Information and Software Technology 52 (2010) 559–575

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# Iterative execution-feedback model-directed GUI testing

Xun Yuan, Atif M. Memon *

Department of Computer Science, University of Maryland, College Park, MD 20742, United States

ARTICLE INFO

*Article history:*
Available online 5 December 2009

*Keywords:*
Test-case generation
Model-based testing
GUI testing
Event-driven software
Event-flow graphs

ABSTRACT

Current fully automatic model-based test-case generation techniques for GUIs employ a static model. Therefore they are unable to leverage certain state-based relationships between GUI events (e.g., one enables the other, one alters the other's execution) that are revealed at run-time and non-trivial to infer statically. We present ALT – a new technique to generate GUI test cases in batches. Because of its "alternating" nature, ALT enhances the next batch by using GUI run-time information from the current batch. An empirical study on four fielded GUI-based applications demonstrated that ALT was able to detect new 4- and 5-way GUI interaction faults; in contrast, previous techniques, due to their requirement of too many test cases, were unable to even test 4- and 5-way GUI interactions.

© 2009 Elsevier B.V. All rights reserved.

## 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 [3] 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 or producing an output, and waits for the next input.

Consider the simple application shown in Fig. 1.[2] 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` 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). Event $e_4$ reverts back to the default color. A summary of the behavior of these events is shown in Table 1.

The GUI of this application is simple, yet quite flexible. The number of length 1, 2, 3, 4, and 5 event sequences that may be executed in the start state of the GUI is 6 (remember that $w_5$ is initially disabled), 37, 230, 1491, and 9641, respectively; quite large for such a simple GUI. In general, the flexibility offered by GUIs 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 [20]. 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 fully automatic model-based GUI testing solutions, motivated by work on search algorithms for test data generation [14,15], is based on a *static* directed graph model of the GUI called event-interaction graph (EIG) [20]. 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- or window-opening events.

Because we want our GUI testing to be fully automatic, the most important property of a GUI's EIG is that it can be constructed automatically using a reverse engineering technique called *GUI Ripping* [20]. The *GUI Ripper* automatically traverses a GUI under

* Corresponding author.
*E-mail addresses:* xyuan@cs.umd.edu (X. Yuan), atif@cs.umd.edu (A.M. Memon).
[1] For brevity, whenever possible, we will use the widget label to denote an event, e.g., Cancel, File, etc.
[2] This unaltered example is used to teach students how to use radio buttons.

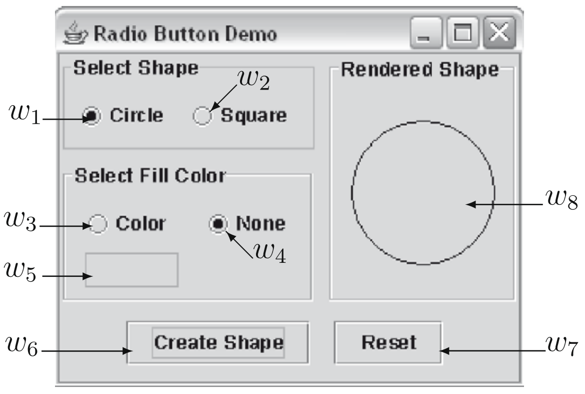

**Fig. 1.** A simple GUI application.

**Table 1**
Events in the simple GUI application of Fig. 1.

| | |
|---|---|
| $e_1$ | Changes shape to circle |
| $e_2$ | Changes shape to square |
| $e_3$ | Enables $w_5$; color updated from text-field |
| $e_4$ | Disables $w_5$; no color in shape |
| $e_6$ | Creates a shape |
| $e_7$ | Resets software to default state |

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. Test cases are also automatically generated from the EIG; the whole process of GUI test-case generation is therefore, fully automatic.

Fig. 2 shows the EIG for the GUI of Fig. 1. Because the EIG is obtained automatically using reverse engineering, it has several limitations – an important limitation is that it does not contain any state-based relationships, e.g., the (enable) relationship between $e_3$ and $e_5$; hence this GUI's EIG is a fully connected directed graph with seven nodes, corresponding to the seven events. The first contribution of this paper is that we address this limitation. Moreover, the EIG is undesirable because its coverage requires a large number of test cases. Consider our EIG-based test case generation algorithm – 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 [20]. Note that some of these test cases may not be executable because of extra EIG edges (e.g., $(e_5, e_3)$ in Fig. 2). 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 [20,30]. For example, a 3-way covering test case is an event sequence $\langle e_1; e_2; e_3 \rangle$ such that $(e_1, e_2)$ and $(e_2, e_3)$ are directed edges in the EIG; similarly, a 4-way covering test case $\langle e_1; e_2; e_3; e_4 \rangle$ 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.

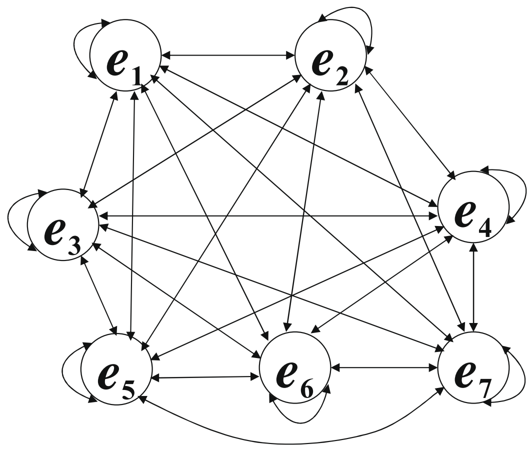

**Fig. 2.** EIG of "Radio Button Demo" GUI.

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 [30]. 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 3. 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. The second contribution of this paper is that we are now able to generate 5-way and above interaction test cases.

With these two contributions, we significantly improve upon the previous ESIG-based approach. We continue to utilize the most important aspect of this previous approach, namely feedback and its use in the ESI relation. First, we formally define the ESI relationships. Second, we generate 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 influence one another's execution behavior (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 those states that do not reflect event interaction. An important property 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., unexecutable 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 and focused test cases that are effective at detecting faults.

The specific contributions of this paper include:

- Formal definition of the run-time relationship between GUI events.
- Generating GUI test cases in batches, where each batch is used to improve the subsequent batch.
- Use of feedback to compute run-time relationships between events and better handle unexecutable test cases.

- Empirical demonstration that the batch-wise approach is successful at identifying complex interactions among GUI event handlers.

The next section provides background and summarizes our previous work on GUI testing. Section 3 explains the ESIG-based approach. Section 4 presents an overview of ALT via an example and Section 5 provides more formal details. Section 6 presents an empirical study to evaluate ALT. Section 8 concludes with a discussion of future work.

## 2. Related work

In the context of this work, execution feedback refers to information obtained during test execution and used to guide further test case generation. This is called *dynamic test case generation* and, to the best of our knowledge, was originally proposed by Miller and Spooner [22]. 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, various types of execution feedback, models, and algorithms have been used for test case generation. For example, branch predicate evaluations along an execution path has been used with a gradient descent approach [12,6,7] and a chaining approach [4], condition–decision coverage has been used with genetic search [21], and object states have been used with a hybrid approach [29]. McMinn [14] provides an excellent survey of metaheuristic search techniques for the automatic generation of test data.

### 2.1. Branch predicate evaluations

Branch predicate evaluation refers to the flow of control during an execution. It has been used with the gradient descent approach to compute an input, i.e., test case, that will execute a given path in the program [12,6,7]. It has also been used with the chaining approach to generate a test case that covers a selected statement [4]. The branch predicate evaluations, which encode control flow information, are collected during software execution on an initial test case. The generation of the test case is modeled as an object function minimization or optimization problem. The evaluation results are applied to gradually adjust the current test case so that it gets closer and closer to the desired test case. One disadvantage of these approaches is that they can get stuck in a local minima during test case generation.

### 2.2. Object properties

Xie and Notkin have developed a feedback-based framework that uses object states to generate new test cases [29]. This framework integrates two techniques: (1) specification-based test generation and (2) dynamic specification inferences for test case generation. This integration provides value considerably beyond what the separate methods can provide alone.

Specification-based test generation is based on formal specifications, which express the desired behavior of a program. However, because formal specifications are difficult to obtain, dynamic specification inference attempts to infer specifications, in the form of operational abstractions, automatically from software execution. The discovered operational abstractions consist of object properties that hold for all the observed executions; these object properties are used to indicate the deficiency of test cases.

The test case generation process starts from an existing test suite. Through executions of these test cases, object states (values of variables and parameters, and return values) are recorded at the entry and exit of method executions. Based on the collected traces and a set of pre-defined axiom-pattern templates, equality patterns are searched to create operational abstractions. By removing or relaxing inferred preconditions on parameter values in the operational abstractions, both legal and illegal test cases are generated. The newly generated test cases are executed. Because they were generated by relaxing inferred preconditions, some of these test cases may cause an uncaught runtime exception. The other, non-crashing test cases are used to obtain new operational abstractions, which are again used to generate additional test cases.

