Pushing the Limits on Automation in GUI Regression Testing

Zebao Gao
Department of Computer Science
University of Maryland, College Park
Email: gaozebao@cs.umd.edu

Chunrong Fang
State Key Laboratory for
Novel Software Technology
Nanjing University, China
Email: chrong.fang@gmail.com

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

***Abstract*—Although there has been much work on automated GUI regression testing of software, full automation continues to elude us. There are two significant impediments to full automation: obtaining (1) test inputs and (2) test oracle. We now push the envelope on full automation of GUI regression testing by fully automatically generating test cases as well as the test oracle, completely eliminating manual work. This allows us to study issues of false positives/negatives in test failure; we provide ways to minimize these. The results of our empirical studies suggest that our approach of using workflow-based test cases, derived from the software under test, may help empower the end user to perform regression testing before applying software updates.**

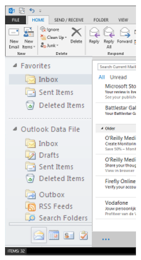
Prologue

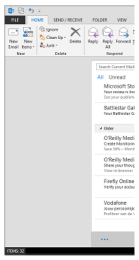
System administrator *Bob* receives a patch p for a heavily used software package A. *Bob* has neither source code nor test cases for p or A. Before "signing off" on p and installing it, *Bob* needs to ensure that p indeed does what it claims to do (e.g., fix a bug) without breaking existing functionality. Current techniques—in the absence of source code—are extremely slow, *ad hoc*, and incomplete, relying mostly on human testers to manually craft and execute test cases that verify a limited number of critical functions; after verification, certain parts of the patched A may still break when used by end-users.

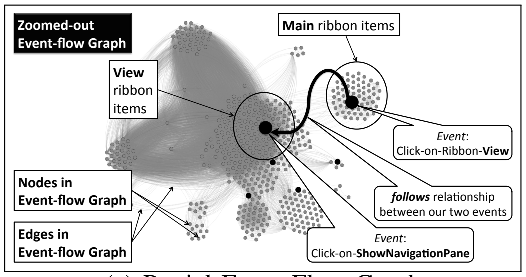
Speed is of the essence when deciding to install patches. An unpatched system contains at least a known bug, which may be encountered by end users. *Bob* must act quickly, however, he is afraid that premature patching without adequate testing may break the system, or worse, introduce new, more serious bugs. *Bob*'s fears are not unfounded. The Internet abounds with reports of applied patches creating new problems. For example, on August 13, 2013 as part of Microsoft's monthly *Patch Tuesday* release, eight patches were released to address 23 bugs. Among numerous other problems, these patches caused corruption of Microsoft Exchange index database; moreover the Active Directory Federation Services stopped working. Microsoft finally recalled the patches.

Fortunately for *Bob*, he uses *testPatch* to test p on A. He quickly creates two clones of A, called A_1 and A_2, sandboxes them individually, and applies p to A_1 to obtain $p(A_1)$. He then launches *testPatch*, which works as follows: it (1) automatically reverse engineers $p(A_1)$ and A_2, without their source code, obtaining workflow models—called event-flow graphs (EFG)—of their user interfaces, (2) automatically generates test cases—each modeled as a sequence of user-interface actions—that (3) automatically *simultaneously* executes on $p(A_1)$ and A_2, in *lockstep* fashion, comparing their outputs after each step for differences. *Bob* checks whether the differences (if any) are consistent with p's documented changes. If yes, then $\underline{p \text{ is safe}}$; else $\underline{A \text{ has regressed due to } p}$. Because *testPatch* uses formal test adequacy criteria based on the executable workflows of A, *Bob* has a certain degree of confidence in his decision.

Example 1: For A = **Outlook 2013**, p = **KB2817630**, a patch released in September 2013; applying p causes A (Figure (a) below) to lose its *Navigation Pane* resulting in a malformed window (b). Microsoft released a new patch to fix this problem. *testPatch* detects this fault because among the test cases it generates is ⟨*Click-on-Ribbon-View, Click-on-ShowNavigationPane*⟩ that mimics a user trying to view the navigation pane. The reverse-engineered EFG, zoomed-out part seen in (c), of this application shows the two events *Click-on-Ribbon-View* and *Click-on-ShowNavigationPane* connected by a *may-follow* relationship that captures the workflow between them. It is this automatically-derived workflow that *testPatch* uses to obtain test cases automatically.


