Alternating GUI Test Generation and Execution

Xun Yuan
Department of Computer Science
University of Maryland,
College Park, MD 20742
xyuan@cs.umd.edu

Atif M. Memon
Department of Computer Science
University of Maryland,
College Park, MD 20742
atif@cs.umd.edu

Abstract

Users of today's software perform tasks by interacting with a graphical user interface (GUI) front-end via sequences of input events. Due to the flexibility offered by most GUIs, the number of event sequences grows exponentially with length. One ubiquitous challenge of GUI testing is to selectively generate those sequences that lead to potentially problematic states. This paper presents ALT, a new technique that generates GUI test cases in batches, by leveraging GUI run-time information from a previously run batch to obtain the next batch. Each successive batch consists of "longer" test cases that expand the state space to be explored, yet prune the "unimportant" states. The "alternating" nature of ALT allows it to enhance the next batch by leveraging certain relationships between GUI events (e.g., one enables the other, one alters the other's execution) that are revealed only at run-time and non-trivial to infer statically. An empirical study on four fielded GUI-based applications demonstrates that ALT is successful at identifying complex failure-causing interactions between GUI events.

1 Introduction

As computers find increasingly more general-consumer oriented applications, the class of software applications that use a graphical-user interface (GUI) front-end [4] is becoming ubiquitous. GUIs are now seen in cars, phones, dishwashers, refrigerators, etc. They are popular because of the flexibility that they offer to both developers and users. They allow a software developer to implement the GUI by coding reusable *event-handlers* (program code that handles or responds to a user input event) that can be developed and maintained fairly independently. Moreover, GUIs give many degrees of freedom to the software user, *i.e.*, the user is not restricted to a fixed ordering of inputs. The user interacts with complex underlying software by performing *events* (*e.g.*, left-click-on-FILE-menu,[1] left-click-on-CANCEL-button) that exercise GUI *widgets*. The software responds by changing its state and/or producing an output, and waits for the next input.

[1]For brevity, whenever possible, we will use the widget label to denote an event, *e.g.*, Cancel, File, etc.

This flexibility creates problems during execution because of the large number of permutations of events that need to be handled by the GUI. In principle, event handlers may be executed in any order; in earlier work, we have shown that certain event interactions lead to serious software failures [12]. Because the *space of all possible interactions* with a GUI is enormous, each event sequence can result in a different state, and the software may, in principle, need to be tested in all of these states.

One of our existing model-based GUI testing solutions, motivated by work on search algorithms for software testing [9], is based on a directed graph model of the GUI called event-interaction graph (EIG) [12]. EIG nodes represent all GUI events except those that open menus and windows; a directed edge from node n_x (representing event e_x) to n_y (representing event e_y) shows that event e_y may be executed either immediately after e_x or after executing some intermediate menu- and/or window-opening events. Test cases are generated, each covering one directed edge (a pair of events) in the EIG. These test cases are referred to as "2-way covering" because each test targets a unique pair of EIG events [12]. Our previous empirical studies showed that although these test cases reveal a large number of GUI faults, additional faults may be detected by executing certain types of multi-way covering (*e.g.*, 3-, 4-, 5-way) test cases [12, 17]. For example, a 3-way covering test case is an event sequence $< e_1; e_2; e_3 >$ such that (e_1, e_2) and (e_2, e_3) are directed edges in the EIG; similarly, a 4-way covering test case $< e_1; e_2; e_3; e_4 >$ covers edges (e_1, e_2), (e_2, e_3), and (e_3, e_4). The challenge, of course, is to generate and execute such "long" test cases; increasing the degree of the event interaction coverage from all possible 2-way to all possible 3-, 4-, . . ., multi-way interactions is not a viable solution as the number of test cases grows exponentially; for most non-trivial applications, executing even all 3-way interactions is not practical.

In previous work, we developed a feedback-based technique to enhance a 2-way covering test suite to a 3-, 4-, and 5-way covering test suite [17]. We did this by analyzing the effect of each GUI event on the GUI's run-time state and obtaining pairs of events that influence one another in how they modify the GUI's state. This "influence" was captured as the *Event Semantic Interaction* (ESI) relation and modeled as a graph called the ESI Graph (ESIG). For most non-trivial applications, the ESIG is much smaller than the EIG, making it possible to generate 3-, 4-, and 5-way covering test cases by enumerating all possible paths of length 3, 4, and 5 in the ESIG. An important property of these test cases is that all adjacent events are related via the ESI relationship. We summarize this technique in Section 2. However, although better than the exhaustive approach, the number of test cases required for the ESIG-based technique also grows exponentially with length for most applications, making it difficult to test 5-way and above interactions. We continue to utilize the most important aspect of this previous approach, namely feedback and its use in the ESI relation.

This paper significantly improves upon the ESIG-based approach by generating test cases "in batches." The first batch consists of all possible 2-way covering test cases, generated automatically using the existing EIG model of the GUI. This batch is executed and the observed execution behavior of the GUI, captured in the form of widgets and their properties, is used to selectively extend some of the 2-way test cases to 3-way test cases via the ESI relation. The new 3-way test cases are subsequently executed, GUI execution behavior is analyzed, and some are extended to 4-way test cases, and so on. In general, the new "alternating approach" (called ALT) executes and analyzes i-way covering tests, identifying sets of events that interact in interesting ways with one another (and hence should be tested together), and generates (i+1)-way covering test cases for members of each set. Hence ALT generates "longer" test cases that expand the state space to be explored, yet pruning the "unimportant" states. A side-effect of the batch-style nature of this new approach is that certain aspects of GUI test cases that are revealed only at run-time and impossible to infer statically, *e.g.*, infeasible test cases, are also used to enhance the next batch. An empirical study on four fielded GUI-based applications shows that ALT allows us to generate longer, more interesting and focused test cases that are effective at detecting faults.