### 2.3. Method-call sequences

Pacheco et al. [23] have improved random unit test generation by incorporating feedback obtained from executing test inputs as they are created. They build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. The key idea of their work is that they build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception indicating a problem. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.

Similarly, Boyapati et al. employ a feedback-based technique to obtain all non-isomorphic inputs (test cases) for a method [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.

### 2.4. Code coverage

Several researchers instrument elements (lines, branches, etc.) of the program code, execute an initial test, 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. [22] use code coverage and decision outcomes to generate floating-point test data.

*Genetic algorithms* have also been used to automatically generate test suites that satisfy the *condition–decision* adequacy criterion [21]. Condition–decision criterion requires that each condition in the program be true for at least one test case and false for at least one test case. A fitness function is defined for each branch. An initial test suite is obtained and executed. The fitness functions are used to evaluate the ‘‘goodness” of each test case. If a test case covers a new condition–decision, it is considered to be ‘‘more fit”. The test cases in the gene pool evolve to obtain a new generation of test cases. The process stops when a desired level of fitness is obtained.

## 2.5. GUI testing

Our own ESIG-based GUI testing approach [30] 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 the runtime 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 discuss details in the next section.

Our ESIG-based GUI testing approach falls under the general umbrella of model-based GUI testing techniques. In earlier work [19,16], we developed an automated GUI testing framework called PATHS. PATHS uses a description of the GUI to automatically generate test cases and test oracles from pairs of initial and goal states by using an AI planner. The other significant work on GUI testing is by White et al. [26,28] who model a GUI in terms of "responsibilities" (user tasks) and their corresponding "complete interaction sequences" (CIS). A CIS is a sequence of GUI objects and selections that may be used to complete a responsibility. Each CIS contains a reduced finite-state machine (FSM) model, which is "traversed" to generate test cases [26]. Other researchers have developed techniques to address isolated problems of GUI testing. For example, a variable finite state machine based approach to generate test cases has been proposed by Shehady and Siewiorek [25].

The most popular GUI testing approach used in practice employs semi-automated tools to do limited testing [5,27]. Examples of some tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module* [11] to create unit tests for GUIs. Other tools include capture and replay tools that "capture" a user session as a test case that can be later "replayed" automatically during regression testing [8]. These tools facilitate only the execution of test cases.

## 3. Modeling the ESI relationship

We now formally define the ESI relationship. But first we introduce some necessary terms.

### 3.1. Preliminaries

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 $S_i$.

GUIs contain two types of windows: (1) *modal windows*[3] (e.g., `FileOpen`, `Print`) that, once invoked, monopolize the GUI interaction, restricting the focus of the user to the range of events within the window until explicitly terminated using a *Termination event* (e.g., using `Ok`, `Cancel`) and (2) *modeless windows* (*e.g.*, the `Find` and `Replace` windows) that do not restrict the user's focus. If, during an execution of the GUI, modal window $\mathcal{M}_1$ is used to open another modal window $\mathcal{M}_2$, then $\mathcal{M}_1$ is called the *parent* of $\mathcal{M}_2$ for that execution.

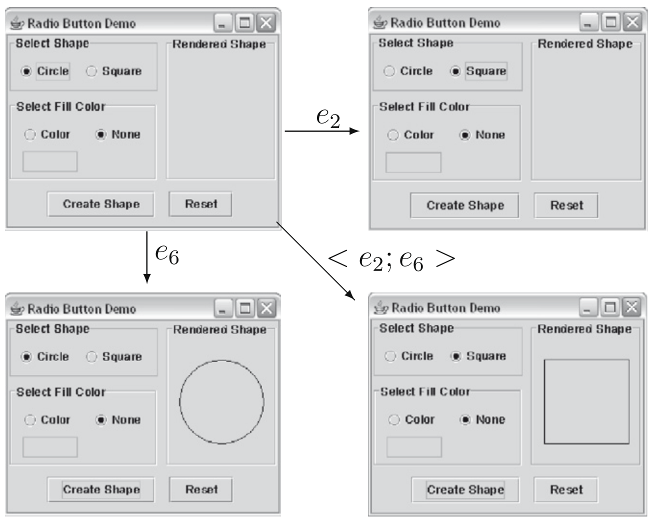

**Fig. 3.** Execution of events $e_2$ and $e_6$.

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 to generate these test cases [20]. 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 adjacent to the edge are generated on-demand at test-execution time.

### 3.2. The ESI relationship

Informally, event $e_x$ and $e_y$ are related via the ESI relation, if, when executed together in a sequence $\langle e_x; e_y \rangle$, 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 Fig. 3 (this application was described earlier). 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 $\langle e_2; e_6 \rangle$ 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 $\langle e_2; e_6 \rangle$ 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 Fig. 4; 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 [24]; they can certainly be used successfully in this example. However, in general, the limitations of static analysis in the presence of mul-

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

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

$e_2$**'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**

**Fig. 4.** Source code of two event handlers in the radio button example.

ti-language GUI implementations, callbacks for event handlers, virtual function calls, reflection, and multi-threading are well known [24]. 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 term "different from" used at the beginning of this section 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 $(e_x(S_0) \neq e_y(e_x(S_0))) \vee (e_y(S_0) \neq e_y(e_x(S_0)))$, where events $e_x$ and $e_y$, and state $S_0$ are universally quantified (for improved readability, we will skip the universal quantification). 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., $e_x(S_0) = \{\boxslash_x, \square_y\}$, $e_y(S_0) = \{\square_x, \boxslash_y\}$, and $e_y(e_x(S_0)) = \{\boxslash_x, \boxslash_y\}$. 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 formalize the notion of interacting events by developing formal predicates [30]. We note that these predicates are not special cases of $\mathcal{P}$.

The predicate for the example of Fig. 3 is written as: $\exists w \in W$, $p \in P_w$, $v \in V_p$, $v' \in V_p$, *s.t.*[4] $((v \neq v') \wedge ((w,p,v) \notin \{S_0 \cap e_x(S_0)\}) \wedge ((w,p,v) \in e_y(S_0)) \wedge ((w,p,v') \in e_y(e_x(S_0))))$; it is read as: "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 $\langle e_x; e_y \rangle$ is executed, i.e., the value of $w$'s property $p$ changes from $v$ to $v'$". This predicate evaluates 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 $\langle e_2; e_6 \rangle$ is executed.

It turns out that the example illustrated in Fig. 3 is just one case of how the GUI state may be used to pinpoint interactions between event handlers – there are many more. In our previous work, we defined six cases; we now extend our previous work by identifying a total of 12 cases. They are presented because they were encountered numerous times in our work on GUI testing. The twelve cases will describe (as evaluative predicates) situations in which events $e_1$ 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 cases, $e_1$ and $e_2$ are system-interaction events in modeless windows; this situation will be called *Context 1*.

**Case 1.** There is at least one widget $w$ with property $p$ with initial value $v$ (hence the triple $(w, p, v)$ is in $S_0$), which is not affected by the individual events $e_1$ or $e_2$ (the triple is also in $e_1(S_0)$ and $e_2(S_0)$); however, it is modified when the sequence $\langle e_1; e_2 \rangle$ is executed, i.e., the value of $w$'s property $p$ changes from $v$ to $v'$. Formally, $\exists w \in W,\ p \in P_w,\ v \in V_p,\ v' \in V_p,\quad s.t.((v \neq v') \wedge ((w,p,v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \ \wedge ((w,p,v') \in e_2(e_1(S_0))))$.

Fig. 5 gives an example of Case 1. This is a "GUI Demo" application with several widgets. The Fill with color check-box fills the currently selected shape (highlighted with a deep grey border) with the chosen color determined by the radio buttons White and Blue. Check-box Fill with pattern determines whether to fill the selected shape with a pattern. Checking Apply to all sets all shapes in the right panel with the same color and pattern.

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

**Case 2.** 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 $\langle e_1; e_2 \rangle$. Formally, $\exists w \in W,\ p \in P_w,\ v \in V_p,\ v' \in V_p,\ v'' \in V_p,\ s.t.\ ((v \neq v') \wedge (v' \neq v'') \wedge ((w,p,v) \in \{S_0 \cap e_1(S_0)\}) \wedge ((w,p,v') \in e_2(S_0)) \wedge ((w,p,v'') \in e_2(e_1(S_0))))$.