(a) Outlook Unpatched


(b) Outlook Patched


Zoomed-out Event-flow Graph
Main ribbon items
View ribbon items
Event: Click-on-Ribbon-View
Nodes in Event-flow Graph
follows relationship between our two events
Edges in Event-flow Graph
Event: Click-on-ShowNavigationPane
(c) Partial Event-Flow Graph

Example 2: For A = **Outlook 2007**, p = **KB2509470**, a patch released in April 2011; applying p causes the software to report printer problems; to print or generate a print preview from within Outlook, the user would get the error message "*There is a problem with the selected printer. You might need to reinstall this printer. Try again, or use a different printer.*" The resolution is to remove the patch. *testPatch* detects this fault because among the test cases it generates is $\langle$*Click-on-PulldownMenu-File, Click-on-MenuItem-Print*$\rangle$ that mimics a user printing a document.

testPatch uses a *workflow* model, called an EFG, of the software. An EFG contains nodes that represent events (user actions) and directed edges that represent the *may-follow* relationship between events. An edge from n_x to n_y means that event e_y (represented by n_y) can be executed *immediately* after event e_x (represented by n_x); i.e., "e_y *may-follow* e_x." Automatically *walking* (via graph traversal) the EFG yields a sequence of events that can be automatically executed as a test case. Different walking algorithms yield different types of test cases, satisfying various *workflow-based test adequacy criteria*. *testPatch*'s default test-case generator tests *all possible workflows* bounded by length. *testPatch* has a fully automatic test oracle mode that will detect the above two flaws. It also has several user-configurable options that allow specification of probes/hooks to collect specific outputs for analysis/comparison.

I. Introduction

When a software system is changed/updated/modified, it is regression tested to ensure that its existing features have not broken [1]. It is typical to expect system developers and software engineers to perform regression testing in house [2]. For extremely critical systems, one expects additional regression testing be conducted on site once the system has been installed in case some previously hidden bugs surface in the customer's configuration (hardware, settings, environment, workload) of the software [3]. Such a two-stage process is less typical in conventional deployments. System administrators and end users are left to their own devices when an update or patch is available, often finding that an update has broken the software.

Such software *maintenance* use-cases preclude the use of fully automatic source-code based tools to *regression* test it [4] because the source code is not available. The alternative—manual regression testing—is dauntingly expensive [2]. Consequently, in the absence of automated testing tools, users/system-managers are afraid to make changes to their systems, lest they cause unintended regressions [5]. The absence of tools is due to a fundamental gap between how existing tools operate (at the code level [6]) and how system testers test software systems (at the user-interface level via multiple workflows [7]). In this paper, we bridge this gap, thereby enabling the development of a new generation of regression testing tools that operate at the workflow level. System testers and administrators, as well as end users will be able to use these tools to automatically *regression* test the software systems they manage and use.

Consider the architecture of modern software systems, typically designed to respond to *sequences of events [8]*, e.g., messages (Android Intents[1], API method calls), user actions via the user interface, or input signals in an embedded system [9]. Such systems are typically composed of loosely-coupled pieces of code called *event handlers* (e.g., *action listeners*, *callbacks*, or *event listeners/handlers*) and *initialization/glue* code [10]. When such software executes, initialization code sets up the handlers, and the software waits for events to occur. Once an event occurs, the corresponding event handler executes to *handle* the event; when finished, the software goes back to waiting for a new event; and so on. The order of execution of event handlers is largely dictated by the order of input events driven by the software's workflows.

The above rather-simplified-for-brevity architecture creates significant challenges for testing. It creates two distinct levels of abstraction for software: (1) *workflow level* that system testers see and (2) *code level* that developers see. A testing tool that works at the workflow level must have access to some type of *blueprint* of all possible workflows allowed/disallowed by the software's specifications [10]. It can use this blueprint to generate workflows (sequences of events) to test the software. Unfortunately, such blueprints are hardly ever available. On the other hand, a testing tool that works at the code level will attempt to maximize some underlying code-based criterion, such as *statement coverage*, i.e., cover each code statement at least once [6]. The code level is not suitable for modern event-driven systems due to several reasons. First, high statement coverage can be obtained by creating a *contrived* test suite that executes each event only once – this suite will execute each event handler once, covering many program statements. Our past work has shown that faults are not detected when each event is executed in isolation; rather, faults are detected when *different permutations* of events are executed together. Second, many event handlers reside in pre-compiled assemblies—often from different programming languages—for which source code is not available; code based techniques cannot be applied. Finally, many stakeholders—for example, system managers who apply patches to software—do not have access to source code; they cannot employ code-based testing tools [4].