The specific contribution of this paper include:

- Iterative enhancement of GUI test suites.
- Use of feedback to compute run-time relationships between events and better handle infeasible test cases.
- Empirical demonstration that the run-time information is successful at identifying complex interactions among GUI event handlers.

The next section provides background, summarizes our previous work on GUI testing, and explains the ESIG-based approach. Section 3 presents an overview of ALT via an example and Section 4 provides more formal details. Section 5 presents an empirical study to evaluate ALT. Section 6 concludes with a discussion of future work.

2 Background and Related Work

In the context of this work, execution feedback refers to information obtained during test execution and used to guide further 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 [13]. 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. Execution feedback is collected and analyzed; results are used to evaluate the "closeness" 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 according to some criterion.

Since then, several techniques have been based on dynamic test generation. They use feedback from the application'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], operational abstractions to cover increased program behaviors [3, 16], partially generated non-exception-throwing method-call sequences to generate longer sequences [14], and as input to a fitness function to guide genetic search [5, 9].

Our own ESIG-based approach [17] was also motivated by the above research. We introduced the idea of employing *feedback* from the execution of a seed test suite (our 2-way covering test cases generated using the EIG) to generate additional multi-way interaction test cases. The key idea was to analyze run-time GUI state to identify sets of events that need to be tested together in multi-way covering test cases. The result of this analysis is called the *Event Semantic Interaction* (ESI) relation between pairs of events.

We now explain the ESI relationship and its background. A GUI is represented as a set W of *widgets* (*e.g.*, buttons, text fields); each widget $w \in W$ is associated with a set P_w of *properties* (*e.g.*, color, size, font); at any time instant, each property $p \in P_w$ may take 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. Hence, the set of triples (w, p, v), where $w \in W, p \in P_w$ and $v \in V_p$ models the GUI's *state* for a time instant. The set of states S_I at the time when a GUI is first invoked is called the *valid initial state set* for the GUI. The state of a GUI is not static; users interact with the GUI by executing events (e_1, e_2, $\ldots$, e_n); hence events are modeled as functions that transform one GUI state 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.

An important aspect of our feedback-based technique is the seed suite. For this work, the seed suite consists of all 2-way covering test cases. We leverage a directed graph model, called the event-interaction graph (EIG) of the GUI

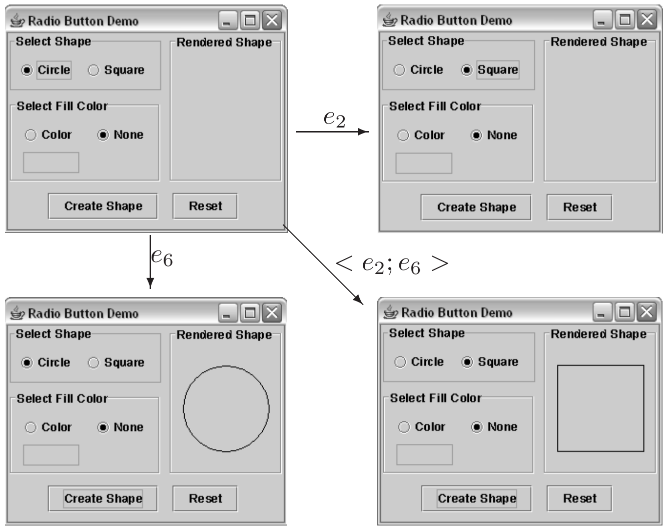

Figure 1. Execution of Events e_2 and e_6

to generate these test cases [12]. An important property of a GUI's EIG is that it can be constructed semi-automatically using a reverse engineering technique called *GUI Ripping* [12]. The *GUI Ripper* automatically traverses a GUI under test and extracts the hierarchical structure of the GUI and events that may be performed on the GUI. The result of this process is the EIG. The 2-way covering test cases are short, each only covering a directed edge in the EIG; extra menu- and window-opening events needed to reach the events in the edge are generated on-demand at test-execution time.

Informally, event e_x and e_y are related via the ESI relation, 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 that would be obtained had e_x and e_y been executed in isolation. Consider the example shown in Figure 1 (more details of this application are given in Section 3). The top-left shows the *initial state* (S_0) of the application. After an event e_2 is executed (click on `Square` radio button), the GUI changes its state to the one shown in the top-right ($e_2(S_0)$). In this state, `Square` is set; `Circle` is reset. Starting from S_0, one can execute another event e_6 (click on `Create Shape` button) and obtain the state shown in the bottom-left ($e_6(S_0)$); a circle is rendered. If, however, the sequence $< e_2; e_6 >$ is executed in S_0, a new state ($e_6(e_2(S_0))$), shown in the bottom-right is obtained; a square has been created. This execution is equivalent to executing event e_6 in the state $e_2(S_0)$. According to the intuition presented at the beginning of this paragraph, because the sequence $< e_2; e_6 >$ produces a GUI state that is different from the two states that would be obtained had e_2 and e_6 been executed in isolation, the two events should be tested together to check for interaction problems.

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. Lets look at the event handlers for e_2 and e_6 in Figure 2; we see that they share variables `created` and `currentShape`; e_6 sets `created` to `true` and influences e_2's flow of control; e_2 sets `currentShape` to a square, which e_6 uses as a parameter to `setShape()`; hence it's not surprising that they interact. One may employ a variety of static program-analysis techniques to identify such interactions [15]; they can certainly be used successfully in this example. However, in general, 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 [15]. 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. Hence, our approach avoids static analysis; instead it approximates the identification of interactions between event handlers by analyzing feedback from the run-time state of the GUI. The remaining question is: *What constitutes event interaction as computed from the GUI's state?*

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 $(e_x(S_0) \neq e_y(e_x(S_0))) \vee (e_y(S_0) \neq e_y(e_x(S_0)))$. 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 formalized the notion of interacting events by developing formal predicates in [17].

