: The work of Xie *et al.* is most closely related to this research [20]. Their goal is to generate a suite of unit tests that can expose more faults than an initial test suite. During the execution of the initial suite, feedback in the form of *operational abstractions* (summaries of program run-time state) is used to automatically generate new unit tests that result in *operational violations* (previously unobserved program behaviors) such as method-precondition violations. Other researchers have also used operational abstractions, combined with symbolic execution, to guide the generation of test cases [3].

Boyapati *et al.* also employ a feedback-based technique to obtain all non-isomorphic inputs (test cases) for a method [2]. 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.

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

Several iterative techniques have been used to generate a test case that executes a given program path [6, 7, 9]. The generation is formulated as a function minimization problem. The gradient-decent approach is used to gradually adjust an initial test case so that it executes the given path. Control-flow information in the form of branch-predicate outcomes is collected during software execution.

The *chaining* approach [5] has been used to generate test cases, each to cover a given program statement. An initial test case is executed; the program's control-flow and data-flow are monitored and 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 [13]. 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.

Model-based GUI Testing: Various types of models have been used for GUI test-case generation. The most popular are state-machine models [17, 18]. These models are created *manually*, requiring considerable effort. Moreover, the fault-detection effectiveness of the generated test cases depends largely on the tester's definition of "GUI state" and the technique used to generate the test cases from the model. For example, the tester may want to "cover all the states" or "cover all the state transitions" in the state-machine. Consequently, two testers who test the same GUI application, using different definitions of GUI state and test adequacy criteria, may detect different sets of faults. Hence, these techniques produce results that are *non-repeatable* across communities of testers. Several new techniques that utilize search-based algorithms to generate (and improve) test cases also suffer from the above problems. Recent examples include AI planning [11] and genetic algorithms [8].

In order to minimize manual work and produce repeatable results across multiple testers, several new systematic techniques based on graph models of the GUI have recently been developed. These techniques leverage a standard reverse-engineering technique [12] to automatically create the GUI model. The most successful graph models that have been used for GUI test-case generation include *Event Flow Graphs* (EFG) [10] and *Event Interaction Graphs* (EIG), also discussed in the previous section. The nodes in these graphs represent events; the edges represent relationships between pairs of events. Test cases are systematically generated to satisfy various types of adequacy criteria. One criterion (called the *event-interaction crite-*

rion [12]) requires each edge in an EIG to be covered by at least one test case; test cases are generated by picking the two events on each edge and using a shortest-path algorithm to reach these events from the application's main window. Such techniques are automated and the algorithms always produce the same test suites, making the results repeatable.

The primary problem with these approaches is that the number of event sequences grows exponentially with sequence length. Hence, in previous work, we have been able to generate test cases that cover all edges in the EIG, *i.e.*, they test two-way interactions between GUI events. This paper extends our previous work by systematically generating test cases for multi-way interactions.

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 all events whose event handlers interact in terms of code elements (*e.g.*, share variables, exchange messages, share data) should be tested together. For example, consider the event handlers for the events e_1 and e_2 shown in Figure 1. 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, since the handlers for e_1 and e_3 do not interact, these events need not be tested together.

One may employ a variety of static program-analysis techniques to identify such interactions [16]. 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 [16]. Also, since most GUI applications employ a large number of library elements (*e.g.*, Java Swing), source code 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? Admittedly, several different types of dynamic behaviors may point to interactions between event handlers. It is not our intention to identify all such behaviors; rather, we will demonstrate proof-of-concept by focusing on a few behaviors. Extensions may easily be devised to make the technique more effective.