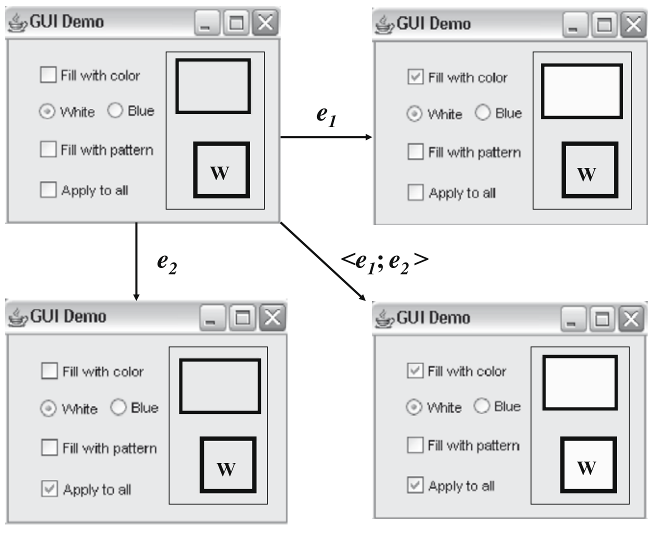

**Fig. 5.** Case 1: $e_1$: *Check* Fill with color; $e_2$: *Check* Apply to all.

[4] Notation for "such that".

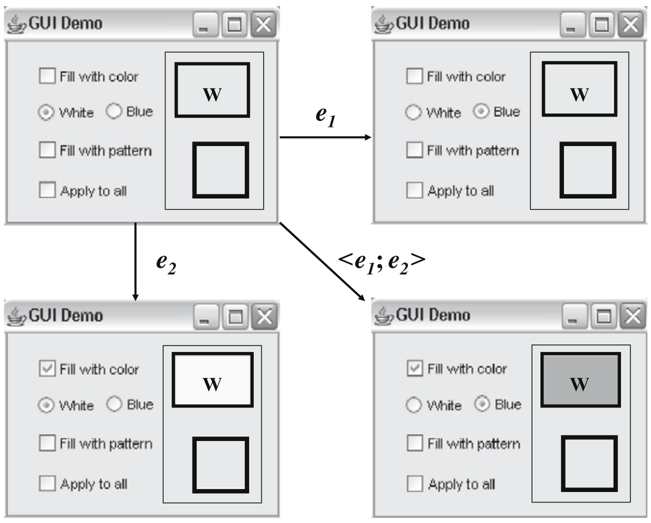

**Fig. 6.** Case 2: $e_1$: *Click radio button* `Blue`; $e_2$: *Check* `Fill with color`.

An example of Case 2 using the "`GUI Demo`" application is given in Fig. 6, where $e_1$ now represents *Click radio button* `Blue` and $e_2$ is *Check* `Fill with color`. The initial state has the rectangle selected and color is set to white. Individually, in this initial state, event $e_1$ sets the current color to blue; event $e_2$ fills the rectangle with the white color. However, executing $\langle e_1; e_2\rangle$ now fills the rectangle with the color blue. Case 2 applies here as $e_1$ influences $e_2$ execution; the widget (marked with **W**) is not modified by $e_1$; it is modified by $e_2$; however, it is modified differently by the sequence $\langle e_1; e_2\rangle$.

A variation of Case 2 is called *Case 2.1*, in which the roles of $e_1$ and $e_2$ are exchanged with the combined sequence $\langle e_1; e_2\rangle$ remaining the same.

**Case 3.** 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 $\langle e_1; e_2\rangle$. Formally, $\exists w \in W,\ p \in P_w,\ v \in V_p,\ v' \in V_p,\ v'' \in V_p,\ \bar{v} \in V_p,\ s.t.\ ((v \neq v') \wedge (v \neq v'') \wedge (v'' \neq \bar{v}) \wedge ((w,p,v) \in S_0) \wedge ((w,p,v') \in e_1(S_0)) \wedge ((w,p,v'') \in e_2(S_0))\ \wedge ((w,p,\bar{v}) \in e_2(e_1(S_0))))$.

Fig. 7 shows one example of this case using the "`GUI Demo`" application. In this example, the initial state has `Fill with color` checked, white is set to be the current color and the rectangle is selected. Event $e_1$ here is *Click radio button* `Blue` and $e_2$ is *Check* `Fill with pattern` that fills the current shape with a pattern. Events $e_1$ and $e_2$ modify the rectangle individually; however, executing $\langle e_1; e_2\rangle$ now modifies the rectangle differently. Therefore, $e_1$ influences $e_2$, i.e., resulting in different modification of the existing widget (marked with **W**), and Case 3 applies.

### 3.3. Object creation

The first three cases 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. According to the terms introduced in Section 3.1, a widget is "created" by adding all its widget–property–value triples to the current state. The next cases handle such newly created widgets.

**Case 4.** There is at least one *new* widget $w$ with property $p$ and value $v$ in state $e_2(e_1(S_0))$, i.e., it is created by event sequence $\langle e_1; e_2\rangle$; but it does not exist in state $S_0$ and could not be created by either $e_1$ or $e_2$ individually, i.e., no triple involving widget $w$ exists in any of the states $S_0, e_1(S_0)$ and $e_2(S_0)$. Formally, $\exists w \in W$, $\exists p \in P_w,\ \exists v \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p,\quad s.t.\ (((w,\bar{p},\bar{v}) \notin S_0) \wedge ((w,\bar{p},\bar{v}) \notin e_1(S_0)) \wedge ((w,\bar{p},\bar{v}) \notin e_2(S_0)) \wedge ((w,p,v) \in e_2(e_1(S_0))))$.

The example using "`GUI Demo 1`" for this case is shown in Fig. 8. In this application, checking `Read-only` forbids inserting text into the bottom panel; checking `Select All` selects all the widgets in the bottom panel. Clicking button `Insert` creates a text-field for inputing text, and clicking button `Cut` removes the current selection (either text-field in the panel or text in text-field). To illustrate Case 4, assume that the initial state has `Read-only` checked and an empty bottom panel. Event $e_1$ unchecks `Read-only` and $e_2$ clicks the button `Insert`. It is clear that $e_2$ cannot insert the text-field into the bottom panel with `Read-only` checked. However, when executing $\langle e_1; e_2\rangle$, $e_1$ first removes the read-only restriction to the panel, and then $e_2$ creates a text-field. Hence, $e_1$ influences $e_2$ by making it create a new widget (marked with **W**) previously non-existent in the initial state; Case 4 is applicable here.

**Case 5.** There is at least one widget $w$ that does not exist in the initial state $S_0$; it is created by $e_1$ with property $p$ and value $v$; $e_2$ does not create $w$. However, $w$ is created differently when the

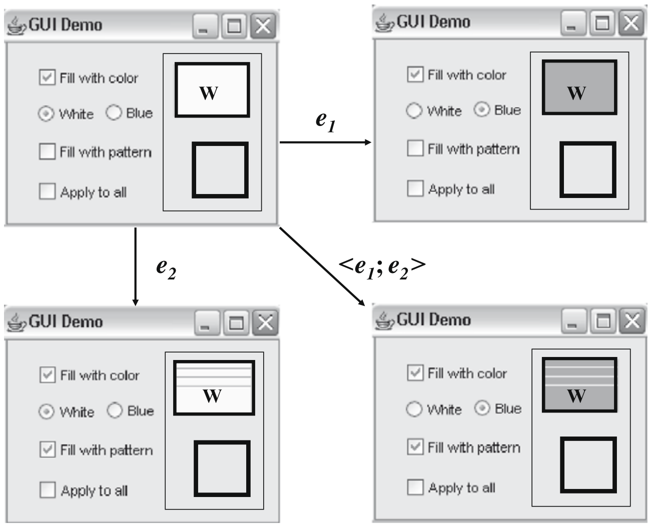

**Fig. 7.** Case 3: $e_1$: *Click radio button* `Blue`; $e_2$: *Check* `Fill with pattern`.

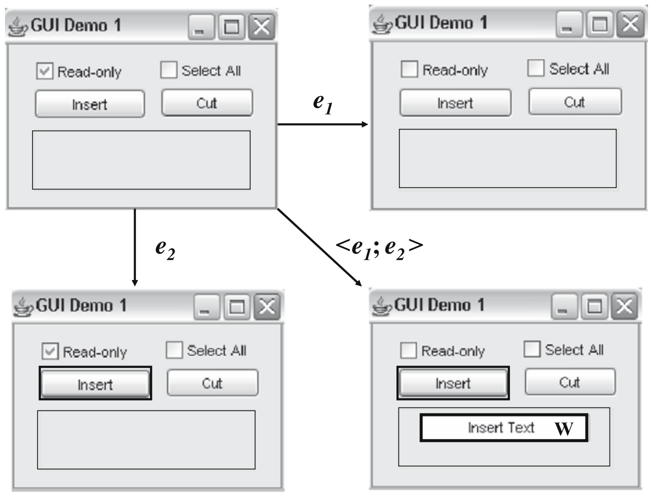

**Fig. 8.** Case 4: $e_1$: *Uncheck* `Read-only`; $e_2$: *Click button* `Insert`.

sequence $\langle e_1; e_2\rangle$ is executed, i.e., the value of $w$'s property $p$ is now $v'$ (not $v$). Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\ s.t.\ ((v \neq v') \wedge ((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v') \in e_2(e_1(S_0))))$.