The predicate for the above example is written as: $\exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.$[2] $((v \neq v') \wedge ((w,p,v) \notin \{S_0 \cap e_x(S_0)\}) \wedge ((w,p,v) \in e_y(S_0)) \wedge ((w,p,v') \in e_y(e_x(S_0))))$; there is at least one widget w that does not exist in the initial state S_0, it is created by e_y with property p and value v. However, the widget is modified when the sequence $< e_x; e_y >$ is executed, ***i.e.***, the value of w's property p changes from v to v'.

This predicate is evaluated to true for e_2 and e_6 because the rendered shape widget does not exist in the initial state. It is created by event e_6 with `Shape` property set to value `Circle`. However, this property changes to `Square` when $< e_2; e_6 >$ is executed.

It turns out that the example illustrated in Figure 1 is just

[2]Notation for ' 'such that"

```
        RBExample::CircleAction(ActionEvent evt){
☑☐☐☐    currentShape = SHAPE_CIRCLE;
☑☐☐☐    if(created) {
☑☐☐☐        imagePanel.setShape(currentShape);
☑☐☐☐        imagePanel.repaint()}}
```

e_1's Event Handler

```
        RBExample::SquareAction(ActionEvent evt){
☑☑☑☑    currentShape = SHAPE_SQUARE;
☑☑☑☑    if(created) {
☑☑☑☑        imagePanel.setShape(currentShape);
☑☑☑☑        imagePanel.repaint();}}
```

e_2's Event Handler

```
        RBExample::ColorAction(ActionEvent evt){
☑☑☑☑    colorText.setEditable(true);
☑☑☑☑    currentColor = getColor();
☑☑☑☑    if (created) {
☑☐☐☐        imagePanel.setFillColor(currentColor);
☑☐☐☐        imagePanel.repaint();}}
```

e_3's Event Handler

```
        RBExample::NoneAction(ActionEvent evt){
☑☐☐☐    colorText.setEditable(false);
☑☐☐☐    currentColor = COLOR_NONE;
☑☐☐☐    if (created) {
☑☐☐☐        imagePanel.setFillColor(currentColor);
☑☐☐☐        imagePanel.repaint();}}
```

e_4's Event Handler

```
        RBExample::CreateAction(ActionEvent evt) {
☑☑☑☑    if (color.isSelected()) {
☑☑☑☑        currentColor = getColor();}
☑☑☑☑    imagePanel.setFillColor(currentColor);
☑☑☑☑    imagePanel.setShape(currentShape);
☑☑☑☑    imagePanel.repaint();
☑☑☑☑    created = true;}
```

e_6's Event Handler

```
        RBExample::ResetAction(ActionEvent evt){
☑☐☐☐    square.setSelected(true);
☑☐☐☐    none.setSelected(true);
☑☐☐☐    colorText.setText("black");
☑☐☐☐    colorText.setEditable(false);
☑☐☐☐    currentShape = SHAPE_NONE;
☑☐☐☐    imagePanel.setShape(currentShape);
☑☐☐☐    currentColor = COLOR_NONE;
☑☐☐☐    imagePanel.setFillColor(currentColor);
☑☐☐☐    imagePanel.repaint();}
```

e_7's Event Handler

```
        ImagePanel::paintComponent(Graphics g) {
☑☑☑☑    clear(g);
☑☑☑☑    Graphics2D g2d = (Graphics2D)g;
☑☑☑☑    if (currentShape == SHAPE_CIRCLE) {
☑☑☑☐        if (currentColor == COLOR_NONE) {
☑☑☐☐            g2d.setPaint(Color.black);
☑☑☐☐            g2d.draw(circle);}
☑☑☑☑        else {
☑☑☑☑            g2d.setPaint(currentColor);
☑☑☑☑            g2d.fill(circle);}}
☑☑☑☑    else if (currentShape == SHAPE_SQUARE) {
☑☑☑☐            if (currentColor == COLOR_NONE) {
☑☑☐☐                g2d.setPaint(Color.black);
☑☑☐☐                g2d.draw(square);}
☐☐☑☑            else {
☐☐☑☑                g2d.setPaint(currentColor);
☐☐☑☑                g2d.fill(square);}}}
        ImagePanel:setFillColor(int inputColor) {
☑☑☑☑    switch(inputColor) {
☐☑☑☑    case COLOR_BLACK:
☐☑☑☑        currentColor=Color.black;
☐☑☑☑        break;
☐☑☑☑    case COLOR_RED:
☐☑☑☑        currentColor=Color.red;
☐☑☑☑        break;
☐☑☑☑    case COLOR_GREEN:
☐☑☑☑        currentColor=Color.green;
☐☑☑☑        break;
☑☑☑☑    default:
☑☑☑☑        currentColor=Color.gray;}}
```

The ImagePanel Class

Figure 2. Some Source Code for the Radio Button GUI Example.

one case of how the GUI state may be used to pinpoint interactions between event handlers; the work in [17] presents six cases, each in three different *contexts* of GUI structure (we now have a total of 16 predicates; for space reasons, we show only three in this and the next section). These cases describe (as evaluative predicates) situations in which e_x and e_y interact, *i.e.*, the combined effect of e_x and e_y is *different from* the effect of the individual events e_x and e_y.

There is an *Event Semantic Interaction* relationship between two events e_x and e_y if and only if at least one of the predicates is TRUE; this relationship is written as $e_x \xrightarrow{n(m)} e_y$, where the number n is one of the predicate numbers; m is the context number. If multiple predicates apply, then one of the numbers is used. Due to the specific ordering of the events in the sequence $< e_x; e_y >$, the ESI relationship is not symmetric. For space reasons, we will omit the discussion of context in this paper; the interested reader is referred to our earlier work [17]; we do provide context numbers in our reported ESI relationships for the sake of completeness.