Informally, event e_x *interacts with* event e_y, if, when executed together in a sequence $< e_x; e_y >$, they produce a GUI state that is, in some sense, *different from* the two states

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 zoom1: zoom2.setBG(pColor); curZoom = zoom2;
    case zoom2: zoom3.setBG(pColor); curZoom = zoom3;... } }}
theCanvas.repaint();... }
```

Figure 1. Example Event Handlers

that would be obtained had e_x and e_y been executed in isolation. Consider the example shown in Figure 2. The top-left shows the *initial state* (S_0) of an application. After an event e_1 (event handler shown in Figure 1) 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) 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 $e_1(S_0)$. According to the intuition presented in the beginning of this paragraph, because the sequence $< e_1; e_2 >$ produces a GUI state that is different from the two states that would be obtained had e_1 and e_2 been executed in isolation, event e_1 interacts with event e_2, and should be tested together to check for interaction problems. The code for e_1 and e_2 shows that they do in fact interact.

The usage of "different from" above is somewhat misleading. It seems to suggest 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 example, events e_x and e_y are non-interacting and need not be tested together. In order to avoid this confusion, we now formalize the notion of interacting events.

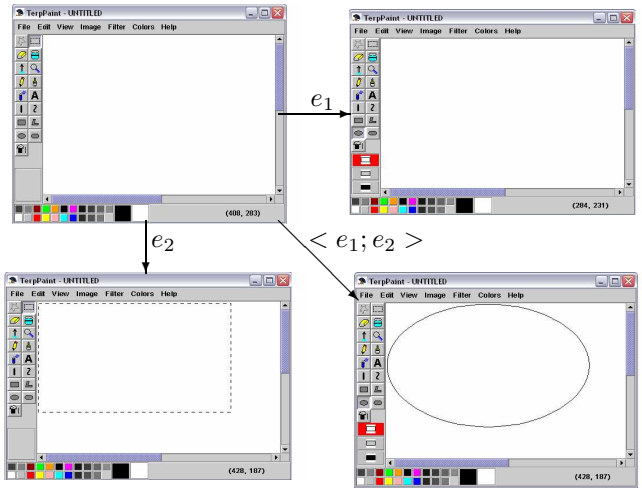


Figure 2. Execution of Events e_1 and e_2

It turns out that the example illustrated in Figure 2 is just one case of how the GUI state may be used to pinpoint interactions between event handlers – there are many more. This research provides a starting point by identifying a total of six cases. Intuitively, these cases will describe (as evaluative predicates) situations in which e_1 and e_2 interact, *i.e.*, the combined effect of e_1 and e_2 is *different from* the effect of the individual events e_1 and e_2. In these six cases, e_1 and e_2 are system-interaction events in modeless windows; this situation will be called *Context 1*.

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

Case 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 $< e_1; e_2 >$.

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

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

The above four cases all handle widgets that persist across the three states being considered, *i.e.*, $e_1(S_0)$, $e_2(S_0)$, 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: $\exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.$ $((v \neq v') \wedge ((w,p,v) \in e_x(S_0)) \wedge ((w,p,v) \notin S_0) \wedge ((w,p,v') \in e_2(e_1(S_0))))$; 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.

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: $\exists w \in W, \texttt{ENABLED} \in P_w, \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}}, s.t.$ $(((w, \texttt{ENABLED}, \texttt{FALSE}) \in S_0) \wedge ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_1(S_0)) \wedge \texttt{EXEC}(e_2, w))$; 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 $\texttt{EXEC}(e_2, w)$.

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 $e_2(e_1(S_0))$ 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.

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.

[3] Notation for "such that"

There is an *Event Semantic Interaction* relationship between two events e_1 and e_2 if and only if at least one of the predicates in Cases 1 through 6 evaluates to `TRUE`; 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 above; m is the context number. If multiple cases apply, then one of the case numbers is used. Due to the specific ordering of the events in the sequence $< e_1; e_2 >$, the ESI relationship is not symmetric.

Once all of the cases have been implemented, the feedback-based process execution is straightforward. The seed suite is executed on the GUI software. The state information is collected and the above predicates are evaluated for each pair of system-interaction events 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). If one of the predicates evaluates to `TRUE`, the two events are ESI-related. When 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 ESIG may be traversed using a modified version of the `GenTestCases` algorithm discussed in Section 2. 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. The feasibility study in the next section uses values 3, 4, and 5 for this parameter.