A variation of Case 5 is called *Case 5.1* in which the roles of $e_1$ and $e_2$ are exchanged and the combined sequence $\langle e_1; e_2\rangle$ remains the same. Fig. 9 shows an example for *Case 5.1*. The "`GUI Demo 2`" application is used in this example. It is used to create a table with a given number of rows and columns. One can input the desired number of rows and columns in the text-fields labeled as `# of Rows` and `# of Columns`. By clicking either the button `Set Row` or `Set Column`, a table with the specified number of rows and columns is created in the bottom panel; or if a table already exists in the panel, the number of rows and columns are changed to the given numbers.

For *Case 5.1*, the initial state has the row and column number both set to 2 and an empty bottom panel. In this initial state, event $e_1$ inputs 1 into the text-field to set the number of rows; $e_2$ clicks the button `Set Row` and creates a new table widget with two rows. However, when executing $\langle e_1; e_2\rangle$, a table with only one row is created. Therefore, $e_1$ influences $e_2$ and modifies its creation of the new widget, i.e., the table (marked with **W**); *Case 5.1* is applicable here.

**Case 6.** There is at least one *new* widget $w$ that does not exist in state $S_0$; but $w$ is created by $e_1$ and $e_2$ individually. However, it is created by the sequence $\langle e_1; e_2\rangle$ with a different value $v''$ for property $p$. Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p,\ v'' \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\ \ s.t.\ ((v' \neq v'') \wedge ((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, p, v') \in e_2(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$.

This case is also demonstrated using the "`GUI Demo 2`" application in Fig. 10. In this example, the initial state has row and column number set to 2 and an empty bottom panel. Event $e_1$ clicks the button `Set Row` and $e_2$ clicks the button `Set Column`. Event $e_1$ individually creates a new table with one column and two rows; event $e_2$ creates a one-row and two-column table. Executing $\langle e_1; e_2\rangle$ creates a table with two rows and two columns. Hence, $e_1$ influence $e_2$, i.e., resulting in different creation of a new widget (marked with **W**), and Case 6 is applicable here.

### 3.4. Object destruction

Event execution may also "remove" existing widgets from a GUI, e.g., by cutting selected components. According to the terms

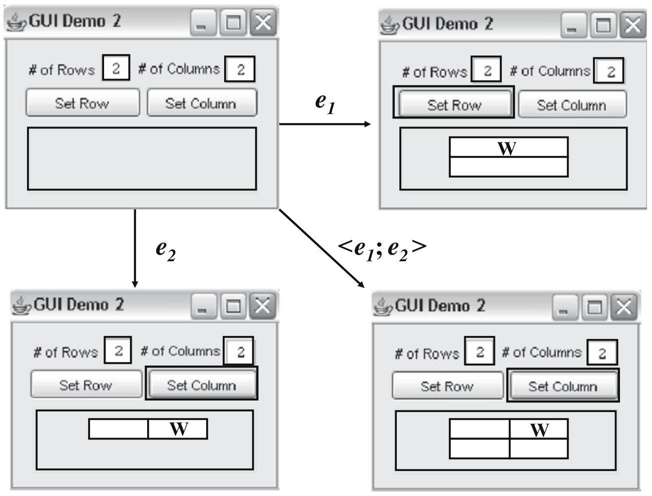

**Fig. 10.** Case 6: $e_1$: *Click button* `Set Row`; $e_2$: *Click button* `Set Column`.

introduced in Section 3.1, a widget is "deleted" by removing all its widget–property–value triples from the current state. The next three cases handle removed widgets.

**Case 7.** There is at least one widget $w$ that exists in the initial state $S_0$ with property $p$ and value $v$; it is not removed by $e_1$ and $e_2$ individually. However, it is removed when the sequence $\langle e_1; e_2\rangle$ is executed. Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p,\ \ v'' \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\ \ s.t.\ (((w, p, v) \in S_0) \wedge ((w, p,\ \ v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(e_1(S_0))))$.

Case 7 is illustrated via an example using "`GUI Demo 1`" application in Fig. 11. In this example, the initial state has all check-boxes unchecked and a text-field with text `Hello World` in the bottom panel. Event $e_1$ checks `Select All` and selects the text in the text-field; $e_2$ clicks the button `Cut`. They individually do not remove the text-field in the bottom panel. However, executing $\langle e_1; e_2\rangle$ results in an empty bottom panel. Therefore, $e_1$'s selection of widgets influences $e_2$ and enables it to remove an existing widget (marked with **W**); Case 7 is applicable here.

**Case 8.** There is at least one widget $w$ that does not exist in the initial state $S_0$; it is created by $e_1$ with property $p$ and value $v$; $e_2$ does not create $w$ individually. However, it is removed when the sequence $\langle e_1; e_2\rangle$ is executed. Formally, $\exists w \in W,\ \exists p \in$

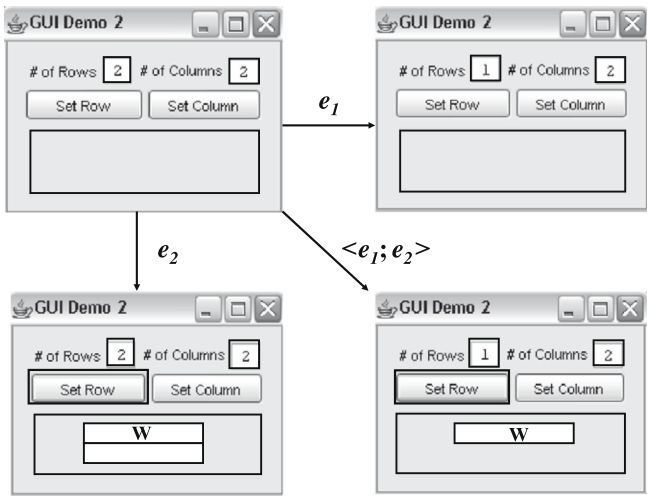

**Fig. 9.** Case 5.1: $e_1$: *Input row number*; $e_2$: *Click button* `Set Row`.

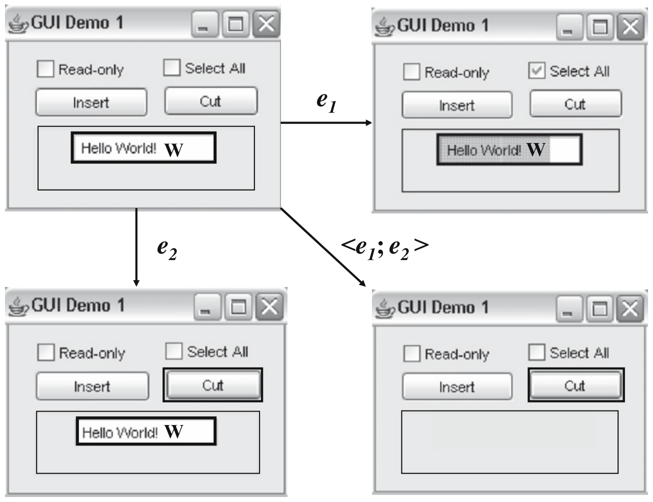

**Fig. 11.** Case 7: $e_1$: *Check* `Select All`; $e_2$: *Click button* `Cut`.

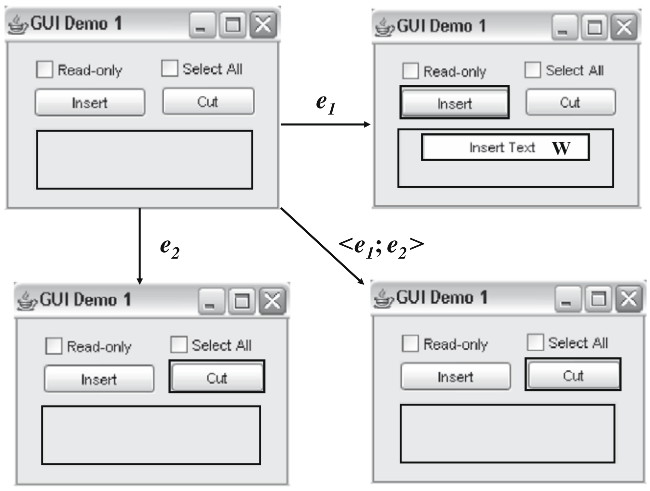

**Fig. 12.** Case 8: $e_1$: *Click button* `Insert`; $e_2$: *Click button* `Cut`.

$P_w,\ \exists v \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\quad s.t.\ (((w,\bar{p},\bar{v}) \notin S_0) \wedge \quad ((w,p,v) \in e_1(S_0)) \wedge ((w,\bar{p},\bar{v}) \notin e_2(S_0))\ \wedge ((w,\bar{p},\bar{v}) \notin e_2(e_1(S_0))))$.