3 Overview of ALT

We now present an overview of ALT via a simple application shown in Figure 3.[3] The GUI contains seven widgets labeled w_1 through w_7 on which a user can perform corresponding events e_1 through e_7. The application's functionality is very straightforward – the *start state* has Circle

[3]This unaltered example is used to teach students how to use radio buttons.

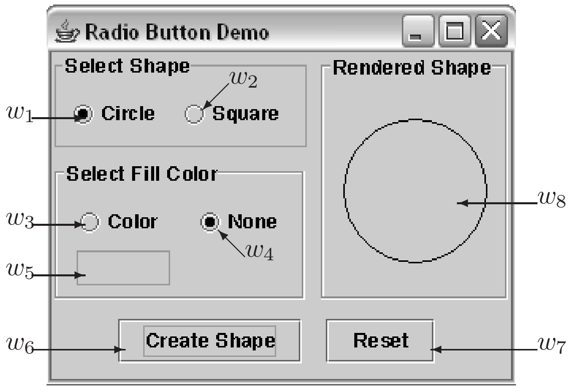

Figure 3. A Simple GUI Application

and None selected; the text-box corresponding to w_5 is empty; and the Rendered Shape area (widget w_8) is empty. 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 start state.

The other events behave as follows. 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-box 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 initial state.

The GUI of this application is simple, yet quite flexible. The number of 1-, 2-, 3-, 4-, and 5-way event sequences (and hence possible test cases) that may be executed in the start state of the GUI is 6 (remember that e_5 is initially disabled), 37, 230, 1491, and 9641, respectively. This is clearly too large a number to test on such a small GUI.

We now present the steps of ALT:

(1) *Obtain the event-interaction graph (EIG).* As mentioned in Section 2, this is done via automated reverse engineering techniques [11]. Because of current limitations of the reverse engineering process, it is unable to automatically infer the (enable) relationship between e_3 and e_5; hence the EIG is a fully connected directed graph with seven nodes, corresponding to the seven events.

(2) *Generate and execute the 2-way covering test suite.* This suite consists of all 2-way covering event sequences, which are obtained by simply enumerating the edges of the EIG. Each of these sequences is executed in the software's start state. As expected, none of the sequences starting with e_5 executed. However, the sequence $< e_3; e_5 >$ executed successfully, indicating that e_3 enables e_5.

Also, the entire state of the GUI is captured after each event for each test case. This includes all the properties of all the GUI's widgets. However, we will restrict our discussion to the state of interest for this example, which includes the state of each radio button, *i.e.*, selected/not-selected and the contents of `Rendered Shape` area. This part of the state will be used to compute the ESI relationships.

(3) *Compute ESI relationships.* Our ESI relationships between two events are based on the ability of an event to influence another event's execution, as captured in the GUI's state. We saw in the previous section that e_2 influences e_6. Event e_6 alone from the start state renders a circle in the `Rendered Shape` area. However, executing e_2 before e_6 changes the behavior of e_6, yielding a square instead. This "interaction" is captured by our ESI predicate number 9 and represented as $e_2 \xrightarrow{9(1)} e_6$.

Another interesting relation in this example is $e_6 \xrightarrow{8(1)} e_2$, *i.e.*, e_6 is ESI related to e_2. In the default start state, e_6 creates a circle. However, the sequence $< e_6; e_2 >$ yields a square because e_2 changes the shape. The predicate (number 8 in our set) used to compute this relation is $\exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.((v \neq v') \wedge ((w,p,v) \notin \{S_0 \cap e_y(S_0)\}) \wedge ((w,p,v) \in e_x(S_0)) \wedge ((w,p,v') \in e_y(e_x(S_0))))$; there is at least one widget w which does not exist in the initial state S_0; it is created by e_x with property p and value v. However, it is modified when the sequence $< e_x; e_y >$ is executed, *i.e.*, the value of w's property p changes from v to v'. This interaction is due to the `created` variable shared between the code of e_6 and e_2.

Another ESI relationship is $e_3 \xrightarrow{6(1)} e_5$. The predicate used to obtain it is number 6 in our set: $\exists w \in W, \texttt{ENABLED} \in P_w, \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}}, s.t. \quad (((w, \texttt{ENABLED}, \texttt{FALSE}) \in S_0) \wedge ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_x(S_0)) \wedge \texttt{EXEC}(e_y, w))$; there exists at least one widget w that was disabled in S_0 but enabled by e_x. Event e_y is performed on w, represented by a predicate $\texttt{EXEC}(e_y, w)$. This predicate applies because widget w_5 is disabled in the start state but enabled by e_3. In e_3's code this is done by `colorText.setEditable(true)`.

The three relations found in this step are: $e_2 \xrightarrow{9(1)} e_6$, $e_6 \xrightarrow{8(1)} e_2$, and $e_3 \xrightarrow{6(1)} e_5$. The first two are used to extend two of the 2-way covering test cases $< e_2; e_6 >$ and $< e_6; e_2 >$ to $< e_2; e_6; e_2 >$ and $< e_6; e_2; e_6 >$, respectively.

The third relation is used to augment all the 2-way covering test cases that started with e_5 but remained unexecutable earlier. Because e_3 enables e_5, the new 3-way test cases are obtained by prefixing $< e_3 >$ to *all* the 2-way covering test cases that start with e_5, thereby yielding: $< e_3; e_5; e_1 >$, $< e_3; e_5; e_2 >$, $< e_3; e_5; e_3 >$, $< e_3; e_5; e_4 >$, $< e_3; e_5; e_5 >$, $< e_3; e_5; e_6 >$, and $< e_3; e_5; e_7 >$. These test cases will give us an opportunity to observe the effect of e_5, previously unexecuted, on other all events that can potentially follow e_5.