5 Feasibility Study

The test cases obtained from the `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 research, 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.

Implementation of the above process also 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:

Q1: 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?

Q2: 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?

Q3: How do the ESIG- and EIG-generated test suites compare in terms of fault-detection effectiveness? Do the former detect faults that were not detected by the latter?

To answer these 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 [19]; 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* was executed on all the applications; after the study, we verified that none of our reported bugs were detected by FindBugs.

STEP 2: Generation of EIGs & seed test suites. The EIGs of all subject applications were obtained using reverse engineering. To address **Q1** 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 3 (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

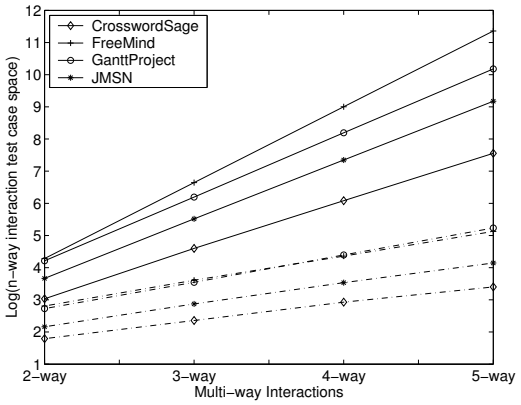


Figure 3. Test Case Space Growth

CrosswordSage, FreeMind, GanttProject, and JMSN, respectively. The GUI's run-time state was recorded during test execution. All faults were fixed in the applications.

STEP 4: Generation of the ESIG. The above feedback was used to obtain the ESIs for each application. To address **Q2**, the number of ESI relationships in which each event participates is shown in Figure 4. 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. In the future, we will create a classification of these dominant events.

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 3. 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.

STEP 5: Execution of the test cases. To address **Q3**, 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. The result shows that the ESIG-based test cases help to detect additional faults.

This study demonstrated that test suites for multi-way GUI event interactions are able to detect additional faults compared to two-way interactions. In earlier work, we have shown that two-way interactions yield high code coverage, while multi-way interactions cover little additional code [10]. The additional fault-detection effectiveness of multi-way interactions is due to the execution of combinations of events in different execution orders. Also, this study did not use the newly-generated test cases in its seed suite to generate additional test cases; the extension will be explored and its benefits demonstrated in future work.

As always, results of studies should be interpreted with threats to validity in mind. Several such threats are identified in this study. First, four Java applications have been used as subject programs. Although they have different types of GUIs, this does not reflect the wide spectrum of possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, *i.e.*, most of the code is written for the GUI. The results will be different for applications that have complex underlying business logic and a fairly simple GUI. Second, all the subject applications are open-source, typically developed by volunteer developers and might be more bug-prone than software implemented by paid developers. Third, the run-time state of GUI widgets is obtained using Java Swing API. These widgets may have additional properties that are not exposed by the API. Hence the set of ESI relationships may be incomplete.

6 Discussion

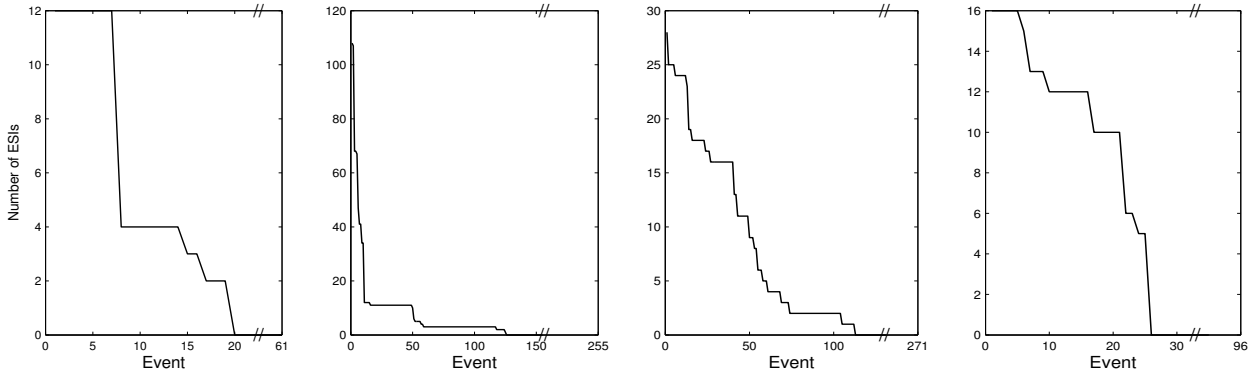
Several lessons were learned from this study. First, the developers of the applications felt that the crashes revealed important faults in the code. Several crashes were reported on each application's bug-reporting site. In response, some of them have already been fixed in subsequent releases of the applications. For example, Bugs #1536224, #1536229, and #1536205 (SourceForge-assigned numbers) have been fixed by the developers of FreeMind.

Second, the study provided evidence that the intuition behind using the GUI's run-time state to find sets of interacting events was useful. Upon closer examination, several test cases that caused crashes had executed events that shared some code elements. The first evidence of the usefulness of the state-based feedback approach was apparent even with the seed test suite.