Case 8 is demonstrated using "`GUI Demo 1`" in Fig. 12. The initial state in this example has all check-boxes unchecked and an empty bottom panel. Event $e_1$ is *Click button* `Insert`; $e_2$ is *Click button* `Cut`. Event $e_1$ inserts a text-field into the bottom panel; $e_2$ removes selected items in the panel if there are any. However, executing $\langle e_1; e_2 \rangle$ first inserts the text-field (selected at the time of creation) by $e_1$, then $e_2$ removes the text-field. Therefore, $e_1$ influences $e_2$, i.e., $e_2$ removes widget (marked with **W**) newly created by $e_1$, and Case 8 is applicable here.

A variation of Case 8 is called *Case 8.1* in which the roles of $e_1$ and $e_2$ are exchanged and the combined sequence $\langle e_1; e_2 \rangle$ remains the same. Fig. 13 shows an example illustrating *Case 8.1*. In this example, the initial state has unchecked check-boxs and an empty panel. Event $e_1$ is *Check* `Read-only` and $e_2$ is *Click button* `Insert`. Event $e_2$ creates a text-field widget in the panel. However, executing $\langle e_1; e_2 \rangle$ does nothing to the panel because $e_1$ first sets the panel to read-only; $e_2$ cannot create a new text-field. Hence, $e_1$ influences $e_2$'s creation of the new widget (marked with **W**; *Case 8.1* is applicable here.

**Case 9.** There is at least one *new* widget $w$ that does not exist in state $S_0$; but it is created by $e_1$ with property $p$ and value $v$, and by $e_2$ with property $p$ and value $v'$ individually. However, it is removed by the sequence $\langle e_1; e_2 \rangle$, i.e., no triple involving widget $w$ is in state $e_2(e_1(S_0))$. Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\ s.t.\ (((w,\bar{p},\bar{v}) \notin S_0) \wedge ((w,p,v) \in e_1(S_0)) \wedge ((w,p,v') \in e_2(S_0)) \wedge ((w,\bar{p},\bar{v}) \notin e_2(e_1(S_0))))$.

### 3.5. Object destruction followed by creation

The next two cases describe interactions in which existing widgets are removed by individual events (all its widget–property–value triples are removed from the current state) but are re-created (its widget–property–value triples are added to the current state) by the sequence $\langle e_1; e_2 \rangle$.

**Case 10.** There is at least one widget $w$ that exists in the initial state $S_0$ with property $p$ and value $v$; it is removed by $e_2$; it is modified by $e_1$ with property $p$ and value $v'$. However, it is re-created when the sequence $\langle e_1; e_2 \rangle$ is executed, i.e., the triple $(w,p,v'')$ is in state $e_2(e_1(S_0))$. Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p,\ \exists v'' \in V_p,\ \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p,\quad s.t.\ (((w,p,v) \in S_0) \wedge ((w,\bar{p},\bar{v}) \notin e_2(S_0)) \wedge ((w,p,v') \in e_1(S_0)) \wedge ((w,p,v'') \in e_2(e_1(S_0))))$.

The example shown in Fig. 14 demonstrates Case 10. The application used here is "`GUI Demo 3`." In this application, clicking button `New Layer` creates a radio button labeled with a layer number in the bottom panel. Clicking button `Remove Layer` removes the radio button labeled with the highest layer number. The example has the initial state with a created `Layer 1` in the panel. Event $e_1$ clicks button `New Layer` and creates `Layer 2`; $e_2$ clicks button `Remove Layer` and removes the existing `Layer 1`. However, when executing $\langle e_1; e_2 \rangle$, $e_2$ now removes the newly created `Layer 2` instead of the original `Layer 1`. Hence, $e_1$ influences $e_2$, i.e., keeping the widget (marked with **W**) that would have been removed. Case 10 captures this scenario.

A variation of Case 10 is called Case 10.1 in which the roles of $e_1$ and $e_2$ are exchanged and the combined sequence $\langle e_1; e_2 \rangle$ remains the same.

**Case 11.** There is at least one widget $w$ that exists in the initial state $S_0$ with property $p$ and value $v$; it is removed by $e_1$ and $e_2$ individually. However, it is re-created when the sequence $\langle e_1; e_2 \rangle$ is executed, i.e., the triple $(w,p,v')$ is in state $e_2(e_1(S_0))$. Formally, $\exists w \in W,\ \exists p \in P_w,\ \exists v \in V_p,\ \exists v' \in V_p, \forall \bar{p} \in P_w,\ \forall \bar{v} \in V_p, s.t.\ (((w,p,v) \in S_0) \wedge ((w,\bar{p},\bar{v}) \notin e_1(S_0)) \wedge ((w,\bar{p},\bar{v}) \notin e_2(S_0)) \wedge ((w,p,v') \in e_2(e_1(S_0))))$.

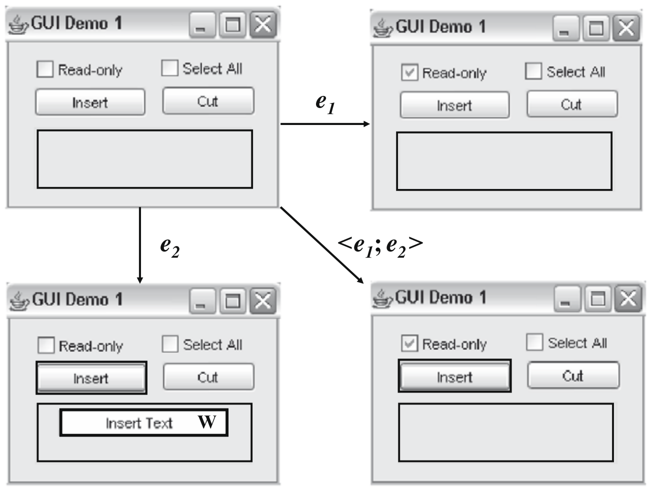

**Fig. 13.** Case 8.1: $e_1$: *Check* `Read-only`; $e_2$: *Click button* `Insert`.

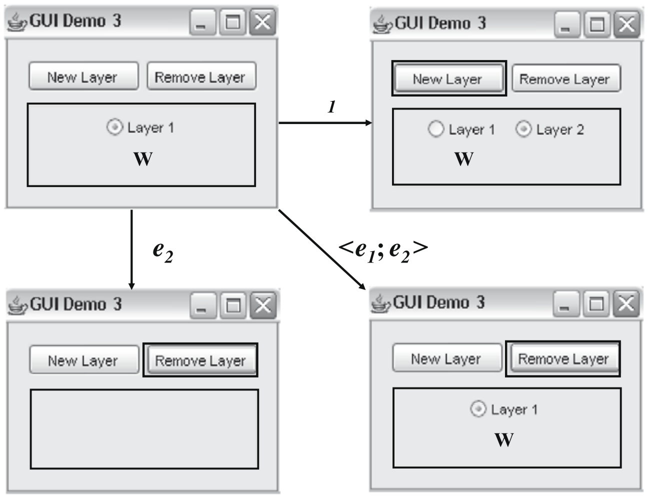

**Fig. 14.** Case 10: $e_1$: *Click button* `New Layer`; $e_2$: *Click button* `Remove Layer`.

### 3.6. Enabling and disabling objects

Finally, a common occurrence of event interaction in GUIs are enabling and disabling widgets, which may be modeled as the widget's ENABLED property being set to TRUE or FALSE.

**Case 12.** 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 $\texttt{EXEC}(e_2, w)$. Formally, $\exists w \in W,\ \texttt{ENABLED} \in P_w,\ \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}},\ s.t.\ (((w, \texttt{ENABLED}, \texttt{FALSE}) \in S_0) \wedge\ ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_1(S_0)) \wedge \texttt{EXEC}(e_2, w))$.

### 3.7. Additional contexts

Modal windows create special situations for Cases 1–12 due to the presence of termination events
(e.g., Ok, Cancel). 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 1–4*.

**Context 1.** 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 $\langle e_1; \texttt{TERM}\rangle$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_2; \texttt{TERM}\rangle$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $\langle e_1; e_2; \texttt{TERM}\rangle$. All the predicates defined in Cases 1–12 apply, using these modified definitions, for $e_1$ and $e_2$ in the same modal window.

**Context 2.** 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 $\langle e_1; \texttt{TERM}\rangle$, $e_2(S_0)$ 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 $\langle e_1; \texttt{TERM}; e_2\rangle$. All the predicates defined in Cases 1–12 apply.