(4) *Execute the new 3-way test cases, obtain new ESI relations, and generate 4-way test cases.* All the GUI states after each event are recorded. This step computes new ESI relations by splitting each 3-way covering test case $< e_x; e_y; e_z >$ into two parts: $< e_x; e_y >$ and e_z; the former is conceptually treated as a single *macro event* E_X and used as input to our existing predicates; the resulting ESI relation is now between E_X (which is really the event sequence $< e_x; e_y >$) and event e_z. We have designed the "splitting" of the test case in the above fashion very carefully so that the E_X part would already have been executed in the earlier batch, thereby requiring no more execution to obtain the new ESI relations.

Consider the event sequence $< e_3; e_5; e_6 >$. This is rewritten as $< E_X; e_6 >$, with E_X being $< e_3; e_5 >$; the semantics of E_X can be imagined as "enter a custom color in an *enabled* text-field w_5"; the ESI predicates are applied. We see that E_X influences e_6. Event e_6 alone from the start state renders an empty circle in the `Rendered Shape` area. However, executing E_X before e_6 changes its behavior, yielding a filled circle instead. Hence, predicate 9 applies; $< e_3; e_5 > \xrightarrow{9(1)} e_6$.

Because $< e_3; e_5 > \xrightarrow{9(1)} e_6$ and $e_6 \xrightarrow{8(1)} e_2$ (as computed earlier), we extend $< e_3; e_5; e_6 >$ to the 4-way test case $< e_3; e_5; e_6; e_2 >$.

None of the other 3-way test cases are extended because the predicates do not apply.

(5) *Execute the new 4-way test cases, obtain new ESI relations, and generate 5-way test cases.* The sole 4-way test case $< e_3; e_5; e_6; e_2 >$ is rewritten as $< E_X; e_2 >$; hence the semantics of E_X are now "enter a custom fill color and create the shape." Note that due to the nature of the splitting,

E_X has already been executed earlier; hence its resulting state is already available for analysis.

We determine that $< e_3; e_5; e_6 > \xrightarrow{8(1)} e_2$. And we already know that $e_2 \xrightarrow{9(1)} e_6$. Hence, only one 5-way covering test case is generated $< e_3; e_5; e_6; e_2; e_6 >$.

(6) *Execute the new 5-way test case, obtain new ESI relations, and generate 6-way covering test cases.* We do not find any new ESI relations; hence ALT terminates.

In all, 37 two-way, 9 three-way, 1 four-way, and 1 five-way test cases were generated in this example. The total number, 48, is much smaller than the *all possible sequences* numbers presented earlier.

We now informally examine how our 48 test cases executed the code of the simple application. Figure 2 shows the event-handler code as well as some helper methods. The statement coverage is summarized as a vector of 4 checkboxes ☑☑☑☑ associated with each statement. The first box is checked if any of the 2-way test cases executed the corresponding line of code; similarly, the second box is for 3-way test cases; third for 4-way, and fourth for 5-way test cases. For example, in the `ImagePanel` class code, lines 16 and 17 were executed only by 4- and 5-way test cases.

There are several points to note about the code and statement coverage. First, each event has a programmer-defined event handler (w_5, which requires no custom functionality, is the exception). Second, the code is implemented in two classes `RBExample` and `ImagePanel` – any code-based analysis must account for interactions across classes. In Section 5, we will see several failures are due to incorrect interactions across classes. Third, event handlers interact either directly or indirectly by using shared variables (*e.g.*, `currentShape`, `created`, `currentColor`) or via method calls (*e.g.*, `setFillColor()`). Detecting such interactions at the code level, especially across classes, is non-trivial. Fourth, while many statements are covered by all types of test cases (*e.g.*, Lines 2-4 in the `ImagePanel` class are executed by 2-, 3-, 4-, and 5-way test cases), a few statements that are guarded by a series of conditional statements are executed by very few test cases (*e.g.*, Lines 16 and 17 in the `ImagePanel` class are executed by the sole 4-way and 5-way test case but was missed by the other 46 test cases.) Finally, although not evident by statement coverage, the 4- and 5-way test cases are able to exercise several combinations of control-flow that are only partially covered by the 2- and 3-way tests.

The above discussion of code coverage is in no way meant to be a formal analysis of the code-covering ability of the ALT test cases. However, it helps to highlight some important aspects of GUI testing that will be investigated in future research.

4 The ALT Algorithm

Having presented an overview of ALT, we now formalize its steps by presenting an algorithm. Intuitively, the algorithm takes an i-way covering test suite as input, in which each test case is fully executable, splits each of its i-way covering test cases $< e_1; e_2; \ldots; e_i >$ into two parts: (1) a macro event $E_X = < e_1; e_2; \ldots; e_{i-1} >$ and (2) the last event e_i. If E_X and e_i are related via an ESI relationship, then for each event e_x that e_i is ESI related to, a new (i+1)-way covering test case $< e_1; e_2; \ldots; e_i; e_x >$ is added to the suite. An extra step handles previously unexecuted events. This approach preserves the property of our earlier ESIG-based test cases that each pair of adjacent events are related via an ESI relation. It imposes a stronger condition that each preceeding sequence starting from the first event is also ESI-related to its subsequent event. Moreover, the alternating approach allows us to detect new ESI relations between newly generated sequences and newly enabled events.