There is a severe need for a new generation of regression testing tools that operate without source code, do not require complex manually created software specifications, and are readily deployable by various software stakeholders, e.g., system administrators [11]. Until such tools are available, the current state of the practice will persist: system administrators will remain too afraid to update/patch their systems; regression testing will remain *ad hoc*, incomplete, and largely manual; regression bugs will remain undetected until they surface in the field; "penetrate-and-patch"-like approaches (test system from the outside, identify a flaw in it, fix the flaw, and then go back to looking) will remain the prevalent practice. This paper takes the first step to change this state of the practice.

We describe *testPatch*, a framework for fully automatic regression testing. *testPatch* does not require source code or any software specifications. It can reverse engineer—fully automatically—formal workflow-based models of the patched and unpatched software, generate test plans from the workflow models, satisfying workflow-based criteria, and execute them automatically. Mismatches indicate regression bugs or intended changes in functionality. We have leveraged much of our previous work in developing *testPatch*. The workflow models are event-flow graphs (EFGs) that model all allowed workflows. We obtain these automatically using *GUI Ripping* [12] and generate test plans using graph-walking algorithms based on

[1]http://developer.android.com/guide/components/intents-filters.html

workflow criteria. The test oracles are based on comparion of states of GUI widgets. The implementation is done with our tool called GUITAR [13].

Even though we have been able to reuse much of our previous infrastructure components, we have never before performed regression testing using real multiple versions of fielded software. Given two versions, S_0 and S_1, of a software, we need to first identify a subset of workflows—from an infinite set—that can reveal regression bugs. We need to execute these on S_0 and S_1 to identify differences. Some of these may not execute because the structure of the GUI may have changed from S_0 to S_1. We need to capture the right elements of the states of S_0 and S_1 to compare, so as to identify errors. Each of these requirements created several research challenges, for which we had to develop novel solutions. First, it was difficult to determine *which workflows to test* in order to reveal regression bugs. In principle, there are an infinite number of workflows that can be executed on a modern GUI-based software; the number grows exponentially with length. Second, it was difficult to map events from one version to the next. Typically, GUI elements do not have unique identifiers. In the face of structural changes to the GUI, it is challenging to determine a correspondence between widgets across versions. For example, if a window has 3 text boxes in version $v0$, and 2 new text boxes were added, and one was deleted, in version $v1$, then it is impossible to automatically determine the mapping between text boxes from $v0$ to $v1$. This creates problems for a text case executor that, while executing a test case, wants to input text in the third text box of $v0$. Correspondingly, it wants to enter the same text in $v1$ (to determine if its behavior changed) but cannot confidently determine which text box to target in $v1$. Third, it was difficult to determine *what to compare* between versions to reveal bugs. One extreme is to *compare everything* in the versions' states. Besides being impractical, this approach leads to false positives as many elements in the state is expected to change. Comparing smaller parts of the state may miss faults.

In addressing these challenges, we report 3 novel contributions: (1) Demonstration that certain workflow-based criteria are adequate with respect to an identified set of bugs. (2) *Signature*-based mapping between versions. (3) Demonstration that certain test oracles provide the best mix of fault detection, and false positives and negatives.

II. Background

We now describe our previous work on automatic GUI testing and test oracles for GUI testing. We start with a description of GUITAR, our automatic GUI Testing frAmewoRk. GUITAR automatically builds a formal model called an Event-Flow Graph (EFG) of the application under test (AUT), generates test cases based on a event coverage criteria, and replays them.

GUITAR adopts a process called GUI Ripping [14] to traverse the available events on the GUI in a depth-first manner. During this process, the GUI ripper extracts GUI structure information, including the hierarchical structure of GUI windows and widgets, as well as their properties (e.g., title, type, position, whether a widget opens a modal/modeless[2] window or a menu). *Restricted-focus events* open *modal windows*, *unrestricted-focus events* open *modeless windows*, and *termination events* close modal windows.

The GUI Ripper then converts the GUI structure to an EFG, which is a directed graph representing all possible event interactions in the GUI. More formally, an *EFG* for a GUI G is a 4-tuple <**V**, **E**, **B**, **I**> where: (1) **V** is a set of vertices representing all the events in G. Each $v \in \mathbf{V}$ represents an event in G; (2) $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. We say that event e_i **may-follow** e_j iff e_j may be performed *immediately* after e_i. An edge $(v_x, v_y) \in \mathbf{E}$ iff the event represented by v_y **may-follow** the event represented by v_x; (3) $\mathbf{B} \subseteq \mathbf{V}$ is a set of vertices representing those events of G that are available to the user when the GUI is first invoked; and (4) $\mathbf{I} \subseteq \mathbf{V}$ is the set of restricted-focus events of the GUI.