**Context 3.** If $e_1$ is associated with a widget contained in a modal window with termination event $\texttt{TERM}_1$, and $e_2$ is associated with a widget contained in another modal window with termination event $\texttt{TERM}_2$ (i.e., the window that is derived through another path from the main window) then $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_1; \texttt{TERM}_1\rangle$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_2; \texttt{TERM}_2\rangle$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $\langle e_1; \texttt{TERM}_1; \texttt{R}; e_2; \texttt{TERM}_2\rangle$, where R is the sequence of events needed to open the modal window containing $e_2$. All the predicates defined in Cases 1–12 apply.

The 12 cases described in this section are not exhaustive and new cases may be added, as needed, in the future. Our implementation of the predicates allow users to add new predicates by implementing a Java method for each new predicate. The method is invoked, whereby the predicate is evaluated. Our future research will involve consolidating some of the cases into fewer abstract cases; instantiations of widgets, properties, values, and states will help to define specific situations. However, we feel that this will also make the implementation of individual predicates more complex and less self-contained.

### 3.8. ESI defined

There is an *Event Semantic Interaction* (ESI) relationship between two events $e_1$ and $e_2$ if and only if at least one of the predicates in Cases 1–12 evaluates to TRUE; this relationship is written as $e_1 \overset{n(m)}{\rightarrow} e_2$, where the number $n$ is one of the case numbers 1–12; $m$ 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 $\langle e_1; e_2\rangle$, the ESI relationship is not symmetric.

## 4. The ALT process

We now present the steps of the ALT process via the example application of Fig. 1.

1. *Obtain the event-interaction graph (EIG).* As mentioned in Section 2, this is done via automated reverse engineering techniques [18]. 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 $\langle e_3; e_5\rangle$ 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 or 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. Our current implementation computes these relationships automatically; the predicates (implemented as Java methods in a queue) are invoked one-by-one to determine if they are TRUE, 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 5.1 and represented as $e_2 \overset{5.1(1)}{\rightarrow} e_6$.
Another interesting relation in this example is $e_6 \overset{5(1)}{\rightarrow} e_2$, i.e., $e_6$ is ESI related to $e_2$. In the default start state, $e_6$ creates a circle. However, the sequence $\langle e_6; e_2\rangle$ yields a square because $e_2$ changes the shape. The predicate (number 5 in our set) used to compute this relation is described in Section 3.4. This interaction is due to the created variable shared between the code of $e_6$ and $e_2$.
Another ESI relationship is $e_3 \overset{12(1)}{\rightarrow} e_5$. The predicate used to obtain it is number 12 in our set. 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: $e_2 \overset{5.1(1)}{\rightarrow} e_6$, $e_6 \overset{5(1)}{\rightarrow} e_2$, and $e_3 \overset{12(1)}{\rightarrow} e_5$. The first two are used to extend two of the 2-way covering test cases $\langle e_2; e_6\rangle$ and $\langle e_6; e_2\rangle$ to $\langle e_2; e_6; e_2\rangle$ and $\langle e_6; e_2; e_6\rangle$, 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: $\langle e_3; e_5; e_1\rangle$, $\langle e_3; e_5; e_2\rangle$, $\langle e_3; e_5; e_3\rangle$, $\langle e_3; e_5; e_4\rangle$, $\langle e_3; e_5; e_5\rangle$, $\langle e_3; e_5; e_6\rangle$, and $\langle e_3; e_5; e_7\rangle$. 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 $\langle e_x; e_y; e_z\rangle$ into two parts: $\langle e_x; e_y\rangle$ 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 $\langle e_x; e_y\rangle$) 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 $\langle e_3; e_5; e_6\rangle$. This is rewritten as $\langle E_X; e_6\rangle$, with $E_X$ being $\langle e_3; e_5\rangle$; 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 5.1 applies; $\langle e_3; e_5\rangle \overset{5.1(1)}{\rightarrow} e_6$.

   Because $\langle e_3; e_5\rangle \overset{5.1(1)}{\rightarrow} e_6$ and $e_6 \overset{5(1)}{\rightarrow} e_2$ (as computed earlier), we extend $\langle e_3; e_5; e_6\rangle$ to the 4-way test case $\langle e_3; e_5; e_6; e_2\rangle$.

   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 $\langle e_3; e_5; e_6; e_2\rangle$ is rewritten as $\langle E_X; e_2\rangle$; 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 $\langle e_3; e_5; e_6\rangle \overset{5(1)}{\rightarrow} e_2$. And we already know that $e_2 \overset{5.1(1)}{\rightarrow} e_6$. Hence, only one 5-way covering test case is generated $\langle e_3; e_5; e_6; e_2; e_6\rangle$.

```
          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

**Fig. 15.** Test coverage of the source code for the "`Radio Button Demo`" GUI – part 1.

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. Figs. 15 and 16 shows the event-handler code as well as some helper methods. The statement coverage is summarized as a vector of 4 check-boxes ☑☑☑☑ 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

```
1            RBExample::ResetAction(ActionEvent evt) {
2   ☑☐☐☐         square.setSelected(true);
3   ☑☐☐☐         none.setSelected(true);
4   ☑☐☐☐         colorText.setText(``black'');
5   ☑☐☐☐         colorText.setEditable(false);
6   ☑☐☐☐         currentShape = SHAPE_NONE;
7   ☑☐☐☐         imagePanel.setShape(currentShape);
8   ☑☐☐☐         currentColor = COLOR NONE;
9   ☑☐☐☐         imagePanel.setFillColor(currentColor);
10  ☑☐☐☐         imagePanel.repaint(); }
```

$e_7$'s Event Handler

```
1            ImagePanel::paintComponent(Graphics g) {
2   ☑☑☑☑         clear(g);
3   ☑☑☑☑         Graphics2D g2d = (Graphics2D)g;
4   ☑☑☑☑         if (currentShape == SHAPE_CIRCLE) {
5   ☑☑☑☐             if (currentColor == COLOR_NONE) {
6   ☑☑☐☐                 g2d.setPaint(Color.black);
7   ☑☑☐☐                 g2d.draw(circle); }
8   ☑☑☑☑             else {
9   ☑☑☑☑                 g2d.setPaint(currentColor);
10  ☑☑☑☑                 g2d.fill(circle); } }
11  ☑☑☑☑         else if (currentShape == SHAPE_SQUARE) {
12  ☑☑☑☐             if (currentColor == COLOR_NONE) {
13  ☑☑☐☐                 g2d.setPaint(Color.black);
14  ☑☑☐☐                 g2d.draw(square); }
15  ☐☐☑☑             else {
16  ☐☐☑☑                 g2d.setPaint(currentColor);
17  ☐☐☑☑                 g2d.fill(square); } } }
18           ImagePanel:setFillColor(int inputColor) {
19  ☑☑☑☑         switch (inputColor) {
20  ☐☑☑☑             case COLOR_BLACK:
21  ☐☑☑☑                 currentColor=Color.black;
22  ☐☑☑☑                 break;
23  ☐☑☑☑             case COLOR_RED:
24  ☐☑☑☑                 currentColor=Color.red;
25  ☐☑☑☑                 break;
26  ☐☑☑☑             case COLOR_GREEN:
27  ☐☑☑☑                 currentColor=Color.green;
28  ☐☑☑☑                 break;
29  ☑☑☑☑             default:
30  ☑☑☑☑                 currentColor=Color.gray; } }
```

The ImagePanel Class

**Fig. 16.** Test coverage of the source code for the "`Radio Button Demo`" GUI – part 2.

interactions across classes. In Section 6, 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). Fifth, some event handlers (i.e., `CircleAction ()`, `NoneAction ()`, and `ResetAction ()`) were executed only by the 2-way test cases. Because their corresponding events were not involved in any ESI relationships, they were never used in any of the 3-, 4-, and 5-way test cases. Examining their code reveals that these event handlers are fairly simple with very little branching; all of their code is completely covered by the 2-way 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 an important aspect of GUI testing that will be investigated in future research. More specifically, 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.