We will assume the availability of several helper functions: (1) $FindState(S_0, E_i)$ that returns the state of the GUI after event sequence E_i has been executed on it, starting in state S_0, (2) *isRelated*(S_0, S_1, S_2, S_3) that returns `TRUE` if at least one of the ESI predicates evaluates to `TRUE`, (3) $pairESI(e_i)$ that returns the set of all events that are ESI-related to e_i, (4) $pairEIG(e_i)$ that returns the set of all events that have an incoming edge from e_i in the EIG, (5) $Last(tc)$ that returns the last event in test case tc, (6) $SubSequence(tc, first, last)$ returns a subsequence of tc starting at $first$ and ending at $last$, (7) *Length*(tc) returns the number of events in tc, and (8) *Union*(T_i, tc) adds tc to T_i. Also, an array `wasNeverExecuted`, indexed by each event, is set to `TRUE` if the event was disabled in the GUI's start state S_0; otherwise it is set to `FALSE`.

The algorithm is shown in Figure 4. It takes the i-way test suite (T_i) as input and returns the (i+1)-way test suite. Each test case is broken into two parts (lines 3–4). If the first "*Length*$(testcase) - 1$" events (E_X) of the test case yield a state that is related via the ESI relationship (determined by the $isRelated$ predicate), to its last event (e_j) (Line 8), then this test case is a good candidate for extension by a new event with all events to which it is ESI related (Lines 9–11). If the last event (e_j) has never been executed before but is made executable by E_X, then it is re-executed to compute new ESI relations (Lines 12–15). The output is the new i+1-way covering test suite.

The algorithm is invoked for T_2, which is obtained from the EIG. Each subsequent invocation with an i-way covering test suite (T_i) as input will yield the (i+1)-way covering suite (T_{i+1}). Testing can be stopped once the testing goals have been met (or the testing team runs out of resources) or ALT returns an empty test suite. This can be if BOTH of the following happen:

```
PROCEDURE::ALT($T_i$){
//$T_i$ is the $i$-way covering test suite.
//$T_{i+1}$ is the output $(i+1)$-way covering test suite.
  $S_0$= GUI's Initial state; $T_{i+1} = \phi$;                          1
  foreach test case $tc \in T_i$ do                                      2
    $E_X$ = SubSequence($tc$, 1, Length($tc$)-1);                        3
    $e_j$ = Last($tc$);                                                  4
    $S_1$ = FindState($S_0$, $E_X$);                                     5
    $S_2$ = FindState($S_0$, $< e_j >$);                                 6
    $S_3$ = FindState($S_0$, $tc$);                                      7
    if isRelated($S_0$, $S_1$, $S_2$, $S_3$)                             8
      foreach $e_x \in pairESI(e_j)$ do                                  9
        $newtc$ = $< E_X; e_j; e_x >$;                                   10
        $T_{i+1}$ = Union($T_{i+1}$, $newtc$);                           11
    if wasNeverExecuted[$e_j$]                                           12
      foreach $e_x \in pairEIG(e_j)$ do                                  13
        $newtc$ = $< E_X; e_j; e_x >$;                                   14
        $T_{i+1}$ = Union($T_{i+1}$, $newtc$);                           15
    wasNeverExecuted[$e_j$] = FALSE                                      16
  return $T_{i+1}$;                                                      17
}
```

Figure 4. The ALT Algorithm

1. no new ESI relations are found (*i.e.*, *isRelated*(S_0, S_1, S_2, S_3) returns FALSE on Line 8) or e_j is not ESI-related to any other event (*i.e.*, $pairESI(e_j)$ returns an empty set in Line 9).
2. e_j has already been executed in an earlier batch or was enabled in S_0.

We observe that this algorithm is fairly conservative in the number of test cases that it generates. Lines 8-9 provide a strict condition to test-case extension, *i.e.*, not only must E_X by ESI-related to e_j, event e_j must also be ESI-related to at least one or more events, *i.e.*, $pairESI(e_j)$ returns a non-empty set. Moreover, we have observed in our experiments that most events have been executed by the second iteration of the algorithm; hence, Lines 12–15 are rarely executed beyond T_3. Because ALT is intended to be one of many algorithms that a tester should have in the "testing tool-box," we feel that having fewer test cases from ALT would help a test designer to conserve resources that may be redirected to other testing techniques, thereby yielding a "healthy" mix of test cases from several techniques.

One final point to note is our use of the function $FindState(S_0, E_i)$. This function maintains a lookup-table to return its output; the table is populated during test-case execution; it is important that all entries exist. Entries corresponding to the three invocations of this function on Lines 5–7 are guaranteed to exist – for the invocation on Line 5, E_X was executed in a previous batch, for Line 6, e_j is a single event, whose resulting state was stored during the execution of the 2-way test cases, for Line 7, tc was executed in the current batch.

5 Empirical Study

The test cases obtained from the ALT algorithm can be generated and executed automatically on the GUI. The only unavailable part is the *test oracle*, a mechanism that determines whether an application under test executed correctly for a test case. In this research, an application 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, ESI, and 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 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 ALT test cases and compare them to 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 does ALT generate? How does this number compare to the EIG- and ESIG-based approaches?
Q2: How many faults are detected by ALT? Of the faults detected in this study, which are detected by ALT and which by the ESIG-based approach? Why does one approach detect a particular fault whereas the other one misses it?

This study was conducted using four 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. More specifically, the following process was used for this study:

1. Choose software subjects with GUI front-ends.
2. Generate and execute the 2-way covering test suite. Obtain the ESI relationships.
3. Generate new test suite using the algorithm of Figure 4.
4. If the newly proposed test suite is empty then stop; else execute it and report crashes.
5. Repeat the last two steps until ALT returns an empty test suite.

To allow comparison, the ESIGs and corresponding test cases were also obtained for all applications.