That is, in an EFG, each vertex represents an event on the GUI (e.g., click-on-File, click-on-Open), and an edge represents a *may-follow* relationship between two events. Note that only events inside the modal window invoked will follow a restricted-focus event. All events in the new invoked window as well as the original invoking window will follow an unrestricted-focus event. All events in the window from which the current modal window is invoked will follow a termination event. In the example shown in Figure 1, a *may-follow* edge from event *Copy* to another event *Paste* means the latter event *may* be performed immediately after the former event.

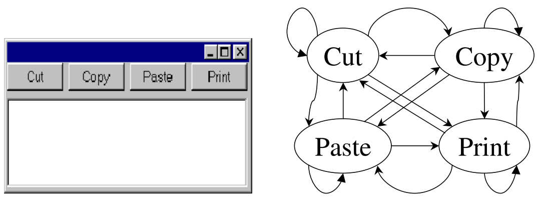


Fig. 1. Example GUI and its EFG

Each test case generated is a sequence of events from the EFG. More formally, a *test case* is $<e_1, e_2, e_3, \ldots, e_n>$ where $(e_i, e_{i+1}) \in \mathbf{E}$, $1 \leq i \leq n-1$. Notice that each test case will need to start from an event available at the initial state of the AUT, thus *reaching events* may be prepended to the test case to make it executable. Event coverage criteria are used to generate GUI test cases. For example, a test suite, T_E, can be generated to cover all events in the EFG. Consider a test suite $T = \{t_1 = <Copy>, t_2 = <Cut, Paste>, t_3 = <Cut, Print>\}$ that covers all events in the GUI. A test suite, T_D, can also be generated in a similar manner to cover all edges in the EFG. GUITAR executes the test cases one by one from the same initial state of the application, capturing the GUI's state after each event. This state is used to create the *test oracle*, a mechanism used to determine whether a test case passed or failed.

We use the GUI state for the test oracle because any part of the GUI state through the sequence of the test execution may be potentially bug revealing. Xie and Memon present automatic test oracles on applications with manually seeded bugs [15]. More specifically, they conduct a thorough study on two important research questions: (1) what to assert, and (2) how frequently to check an assertion. To answer their research

[2]Standard GUI terminology; see detailed explanations at msdn.microsoft.com/library/en-us/vbcon/html/vbtskdisplayingmodelessform.asp and documents.wolfram.com/v4/AddOns/JLink/1.2.7.3.html.

questions, Xie and Memon [15] propose six different instances of test oracles, namely, L1, L2, ..., and L6.

L1 Compare the properties of the GUI widgets associated with the event after execution of each event.
L2 Compare the properties of all GUI widgets in the current active window after execution of each event.
L3 Compare the properties of all GUI widgets of all windows after execution of each event.
L4-6 After the last event of the test case, compare the properties of widgets associated with the event, widgets in current active window, or in all windows, respectively.

Their study shows that a comparing all properties at the end of test case execution provides the most effective test oracle.

III. Our Approach

We now discuss the design of *testPatch*. There are several key components in *testPatch*, including automatic test generation, matching, execution and oracle. We utilize our test harness, GUITAR, for test generation; we discuss matching, execution and oracle.

A. Automatic Matching

We intend to execute test cases developed for version V_{X-1} on version V_X. Some differences between the versions may make test cases from V_{X-1} unexecutable on V_X. For example, we have modified the GUI of Figure 1 to obtain the one shown in Figure 2. Of the 3 test cases we showed earlier, none of t_1, t_2, or t_3 are executable. Each test case requires the reaching event $Edit$ to be performed first. Moreover, instead of being instances of the *JButton* class, all 3 widgets *Cut*, *Copy*, and *Paste* are now instances of *JMenuItem*. A test harness that considers the widget's class to locate it will fail.

We use two mechanisms to promote reusability of test cases between the versions. First, we ignore all structural reaching events, focusing instead on system interaction events. Second, we use a *signature* to identify each event. We now discuss these mechanisms.

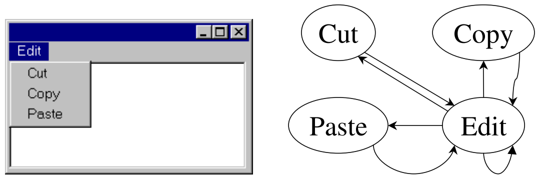


Fig. 2. A Regression Testing Example.