```
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
}
```

**Fig. 17.** The ALT algorithm.

## 5. 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 $\langle e_1; e_2; \ldots; e_i \rangle$ into two parts: (1) a macro event $E_X = \langle e_1; e_2; \ldots; e_{i-1} \rangle$ 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 $\langle e_1; e_2; \ldots; e_i; e_x \rangle$ 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 preceding 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`.

All these helper functions have been implemented in Java and are invoked by an algorithm that generates test cases fully automatically. The algorithm is shown in Fig. 17. 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 and 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:

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 and 9 provide a strict condition to test-case extension, i.e., not only must $E_X$ be 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 "diverse" 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. We examine the space requirements for this table in our empirical study, presented next.

## 6. Empirical study

The test cases obtained from the ALT algorithm may 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?

### 6.1. Study procedure

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 Fig. 17.
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.

*Step 1*: Selection of subject applications. We downloaded four popular GUI-based OSS (FreeMind 0.8.0, GanttProject 2.0.1, jEdit 4.2, OmegaT 1.7.3) from SourceForge. FreeMind and GanttProject have been used in our previous experiments [17].

1. *FreeMind*,[5] which is a mind-mapping[6] software written in Java. It has an *all-time activity*[7] of 99.72%.
2. *GanttProject*,[8] which is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import and export (MS Project, HTML, PDF, spreadsheets). It has an all-time activity of 98.12%.
3. *jEdit*,[9] which is a programmer's text editor written in Java. It uses the Swing toolkit for the GUI and can be configured as a powerful IDE. When tested, it had an all-time activity of 99.95%.
4. *OmegaT*,[10] which is a multi-platform Computer Assisted Translation tool with fuzzy matching, translation memory, keyword search and glossaries. It has an all-time activity of 99.80%.

The characteristics of these OSS are shown in Table 2. The entries show that the applications have non-trivial sizes in terms of GUIs elements, containing more than a dozen windows (*Windows*) and hundreds of widgets (*Widgets*), as reported by the GUITAR[11] tool. They also have large code-bases, reported by the Metrics[12] software, in terms of non-comment lines of code (*LOC*), number of classes (*Classes*) and methods (*Methods*). All of the above 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* [10] 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 and seed test suites; execution of seed suite; computation of ESI relations: We used the GUITAR tool to generate, using automated reverse engineering, the EIGs of all subject applications. The seed suite was also generated automatically and executed without any human intervention. The GUI's run-time state was recorded during test execution. We fixed all the detected faults in the applications. The feedback was used to obtain the ESIs for each application.

*Step 3*: Execution of ALT algorithm: Our tool used the initial set of ESI relations to obtain the 3-way test cases. The number of test cases is shown in Table 3. These test cases were executed automatically and the algorithm was invoked again. This process continued until ALT returned an empty test suite. Table 4 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 "–"

[5] http://sourceforge.net/projects/freemind.
[6] http://en.wikipedia.org/wiki/Mind_map.
[7] This measure is useful for users interested in knowing which of the projects on SourceForge.net are the most popular historically. The software from projects with a high all-time activity has been downloaded by thousands of people and are most frequently mature and stable applications which can be of immediate benefit to their users.
[8] http://sourceforge.net/projects/ganttproject.
[9] http://sourceforge.net/projects/jedit.
[10] http://sourceforge.net/projects/omegaT.
[11] http://guitar.sourceforge.net/.
[12] http://metrics.sourceforge.net/.

**Table 2**
Subject applications for study 1.

| Subjects | Windows | Widgets | LOC | Classes | Methods |
|---|---|---|---|---|---|
| FreeMind | 30 | 611 | 13,463 | 765 | 3114 |
| GanttProject | 18 | 326 | 22,711 | 840 | 5189 |
| jEdit | 27 | 498 | 48,444 | 829 | 5582 |
| OmegaT | 18 | 228 | 22,708 | 274 | 1522 |
| Total | 93 | 1663 | 107,326 | 2798 | 15,407 |

**Table 3**
Test cases generation.

| | *i*-Way suites | | | | |
|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 |
| *FreeMind* | | | | | |
| EIG | 1.72*e*8 | 9.56*e*10 | 5.31*e*13 | 2.95*e*16 | 1.64*e*19 |
| ESIG | 10208 | (122426) | (1690861) | ((21857767) | (353090927) |
| ALT | 10208 | 2821 | 11 | 2 | – |
| *GanttProject* | | | | | |
| EIG | 4.94*e*6 | 7.17*e*9 | 2.09*e*12 | 6.07*e*14 | 1.77*e*17 |
| ESIG | 3070 | 14742 | 27933 | (63994) | (125362) |
| ALT | 3070 | 2229 | 226 | 34 | 4 |
| *jEdit* | | | | | |
| EIG | 9.17*e*7 | 4.14*e*10 | 1.87*e*13 | 8.42*e*15 | 3.80*e*18 |
| ESIG | 7572 | 84488 | (1024424) | (10225602) | (105931205) |
| ALT | 7572 | 1258 | 738 | 171 | – |
| *OmegaT* | | | | | |
| EIG | 7.65*e*6 | 1.51*e*9 | 2.97*e*11 | 5.85*e*13 | 1.15*e*16 |
| ESIG | 2335 | 8935 | 42859 | (219415) | (1135743) |
| ALT | 2335 | 1440 | – | – | – |

**Table 4**
ESI relationships.

| 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 5**
Number of entries in the *FindState*() table.

| Subjects | 1-way | 2-way | 3-way | 4-way | 5-way | 6-way | 7-way |
|---|---|---|---|---|---|---|---|
| FreeMind | 256 | 309,136 | 10,208 | 2821 | 11 | 2 | – |
| GanttProject | 291 | 84,681 | 3070 | 2229 | 226 | 34 | 4 |
| jEdit | 452 | 204,304 | 7572 | 1258 | 738 | 171 | – |
| OmegaT | 197 | 38,809 | 2335 | 1440 | – | – | – |

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.

**Table 6**
Fault detection.

| Subject application | Technique | *i*-Way test suite | | |
|---|---|---|---|---|
| | | 3 | 4 | 5 |
| FreeMind | ESIG | ☑☑ | – | – |
| | ALT | ☑☑ | – | – |
| GanttProject | ESIG | ☑☑☑☐ | – | ☐ |
| | ALT | ☑☑☑☑ | – | ☑ |
| jEdit | ESIG | ☑☑ | ☐ | – |
| | ALT | ☑☑ | ☑ | – |
| OmegaT | ESIG | – | – | – |
| | ALT | – | – | – |

**Table 7**
Time (in hours) required to execute 1- and 2-way, and ALT test cases.

| Subjects | 1-way | 2-way | 3-way | 4-way | 5-way | 6-way | 7-way |
|---|---|---|---|---|---|---|---|
| FreeMind | 1.85 | 1,293.53 | 58.78 | 19.30 | 0.08 | 0.02 | – |
| GanttProject | 0.81 | 291.68 | 12.86 | 15.02 | 1.40 | 0.17 | 0.02 |
| jEdit | 1.13 | 662.52 | 32.67 | 5.04 | 3.79 | 0.92 | – |
| OmegaT | 0.49 | 143.32 | 9.16 | 5.97 | – | – | – |

The space requirement of the *FindState*$(S_0, E_i)$ function used by ALT were quite low. Table 5 shows the number of entries needed. As expected, the number of entries is largest for 2-way tests and goes down with higher way test case. This is simply because of the smaller number of higher-way tests.

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 6. 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 was 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 1–3 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 that executed in very little time (Table 7). This helps to partly answer Q2.

## 6.2. Discussion

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 determined by alternating between test execution and generation.

Fault 4 in GanttProject results in a NumberFormatException. It is detected by a 3-way test case $\langle e_1$: *Create new task*; $e_2$: *Set general task property*; $e_3$: *Set non-integer value in task duration*$\rangle$. We now examine the source code and explain why ALT was able to successfully generate a test case that resulted in a crash.

Let's examine the code for $e_1$'s event handler, which creates a new task and sets the selected index (using `setSelectedIndex`) to `UIFacade.GANTT_INDEX`:

```
public Task newTask () {
  getTabs().setSelectedIndex(UIFacade.GANTT_INDEX);
  ...
}
```

Event $e_2$ is associated with a "tab" widget `bProperties`; this widget is registered to a listener that handles state changes in the following method:

```
public void stateChanged (ChangeEvent e) {
  ...
  bProperties.setEnabled (
    getTabs ().getSelectedIndex () ==
  UIFacade.GANTT_INDEX||
    getTabs ().getSelectedIndex () ==
  UIFacade.RESOURCES_INDEX);
  ...
}
```

Note that $e_1$'s execution sets the selected index (using `setSelectedIndex`) to `UIFacade.GAN- TT_INDEX` and the above listener uses this value in a conditional statement to enable the `bProperties` tab. Hence, in the GUI, event $e_1$ enables $e_2$; event $e_2$ is disabled otherwise. In the first iteration of ALT, all 2-way covering test cases that started with $e_2$ remained unexecuted. However, the test case $\langle e_1; e_2\rangle$ 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 $\langle e_1; e_2; e_3\rangle$.

Once enabled, event $e_2$ shows the general task property, using the code:

```
public TaskPropertiesAction (IGanttProject
project, TaskSelectionManager selectionManager,
  UIFacade uiFacade) {
  GanttTask [] tasks = new GanttTask [] (GanttTask)
  selection.get(0);
  GanttDialogProperties pd = new
  GanttDialogProperties (tasks);
  // show different task properties tab
  pd.show (getTaskManager (), myHumanManager,
  myRoleManager, getUIFacade ());
  ...
}
```

Event $e_3$ is the action of entering text in the JTextField `durationField1` contained in the JPanel named TaskPropertiesBean which is the GUI for task properties setting dialog. Hence, event $e_3$ remains unavailable until event $e_2$, which brings up the dialog containing $e_3$, is executed. Because $e_3$ is disabled during the first iteration, as for the case with $e_2$, none of the 2-way test cases that started with $e_3$ executed.

In the first iteration of ALT, all 2-way covering test cases that started with $e_3$ remained unexecuted. Moreover, $\langle e_2; e_3\rangle$ 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 $\langle e_1; e_2; e_3\rangle$ was executed, it was used to determine that $\langle e_1; e_2\rangle$ enables $e_3$. Lines 12–14 used this information to obtain new test cases for the third iteration.

Once $e_3$ is executed, i.e., text is entered, the test case closes the dialog via an `Ok` button, which causes the execution of the following code:

```
new OkAction () {
  public void actionPerformed (ActionEvent arg0) {
    uiFacade.getUndoManager ().undoableEdit
  (''Properties changed'',
    new Runnable () {
      public void run () {
        Task [] returnTask =
  taskPropertiesBean.getReturnTask ();
      }
    });
  }
}
```

The invocation `taskPropertiesBean.getReturnTask ()` above in turn invokes getLength ():

```
public Task [] getReturnTask () {
  ...
  if (getLength () > 0) {
    mutator.setDuration (returnTask [i].getManager
  ().createLength (getLength ())
);
  }
  ...
}
```

And the code for getLength () is:

```
public int getLength () {
  length = Integer.parseInt (durationField1.getText
  ().trim ());
  return length;
}
```

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 (duration- Field1. getText ().trim ())` which throws a NumberFormatException.

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 $\langle e_1$: *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*$\rangle$. Once again, the enabling relationship is complex – $e_1$ enables $e_5, \langle e_1; e_4\rangle$ enabled $e_5, \langle e_1; e_4; e_5\rangle$ 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 $\langle$ $e_1$: *Download QuickNotepad plugin*; $e_2$: *Select QuickNotepad plugin*; $e_3$: *Install QuickNotepad plugin*; $e_4$: *Choose QuickNotePad file*$\rangle$. 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 $e_2, \langle e_1; e_2\rangle$ enabled $e_3$; hence the $\langle e_1; e_2; e_3\rangle$ part of the test case was generated by Lines 12–14 of the ALT algorithm. Finally, $\langle e_1; e_2; e_3\rangle \overset{5(2)}{\rightarrow} e_4$; 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.

## 7. 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 `GanttDialogPro-perties`; $\{e_5, e_6, e_7\}$ were in `GanttTreeTable`. Similarly, for jEdit, $e_1$ was in the `PluginMana-ger` 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. In terms of threats to external validity, 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 other GUI applications that have complex underlying business logic and a fairly simple GUI. Also, all the subject applications are open-source, typically developed by volunteer developers and might be more bug-prone than software implemented by paid developers. In terms of threats to internal validity, the instruments used for run-time state collection of GUI widgets was based on 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.

## 8. Conclusions and future work

This paper presented a new alternating technique to generate n-way covering test cases. The key contributions of the paper are (1) the formal definition of the ESI relationship with examples and (2) the iterative refinement of test cases by generating them in batches. Analysis of the run-time state of GUI widgets obtained from a previous test batch are used 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 test cases are executed; their results are used for analysis and further test-case generation; this process is repeated, iteratively yields test cases. 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 4, we will examine aspects of the ESI relationship that helps to improve the reachability of fault-revealing code. 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.

In the medium term, we will leverage other researchers' work to further study GUI faults and failures. 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. [13], 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 [9]. Although traditionally used for finite-state machines, we feel that it may be extended to our flow-graphs for GUI.

In the long term, we will extend this work to non-GUI domains. Some of our earlier work based on the ESIs has already 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.

## 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, F. Ricca, State-based testing of Ajax web applications, in: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, April 9–11, 2008, pp. 121–130.
[2] C. Boyapati, S. Khurshid, D. Marinov, Korat: automated testing based on Java predicates, in: Proceedings of International Symposium on Software Testing and Analysis (ISSTA '02), 2002, pp. 123–133.
[3] M.B. Dwyer, V. Carr, L. Hines, Model checking graphical user interfaces using abstractions, in: M. Jazayeri, H. Schauer (Eds.), Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering: ESEC/FSE '97, Lecture Notes in Computer Science, vol. 1301, Springer/ACM Press, 1997, pp. 244–261.
[4] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering and Methodology 5 (1) (1996) 63–86.
[5] M. Finsterwalder, Automating acceptance tests for GUI applications in an extreme programming environment, in: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, May 2001, pp. 114–117.
[6] M.J. Gallagher, V.L. Narasimhan, Adtest: a test data generation suite for Ada software systems, IEEE Transactions on Software Engineering 23 (8) (1997) 473–484.
[7] N. Gupta, A.P. Mathur, M.L. Soffa, Automated test data generation using an iterative relaxation method, in: Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering: FSE-6, 1998, pp. 231–244.
[8] J.H. Hicinbothom, W.W. Zachary, A tool for automatically generating transcripts of human–computer interaction, in: Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting, Special Sessions: Demonstrations, vol. 2, 1993, p. 1042.
[9] R.M. Hierons, H. Ural, Optimizing the length of checking sequences, IEEE Transactions on Computers 55 (5) (2006) 618–629.
[10] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM SIGPLAN Notices 39 (12) (2004) 92–106.
[11] JUnit, Testing Resources for Extreme Programming. <http://junit.org/news/extension/gui/index.htm>.
[12] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[13] W. Masri, A. Podgurski, D. Leon, An empirical study of test case filtering techniques based on exercising information flows, IEEE Transactions on Software Engineering 33 (7) (2007) 454–477.
[14] P. McMinn, Search-based software test data generation: a survey: research articles, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[15] P. McMinn, M. Harman, D. Binkley, P. Tonella, The species per path approach to search-based test data generation, in: Proceedings of International Symposium on Software Testing and Analysis (ISSTA '06), 2006, pp. 13–24.
[16] A.M. Memon, A Comprehensive Framework for Testing Graphical User Interfaces, Ph.D. Thesis, Department of Computer Science, University of Pittsburgh, July 2001.
[17] A.M. Memon, Automatically repairing event sequence-based GUI test suites for regression testing, ACM Transactions on Software Engineering and Methodology 18 (2) (2008) 1–36.

[18] A.M. Memon, I. Banerjee, A. Nagarajan, GUI ripping: reverse engineering of graphical user interfaces for testing, in: Proceedings of The 10th Working Conference on Reverse Engineering, November 2003.
[19] A.M. Memon, M.E. Pollack, M.L. Soffa, Hierarchical GUI test case generation using automated planning, IEEE Transactions on Software Engineering 27 (2) (2001) 144–155.
[20] A.M. Memon, Q. Xie, Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software, IEEE Transactions on Software Engineering 31 (10) (2005) 884–896.
[21] C.C. Michael, G. McGraw, M. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27 (12) (2001) 1085–1110.
[22] W. Miller, D.L. Spooner, Automatic generation of floating-point test data, IEEE Transactions on Software Engineering 2 (3) (1976) 223–226.
[23] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th International Conference on Software Engineering (ICSE '07), May 23–25, 2007, pp. 396–405.
[24] A. Rountev, S. Kagan, M. Gibas, Evaluating the imprecision of static analysis, in: Workshop on Program Analysis for Software Tools and Engineering, 2004, pp. 14–16.
[25] R.K. Shehady, D.P. Siewiorek, A method to automate user interface testing using variable finite state machines, in: Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97), Washington/Brussels/Tokyo, June 1997, IEEE Press, pp. 80–88.
[26] L. White, H. Almezen, Generating test cases for GUI responsibilities using complete interaction sequences, in: Proceedings of the International Symposium on Software Reliability Engineering, October 2000, pp. 110–121.
[27] L. White, H. AlMezen, N. Alzeidi, User-based testing of GUI sequences and their interactions, in: Proceedings of the 12th International Symposium Software Reliability Engineering, 2001, pp. 54–63.
[28] L. White, H. Almezen, S. Sastry, Firewall regression testing of GUI sequences and their interactions, in: Proceedings of the International Conference on Software Maintenance, The Netherlands, September 22–26, 2003, pp. 398–409.
[29] T. Xie, D. Notkin, Mutually enhancing test generation and specification inference, in: A. Petrenko, A. Ulrich (Eds.), Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Lecture Notes in Computer Science, vol. 2931, Springer, 2003, pp. 60–69.
[30] X. Yuan, A.M. Memon, Using GUI run-time state as feedback to generate test cases, in: Proceedings of the 29th International Conference on Software Engineering (ICSE '07), May 23–25, 2007, pp. 396–405.