Subject Application	i-way Suites				
	2	3	4	5	6
FreeMind	614	204	86	3	-
GanttProject	710	617	109	63	1
jEdit	591	419	54	38	-
OmegaT	469	310	11	-	-

Table 1. ESI relationships

	i-way Suites				
	3	4	5	6	7
FreeMind					
EIG	$1.72e8$	$9.56e10$	$5.31e13$	$2.95e16$	$1.64e19$
ESIG	10208	(122426)	(1690861)	(21857767)	(353090927)
ALT	10208	2821	11	2	-
GanttProject					
EIG	$4.94e6$	$7.17e9$	$2.09e12$	$6.07e14$	$1.77e17$
ESIG	3070	14742	27933	(63994)	(125362)
ALT	3070	2229	226	34	4
jEdit					
EIG	$9.17e7$	$4.14e10$	$1.87e13$	$8.42e15$	$3.80e18$
ESIG	7572	84488	(1024424)	(10225602)	(105931205)
ALT	7572	1258	738	171	-
OmegaT					
EIG	$7.65e6$	$1.51e9$	$2.97e11$	$5.85e13$	$1.15e16$
ESIG	2335	8935	42859	(219415)	(1135743)
ALT	2335	1440	-	-	-

Table 2. Test Cases Generation

Subject Application	Technique	i-way test suite		
		3	4	5
FreeMind	ESIG	☑☑	-	-
	ALT	☑☑	-	-
GanttProject	ESIG	☑☑☑☐	-	☐
	ALT	☑☑☑☑	-	☑
jEdit	ESIG	☑☑	☐	-
	ALT	☑☑	☑	-
OmegaT	ESIG	-	-	-
	ALT	-	-	-

Table 3. Fault Detection

STEP 1: Selection of subject applications. Four popular GUI-based OSS (FreeMind 0.8.0, GanttProject 2.0.1, jEdit 4.2, OmegaT 1.7.3) were downloaded from SourceForge. FreeMind and GanttProject have been used in our previous experiments [10]; details of why they were chosen have been presented therein; we added jEdit and OmegaT to reduce threats to external validity. 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* [7] 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; execution of seed suite; computation of ESI relations: The EIGs of all subject applications were obtained using reverse engineering. The seed suite was generated and executed without any human intervention. The GUI's run-time state was recorded during test execution. All faults were fixed in the applications. The feedback was used to obtain the ESIs for each application.

STEP 3: Execution of ALT algorithm: The initial set of ESI relations was used to obtain the 3-way test cases. The number of test cases is shown in Table 2. These test cases were executed and the algorithm was invoked again. This process continued until ALT returned an empty test suite. Table 1 shows the number of ESI relations obtained *from* each of the i-way suites, for $i = 2 \ldots 6$. For example, only one ESI relation was obtained from the 6-way suite of GanttProject. A "-" indicates that we did not have an entry. As the numbers show, the ESI relations decrease with each iteration, thereby helping to terminate the ALT algorithm. This differed across applications: we went as high as 7-way covering test cases for GanttProject and 4-way covering test cases for OmegaT. From these results, we see that the total number of EIG-generated test cases is simply too large (so large that we had to represent them using the "exponent" notation to fit in the table). The 3-way ESIG-generated test suites are manageable; 4-way and beyond becomes quite large. The parenthesized ESIG entries are shown for comparison only – we could not execute such large numbers of test cases; the others were generated and executed. On the other hand, the ALT approach generates a reasonable number of test cases that goes down with each test suite iteration. This helps to answer **Q1**.

Both ALT and the ESIG-approach were successful at detecting faults in the applications, except OmegaT (only 2 faults were detected by the 2-way covering test cases for this application). We show these results in Table 3. Each detected fault is shown as a check-box ☑, which is checked if the fault was detected; otherwise it is unchecked. A "-" indicates that no fault is detected. To allow easy comparison, we show the check-box vector (for the same faults in the same order) for both ALT and ESIG. For example, faults 1, 2, and 3 in GanntProject were detected by both ESIG and ALT. Faults 4 and 5 were not detected by ESIG; they were however detected by ALT, fault 4 by the 3-way test suite and fault 5 by a 5-way covering suite. We see that ALT detected all the faults that ESIG detected and some more using much fewer test cases. This helps to partly answer **Q2**.

We now provide more details of faults 4 and 5 of

GanttProject, and Fault 3 of jEdit. These faults were not detected by ESIG because several events required a complex chain of enabling events, which could only be detected by alternating between test execution and generation.

Fault 4 in GanttProject results in a NumberFormatException. It is detected by a 3-way test case <e_1: *Create new task*; e_2: *Set general task property*; e_3: *Set non-integer value in task duration*>. Event e_3 causes GanttProject to crash because it expects an integer to be entered for the duration text-field in the task property window. However, if a non-integer value is set, GanttProject redraws the task shown in its schedule panel; the method getLength() invokes Integer.parseInt(durationField1.getText().trim()) which throws a NumberFormatException.

In the GUI, event e_1 enables e_2, and the sequence $< e_1; e_2 >$ enables e_3. During ALT test-case generation, none of the 2-way test cases that started with e_2 and e_3 executed; however, the test case $< e_1; e_2 >$ executed, indicating that e_1 enables e_2. Lines 12–14 of the algorithm used this information to extend all 2-way covering test cases that contained e_2 by prefixing e_1 to them; one important test case was $< e_1; e_2; e_3 >$.

In the first iteration of ALT, all 2-way covering test cases that started with e_3 remained unexecuted. Moreover, $< e_2; e_3 >$ was also unexecuted. Hence, by this iteration, ALT did not know how to execute e_3. In the second iteration, once the above-generated 3-way covering test case $< e_1; e_2; e_3 >$ was executed, it was used to determine that $< e_1; e_2 >$ enables e_3. Lines 12–14 used this information to obtain new test cases for the third iteration.