We classify GUI events into groups of *reaching events* that perform only structural GUI operations such as opening menus, and opening and closing windows. The rest are *system interactive events* which invoke underlying business logic. We map only system interactive events from V_{X-1} to V_X to reduce the risk of test case infeasibility. We can identify these event types during GUI ripping.

Our *signature* mechanism is based on using the state (set of property-value pairs) of the widget for its identification. A naive approach to checking whether a widget w_a in V_{x-1} is the same as widget w_b in V_x is by using the expression:

$$match(w_a, w_b) \leftarrow (v_{a1} == v_{b1}) \& \ldots \& (v_{aN} == v_{bN}) \quad (1)$$

where v_{a1}, v_{a2}, ..., v_{aN} are property values representing widget w_a's state, and v_{b1}, v_{b2}, ..., v_{bN} are property values representing widget w_b's state.

However, we cannot blindly use all the elements of the state because it will contain some property values that change during the GUI's execution, and play little role in identifying that widget. For example, the value of the *text* property for a *JTextField* object will change when the text changes; the *enabled* property changes when the object is enabled/disabled. Other properties are *more likely* to persist across executions, e.g., the *accessibility label (content description)*. Hence, certain properties are better candidates for matching than others. This observation leads to the following matching definition we used in this research.

$$match(w_a, w_b) \leftarrow \phi_1(v_{a1} == v_{b1}) + \ldots + \phi_N(v_{aN} == v_{bN}) \quad (2)$$

i.e., instead of a boolean match, we associate a weight ϕ with each property. Properties that are more likely to mislead us are given low weight (in some cases 0). Other properties that are relatively more stable are given higher weight. Then for a specific widget from V_{X-1}, we calculate the weighted distances of all widgets from V_X to figure out the "closest" widget as its corresponding widget.

The only remaining issue is how we obtain the weight values ϕ. For this we leverage our prior work [16], wherein we designed a large empirical study to evaluate for GUI applications what extent of variation is seen in the widget property values. We developed an entropy based metric to quantify the stability of test runs. In this paper, we reuse the entropy-based metric but on property values. A higher entropy value indicates greater variation, and hence, lower stability. Such widget properties are given lower weight in our formula.

In our empirical study of this paper, we present Tables VI with weights of different properties. For example, the *title* of a widget is very unlikely to change across versions. In contrast the *index* property may change when new widgets are added (e.g., the *index* of a menu item will change when another menu item is inserted into the same menu list before it). Thus we will give a higher weight to the *title* property than *index*.

B. Automatic Test Execution

Having the matching between widgets, we can then transform each test case of V_{X-1} to execute on V_X. For the purpose of automatic GUI testing, we extended our test harness so it can execute test cases on the two program versions (unpatched and patched) in lock-step manner and collect output for comparison. Specifically, our test harness will record all available runtime GUI structure information of the application (i.e., all properties of all GUI widgets of all windows after the execution of each event) to have a strong automatic test oracle based on GUI states.

C. *Automatic Test Oracle*

Our previous study on automatic GUI oracles used only one version of the AUT; the "modified" version was simulated by seeding an artificial fault [15]. Mismatches detected between these versions' execution were due to the seeded fault. However, in a real regression scenario, such as ours in this paper, the modified version contains multiple changes. Mismatches during test execution may be due to one of two reasons: (1) bugs and (2) changes in functionality. Unlike our previous work, all mismatches cannot be simply regarded as faults.

We will determine what information to collect for each application, balancing three goals: (1) important differences are not missed, (2) spurious differences (false positives) are minimized, and (3) the overall approach remains practical.

First of all, we need to decide when to capture GUI states. Based on our observations on where the bug-revealing widgets lie, we provide following three options:

ALW	Properties associated with the Window that is Active at the Last step (ALW) of the test execution;
LW	Properties associated with all the Window at the Last step (LW) of the test execution;
AW	Properties associated with All Windows at all steps (AW) of the test execution.

The ALW oracle is the most concise one and thus generates the fewest false positives but it may also miss more bugs compared to the latter two oracles which check a bigger set of GUI properties. The AW oracle will check significantly more GUI states especially when the length of test cases and the number of windows opened in the test execution is big. Thus it may have limited practical value because it may yield too many false positives.