Bug#1536205 of FreeMind was detected using a test case from the seed suite. It caused a NullPointerException when *reverting* back from a newly created FreeMind map to its previously saved version.[4] The test case contained two system-interaction events: e_1 – *Create a new FreeMind map*, and e_2 – *Revert*. FreeMind starts with a default map; event e_1 creates a new map with one node; event e_2 reverts the map back to the previously saved version. These events are related using Case 2, *i.e.*, $e_1 \xrightarrow{2(1)} e_2$. When executed together, they modify the map object; executed individually, e_2 does not change the state, as there is no saved map.

The crash occurred because the event handlers of e_1 and

[4]FreeMind is a "mind mapping" software.



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

Figure 4. ESI Distribution in OSS

e_2, contained in the `NewMapAction.class` and `RevertAction.class`, respectively, improperly handled the map object's instance variable `file` used to keep track of the physical file corresponding to the map. A new map object, created by e_1 has no associated file; the variable `file` remains `null`. A subsequent revert event invokes `createRevertXmlAction(file)` that in turn invokes `file.getAbsolutePath()`. With `file` being `null`, a NullPointerException is thrown.

This example reinforced our intuition linking the runtime state resulting from the execution of events to interactions among event-handler code. The ESIG test cases were more interesting and provided additional insights. The remainder of this section describes the ESIG test cases in detail and the causes of the crashes they detected.

Crash 1: One test case, which executed two events in a modal window, then the window's termination event, and finally an event in the parent window, caused a NullFormatException in FreeMind by trying to create a new FreeMind map with a non-numeric font size. The test case contained four events: e_1 – *Select default parameters to set*, e_2 – *Set default font size in text field*, e_3 – *Save preference settings*, and e_4 – *Create a new FreeMind map*. Events e_1 and e_2 are system-interaction events in a modal window titled `Settings`; e_3 is a termination event in the same window; and e_4 is a system-interaction event in FreeMind's main window. The relationships between the events are $e_1 \xrightarrow{6(2)} e_2 \xrightarrow{3(3)} e_4$. Event e_1 enables e_2; the font-size cannot be set unless the text-field (for e_2) is enabled by e_1. Setting a non-numeric font size with e_2, then executing the termination event e_3, causes e_4 to try to create a a new map with an invalid font size, resulting in a crash.

The crash occurred because the event handlers for e_1, e_2, e_3, and e_4, contained in `OptionPanel$ChangeTabAction`, `OptionPanel.class`, `OptionPanel.closeWindow()`, and `NewMapAction.class`, respectively, made different assumptions about the default font size, which FreeMind maintains as a string object. Event e_4, performed by clicking on the `New...` menu-item, creates a new map. During map creation, it obtains the preference settings, including the default font size, which it (incorrectly) assumes to be an integer value. An invocation of `Integer.parseInt()` on the string causes the crash.

Lessons Learned: This example demonstrated that event handlers interact in complex ways. Because event handlers may have been developed by different programmers, they may have made incorrect assumptions about the validity of shared objects, leading to integration problems. Moreover, Context 3 that handles interactions among events contained in multiple windows are extremely useful, since achieving the combined effect of events in a modal window requires the execution of the window's termination event.

Crash 2: An event sequence that executed across two *cascading* (one opened by the other) modal windows, followed by the parent modal window's termination event, caused a NullPointerException in GanttProject. The sequence tried to change the type of a file to be imported after it had selected the name of the file from a list. Developers of the software had expected that users would select the type first, and then select the file name. The sequence contained five events: e_1 – *Choose import file type* T_1, e_2 – *Choose import file*, e_3 – *Click* `OK` *to close* `FileChooser` *window*, e_4 – *Choose import file type* T_2, e_5 – *Click* `OK` *to close import file window*. Events e_1 and e_4 are system-interaction events in a modal window titled `Import`; event e_5 is the window's termination event. Event e_2 is a system-interaction event in the `FileChooser` modal window; event e_3 is the window's termination event. Note that another event is used after e_1 to open the `FileChooser` window; it is not shown here because it is not a part of the EIG. The relationships between the events are $e_1 \xrightarrow{4(2)} e_2 \xrightarrow{4(2)} e_4$. The test case selects a file type (e_1) and the file name (e_2), but then changes the type (e_4) without changing the file name. Executing the