The above 3-way test case was the shortest and only sequence needed to reveal this fault starting in state S_0; none of the 2-, and other 3-way test case could have detected it.

Fault 5 in GanttProject results in a NullPointerException. It is detected by a 5-way covering test case <e_1: *Create new task*; e_4: *Custom columns*; e_5: *Add columns (with a name)*; e_6: *Select newly created column in column table*; e_7: *Delete column*>. Once again, the enabling relationship is complex – e_1 enables e_5, $< e_1; e_4 >$ enabled e_5, $< e_1; e_4; e_5 >$ enables e_6 and e_7. We note that it cannot be detected by any other 5-way or lower test case.

Fault 3 in jEdit results in a NullPointerException. It is detected by the 4-way covering test case < e_1: *Download QuickNotepad plugin*; e_2: *Select QuickNotepad plugin*; e_3: *Install QuickNotepad plugin*; e_4: *Choose QuickNotePad file*>. After installing the QuickNotepad plugin, jEdit allows the user to open a file by entering its path in a text-field. The user is free to enter any string in this text-field, including an incorrect path or the name of a non-existing file. Hence, when opening a non-existing file in QuickNodePad (e_4), the NullPointerException is thrown. In this test case, e_1 enables e_2, $< e_1; e_2 >$ enabled e_3; hence the $< e_1; e_2; e_3 >$ part of the test case was generated by Lines 12–14 of the ALT algorithm. Finally, $< e_1, e_2, e_3 > \xrightarrow{8(2)} e_4$; Lines 8–11 of the ALT algorithm add the event e_4. In this example, we see that the combination of the *enabling* and ESI parts of ALT was important to obtain the test case.

Summary: This study demonstrated that ALT tests are able to detect all the ESIG-detected faults, as well as some additional faults, using fewer test cases. Among the three faults that we discussed, we note that the test cases that detected them were the shortest sequences needed to reveal the faults. Moreover, the ESIG-based approach could not detect them because of its inability to handle disabled events. An alternative algorithm, based on a random walk of the EIG, would have a very low probability of generating the fault-revealing test cases. For example, $\frac{1}{4.94e6}$ probability for Fault 4 of GanttProject. (Recall that the total number of 3-way sequences from the EIG is 4.94e6 for GanttProject.)

The event handlers in the fault-revealing test cases were distributed across multiple classes. For example, for GanttProject, e_1 was in the `NewTaskAction` class; $\{e_2, e_3, e_4\}$ were in `GanttDialogProperties`; $\{e_5, e_6, e_7\}$ were in `GanttTreeTable`. Similarly, for jEdit, e_1 was in the `PluginManager` class, $\{e_2, e_3\}$ in `PluginList`, and e_4 in `BeanShell`. As mentioned earlier, interactions across classes are difficult to infer statically; our run-time state based techniques are agnostic to how the event handlers are distributed.

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 Conclusions and Future Work

This paper presented a new alternating technique to generate n-way covering test cases. It is based on analysis of the run-time state of GUI widgets obtained from a previous test batch to obtain a new batch; the process cycles through test-case generation, execution, and analysis. Our existing 2-way covering test cases are used as a starting point for GUI state collection. Subsequently-generated and- executed test cases are used for the analysis, iteratively yielding additional test cases; no extra test cases are needed. The technique was demonstrated via an empirical study on four fielded software applications. The results of the study showed that the test cases generated using the GUI state were useful at detecting serious faults in the applications; the alternating nature of the technique helped to detect complex enabling relationships between events.

The results of the empirical study afforded two high-priority tasks for future research. First, as discussed in Section 3, we need to understand the subtle nature of the ESI relationship that helps to improve the reachability of critical fault-revealing code. We hypothesize that this improvement is caused by the linking of events that, in some sense, are functionally related; executing them together causes the revelation of problems due to shared objects. Second, several events are ESI-related because of multiple predicates. We currently do not “count” the predicates per relation; in the future, we will explore assigning “strengths” to ESI relations based on how many predicates are `TRUE` for each pair of events.

Some of our earlier work based on the ESIs has been extended to testing Ajax-based web applications [1]. The Document Object Model (DOM) of the page manipulated by the Ajax code is abstracted into a state model. Test cases are derived from the state model based on the notion of semantically interacting events. We expect that our alternating approach will also be applicable in the Ajax domain.

Most faults that we continue to find in our work on GUI testing are triggered only when certain interactions between event handlers occur, *e.g.*, one event handler passes incorrect data to another. As observed by Marsi *et al.* [8], these interactions may also be modeled by information flows, program dependences, and program slices. We will explore the use of these models in our work. From a GUI development point of view, 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. We will also explore the application of a *checking sequence* for GUI testing; a checking sequence is a test sequence that, under certain conditions, is guaranteed to lead to a failure [6]. Although traditionally used for finite-state machines, we feel that it may be extended to our flow-graphs for GUI.

Acknowledgments

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

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA ’02*, 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] M. B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *ESEC/FSE ’97*, pages 244–261, 1997.

[5] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Comput. Oper. Res.*, 35(10):3125–3143, 2008.

[6] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, 2006.

[7] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[8] W. Masri, A. Podgurski, and D. Leon. An empirical study of test case filtering techniques based on exercising information flows. *IEEE Trans. on Soft. Eng.*, 33(7):454–477, 2007.

[9] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *ISSTA ’06*, pages 13–24, 2006.

[10] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. on Softw. Eng. and Method.*, 2008.

[11] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

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

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE ’07*, pages 396–405, May 23–25, 2007.

[15] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Workshop on Program analysis for software tools and eng.*, pages 14–16, 2004.

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

[17] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE ’07*, pages 396–405, May 23–25, 2007.