Secondly, as mentioned in test matching, some common patterns of GUI properties are more likely to change across versions and observed to be potential reasons for false positives [16]. Thus we provide one more degree of freedom in our automatic oracle with two options:

SP	A Subset of Properties after filtering out common patterns of spurious properties.
AP	All Properties of GUI widgets that can be extracted by our test harness.

Here we provide some examples of spurious properties. In GUI applications, the position (*x-coordinate*, *y-coordinates*) and size (*height*, *width*) of a window may change often based on the context when the window in redrawn. Also, the relative position of GUI widgets inside a window often changes when the window is resized. When we compare between versions when the application evolves, it is even more likely that there are minor or big mismatches in the positions and sizes of windows and widgets. Thus GUI properties related with the position or size are included in our list of ignored properties. In our empirical study, we show that it may cause one bug to be missed which is related with size of GUI widget. But this happens very rarely and helps avoid a great number of false positives. There are some other common spurious patterns of properties, for example, by using a regular expression, we ignore the string which is often used to show the current time at the status bar at the bottom of some GUI applications.

Finally, when our GUI oracles in two independent dimensions are combined, we obtain 6 different oracles in total, namely, SP-ALW, SP-LW, SP-AW, AP-ALW, AP-LW and AP-AW. We will evaluate all of them in our empirical study.

IV. Experiment

We pose the following research questions regarding our approach of *testPatch*:

RQ1: How effective are workflow-based test plans, derived from event-flow graphs, together with GUI-state-based reference testing oracles in detecting regression bugs in fielded systems?

RQ2: What settings of the GUI-state-based reference testing oracle are most effective?

RQ3: Do workflow-based test plans, derived from event-flow graphs, detect more bugs than fuzzing in the same number of test steps?

A. *Metrics*

RQ1 addresses *effectiveness* of our overall approach. We measure effectiveness in terms of reduction of the *window of exposure* (γ) of a bug. We define γ as the time for which a *regression* bug has existed in the software. We first determine two versions of the software (1) V_X, the version in which the bug first appeared and (2) V_{X+1}, the version in which the bug was fixed. We then compute γ as the time that elapsed between V_X and V_{X+1}. As part of addressing this question, we also discuss real regression bugs that we detected in fielded systems.

RQ2 addresses the *effectiveness* of our test oracles. Because the test oracle determines whether a test case has passed or failed, the effectiveness of a test oracle has to do with coming up with the right determination, while minimizing manual work. If a test oracle fails a test case in the absence of a bug, then we have a false positive; if it passes a test case when it should in fact have failed, we have a false negative. Hence we compare the effectiveness of the six test oracles in terms of bugs, and false positives and negatives.

In practical terms, false positives increase the number of (incorrectly) failed test cases that are output from a tool, requiring manual work to weed them out. A developer/tester will need to go through the list of reported failed test cases, determine why each failed by examining the mismatches reported by the test oracle. The nature of our test oracle—one that compares widget properties—leads to mismatches in terms of mismatched widget properties. Hence, to quantify false positives, we count the number of mismatched widgets that are not related to the bug in question. More specifically, we define False Positives (FPs) for a given bug and test oracle as follows:

Definition 1 (FP(oracle, bug)): The false positive measure for a given oracle and bug is the total number of mismatched widgets minus the number of widgets that *should* have mismatched due to the bug.

Hence, for a given oracle, the average false positives rate per bug is computed as follows:

TABLE I. BUGS IN JEDIT, JMOL AND JABREF

AUT	Bug Id	Bug Description	V_{X-1}	V_X	V_{X+1}
jEdit	1324	CheckboxMissing	4.1-pre4	4.1-pre5	4.1-pre6
	3538	BeanShellError	4.3.3	4.4-pre1	4.4.1
	3645	DropdownlistUnenabled	4.4.2	4.5.1	5.0-pre1
	3899	DropdownlistEmpty	4.3.3	4.4-pre1	5.0-pre1
Jmol	T1	LogoInNewWindow	11.6.27	11.7.1	12.0.38
	T2	NewWindow	13.1.3	13.1.4	13.1.14
	T3	LogoInAboutWindow	12.2.34	13.0.1	14.2.11
	T4	MainWindowTitle	12.2.34	13.0.1	NA
JabRef	65	HelpContent	1.1	1.2	1.3.1
	160	SearchColumn	1.5	1.6-beta	1.7-beta2
	1130	CloseDatabase	1.7.1	1.8-beta	2.0-beta
	1132	SaveDatabase	1.7.1	1.8-beta	2.0-beta
	1133	Search	1.7.1	1.8-beta	2.0-beta
	1134	Export	1.7.1	1.8-beta	2.0-beta
	1135	EditEntry	1.7.1	1.8-beta	2.0-beta
	1136	SearchPanel	1.7.1	1.8-beta	2.0-beta

Definition 2: $$FP(oracle) = \frac{\sum_{bugs} FP(oracle, bug)}{\#bugs}$$

The metric for false negatives is simpler as a real fault is missed, the tester does not expend effort. We can simply count the number of faults missed when using a certain oracle.