termination event results in a crash.

The crash was caused because the event handlers of the events e_1, e_2, e_3, e_4, and e_5 contained in `ImporterChooserPage.class`, `FileChooserPage.class`, `JFileChooser.closeWindow()`, `ImporterChooserPage.class`, and `ImportFileWizardImpl.onOkPressed()`, respectively, failed to keep track of the relationship among a file's name, its type, and the file suffix in the name. GanttProject records the import file's type in `myProject` and its name in `myState`. `ImportFileWizardImpl.onOkPressed()` examines the selected type and assumes that the file name will have the correct suffix. If, however, the filename does not have the assumed suffix, an object `Open` remains `null`. Execution of `Open.load(file)` results in a NullPointerException.

Lessons Learned: This example reinforced our original motivation for developing new techniques for model-based GUI testing. Developers often cannot predict how users will use the software. They typically test the software for a small number of obvious, predictable use-cases. Developers need to use automated techniques to test their software for a larger number of unpredictable event sequences.

Crash 3: One event sequence caused a NullPointerException in FreeMind after executing a `Cancel` event in a modal window. It tried to close a window that was trying to save a file with an empty name. The sequence contained six events: e_1 – *Create a new FreeMind map*, e_2 – *Add a child node on current FreeMind map*, e_3 – *Insert a hyperlink to the child node*, e_4 – *Click* `OK` *to close window*, e_5 – *Click the hyperlink; side effect opens the* `Save` *window*, and e_6 – *Click the* `Cancel` *button to close the* `Save` *window*. e_1 and e_2 are system-interaction events in FreeMind's main window; e_3 is a system-interaction event in a modal window `Hyperlink`; e_4 is `Hyperlink`'s termination event; e_5 is a system-interaction event in the main window with the side effect of opening a modal window `Save`; e_6 is `Save`'s `Cancel` termination event. Note that another event (not shown here) is used to open the `Hyperlink` window. The relationships between the events are $e_1 \xrightarrow{4(1)} e_2 \xrightarrow{4(1)} e_3 \xrightarrow{6(3)} e_5$.

The crash occurred because the event handlers of e_1, e_2, e_3, e_4, e_5, and e_6 contained in `NewMapAction.class`, `NewChildAction.class`, `SetLinkByTextFieldAction.class`, `HyperLinkDialog.closeWindow()`, `NodeMouseMotionListener.mousePressed()`, and `ControllerAdapter$loadURL()`, respectively, failed to set an instance variable `file`, and check its validity before associating it with a hyperlink. The association, caused by `getFile().toURL()`, results in a crash as soon as the termination event is executed.

Lessons Learned: This example demonstrated two points. First, software crashes can occur after the execution of the `Cancel` button; we had not expected this result. Second, this sequence contained 6 events; in our study the `GenTestCases` algorithm generated sequences that had at most 5 events. The only reason that `Cancel` was executed is that `GenTestCases` closes all open windows (using `Cancel`) after a test case has completed execution.