We consider false positives as the most relevant measure because the entire list of mismatches will need to be manually examined by a tester/developer. Too many false mismatches (false positives) will degrade the usability of our approach.

For RQ3, we measure the number of bugs detected by a fuzzing tool called the *Monkey*.

B. Subjects of Study & Bugs

Three Java applications under test (AUT) are used in our empirical study: jEdit,[3] Jmol,[4] and JabRef.[5] jEdit is a text editor for programmers. Jmol is a molecular viewer for three-dimensional chemical structures. And JabRef is a bibliography reference manager. We select 16 reported regression bugs from these applications' bug reporting sites (usually SourceForge[6]). Our choice of bugs was driven only by their manifestation as failures on the GUI. Table I shows the bugs and the versions in which they first appeared and fixed. For each bug, we denote the version in which the bug first appeared as V_X, the past version in which the broken feature still worked as V_{X-1} and the future version in which the bug was first fixed as V_{X+1}. We also use a concise, descriptive keyword to describe each bug. As the keywords indicate, these bugs are all reflected on the GUI.

C. Addressing RQ1

We start by visually showing, in Figure 3, the window of exposure of each of our bugs under consideration. The x-axis shows clock time, in years, increasing from left to right, and the corresponding version numbers. The y-axis shows the individual bugs. Each bug is represented by a horizontal bar that starts at the time the bug was first introduced (not detected) into the software and ends when the bug was removed. The length of the bar indicates the window of exposure. We see that our selected bugs' window of exposure varies across

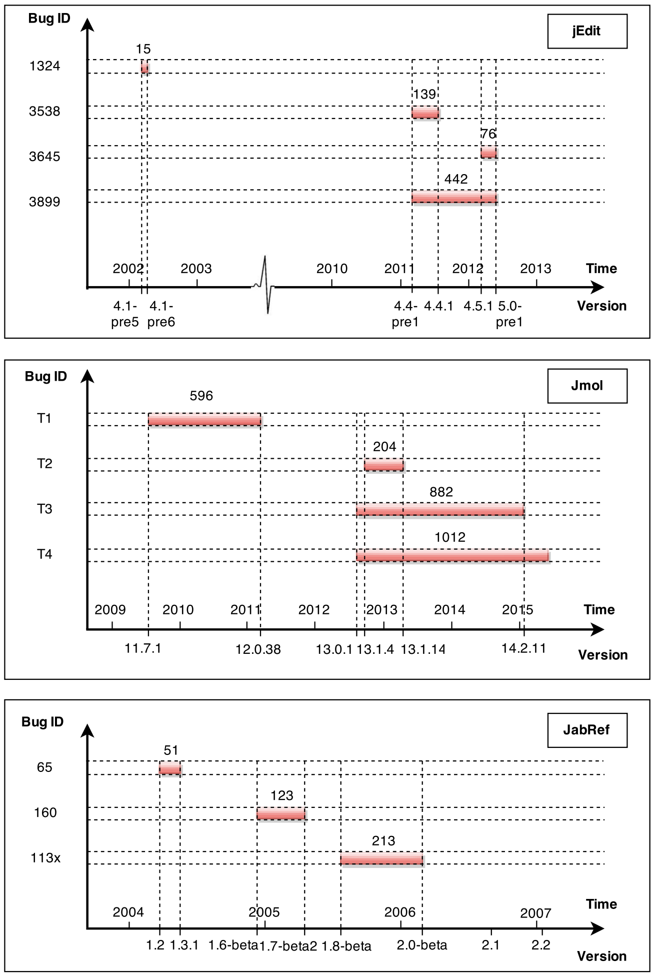


Fig. 3. Window of Exposure for Bugs in Our Study

TABLE II. TEST CASE GENERATION BASED ON COVERAGE CRITERIA

Application	Windows	$\lvert T_E \rvert$	$\lvert events(T_E) \rvert$	$\lvert T_D \rvert$	$\lvert events(T_D) \rvert$
JE4.1Pre4	22	426	556	11650	26410
JE4.3.3	34	780	1008	22749	51433
JE4.4.2	18	663	780	19439	42415
JM11.6.27	17	278	488	4536	12776
JM12.2.34	22	354	753	7202	22131
JM13.1.3	25	445	1154	10636	40581
JR1.1	12	213	324	3601	8878
JR1.5	20	537	989	12697	36591
JR1.7.1	23	631	1063	27923	79237