Summary: The above crashes (and the ones not presented here) illustrated several important points. First, the event handlers for events are typically implemented in multiple classes. Static analysis that is limited to intra-class analysis fails to reveal problems with interacting events. Second, with the increasing flexibility of new user interfaces, programmers must take steps to ensure that their software works correctly for a large input space. They should check the validity of objects whenever possible before use; text fields in particular should be restricted to the smallest input domains possible. For example, an e-mail address text field in JMSN caused a crash after a non-e-mail string was entered. The developers had simply checked whether the length of the entered text was non-zero, which is clearly inadequate. Third, most of the test cases that revealed the crashes did not add to the code coverage (statement and branch) of the seed test suite. They were able to detect the faults because of the permutations of events that were executed on the GUI. Finally, we expect that we would not have found many of these faults had we simply performed a random walk of the EIGs. The walk would have used four times as many events, adding a significant space of interactions, and it would have generated sequences of unrelated events. Comparison of our technique to such a "random walk" approach is a topic for future research.

7 Conclusions and Future Work

This paper presented a new 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 as a starting point for feedback collection. Subsequently-generated and- executed test cases are expected to be used for the analysis, iteratively yielding additional test cases. This algorithm can be stopped any time to yield useful results. The technique was demonstrated via a feasibility study on four fielded software applications. The results of the study showed that the test cases generated using the feedback were useful at detecting serious and relevant faults in the applications.

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 ex-

plored. This revision will also lead to the definition of new, fundamentally different cases for the ESI relationship.

The results of the feasibility study 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 beyond crashes 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 predetermined length. As the goal of this work is to generate multi-way interactions among GUI events, other techniques (such as *covering arrays* [21]) designed to minimize the number of test cases while retaining high interaction coverage will be explored.

The analysis summarized in Section 6 led to a deeper understanding of the relationship between GUI events and the underlying code. 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.

Some of the challenges of GUI testing are also relevant to testing of Web applications and object-oriented software. One way to test these classes of software is to generate test cases that are sequences of events (either Web user actions or method calls). Some of the techniques developed in this research may be used to prune the space of all possible interactions that need to be tested.

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.

References

[1] F. Belli. Finite-state testing and analysis of GUIs. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 34–43, 2001.

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

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

[4] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. 1999.

[5] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[6] M. J. Gallagher and V. L. Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Trans. Software Eng.*, 23(8):473–484, 1997.

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

[8] D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In *CHI*, pages 244–251, 1996.

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

[10] A. M. Memon, A. Nagarajan, and Q. Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution*, 17(1):27–64, Jan. 2005.

[11] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Software Eng.*, 27(2):144–155, 2001.

[12] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[13] C. C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng.*, 27(12):1085–1110, 2001.

[14] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[15] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI*, pages 195–202, 1992.

[16] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 14–16, 2004.

[17] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, page 80, 1997.

[18] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, page 110, 2000.

[19] Q. Xie and A. M. Memon. Automated model-based testing of community-driven open source GUI applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 2006.

[20] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Autom. Softw. Eng.*, 13(3):345–371, 2006.

[21] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *Proceedings of the 2004 international symposium on Software testing and analysis*, pages 45–54, 2004.