applications and bugs. Most bugs in JabRef and jEdit persisted for months whereas most from Jmol persisted across years. For example, Bugs T3 and T4 of Jmol, and Bug 113x of JabRef persisted across major versions. Other bugs only persisted across multiple minor versions. Bug T4 of Jmol remained unresolved by the writing of this paper, and we denote its V_{X+1} as NA.

As discussed in Section III, our algorithm underlying *testPatch* has two tunable parameters: (1) the nature of workflow-based test plans in terms of how they cover the event-flow graphs and (2) the test oracle. For the former, we generated our test plans to satisfy event coverage and edge coverage criteria. These test plans are used to generate test suites that we call T_E and T_D, for event and edge coverage, respectively.

[3]http://sourceforge.net/p/jedit/
[4]http://sourceforge.net/p/jmol/
[5]http://sourceforge.net/projects/jabref/
[6]http://sourceforge.net/

TABLE III. JEDIT BUGS

Bug Id	$\lvert TC \rvert$	Steps	TC Length	Oracle
1324	303	1. Click on the tool bar icon "Buffer options"	1	A checkbox and a text label "Indent line when Enter key is pressed" are shown in V_{X-1} but missed in V_X.
3538	30	1. Click on menu "View"; 2. Select the menu item "Global Scope" under "Buffer Sets".	2	A dialog "BeanShell Error" is shown in V_X.
3645	41	1. Click on tool bar icon "Global options"; 2. Click on menu tree item "Status Bar"; 3. Click on tab "Widgets"; 4. Click on button "+".	4	The "Enabled" property of the JComboBox "Choose a Widget" is True in V_{X-1} but false in V_X.
3899	41	1. Click on tool bar icon "Global options"; 2. Click on menu tree item "Status Bar"; 3. Click on tab "Widgets"; 4. Click on button "+".	4	The JComboBox "Choose a Widget" is empty in V_X, whereas an element, a BasicComboBoxRenderer titled "lineSep", is shown in V_{X-1}.

TABLE IV. JMOL BUGS

Bug Id	$\lvert TC \rvert$	Steps	TC Length	Oracle
T1	19	1. Click on menu "Help"; 2. Click on menu item "What's new".	2	The logo of Jmol is shown in V_{X-1} but not shown in V_X in dialog "What's new".
T2	19	1. Click on toolbar icon "new" to create a new file.	1	A new file is created and shown in a new window in V_{X-1}, but nothing is shown in V_X.
T3	23	1. Click on menu "Help"; 2. Click on menu item "About Jmol".	2	The logo of Jmol is shown in V_{X-1} but not shown in V_X in dialog "About Jmol".
T4	20	1. Click on any widget in the main window.	1	The main window title is "Jmol" in V_{X-1}, but is some random string in V_X.

TABLE V. JABREF BUGS

Bug Id	$\lvert TC \rvert$	Steps	TC Length	Oracle
65	369	1. Click on menu "Options"; 2. Click on menu item "Customize entry types"; 3.Click on the help icon in the new window.	3	The widget "JabRef Help" contains help content in V_{X-1} whereas is empty in V_X.
160	24	1. Click on toolbar icon "new" to create a new database.	1	The search column is not wide enough nor expandable in V_X.
1130	490	1. Click on toolbar icon "new" to create a new database.	1	The "Enabled" property of the menu item "Close database" has a value true in V_{X-1} but a value false in V_X.
1132	490	1. Click on toolbar icon "new" to create a new database.	1	The "Enabled" property of the menu item "Save database" has a value true in V_{X-1} but a value false in V_X.
1133	490	1. Click on toolbar icon "new" to create a new database.	1	The "Enabled" property of the menu item "Search" has a value true in V_{X-1} but a value false in V_X.
1134	490	1. Click on toolbar icon "new" to create a new database.	1	The "Enabled" property of the menu item "Export" has a value true in V_{X-1} but a value false in V_X.
1135	490	1. Click on toolbar icon "new" to create a new database.	1	The "Enabled" property of the menu item "Edit Entry" has a value true in V_{X-1} but a value false in V_X.
1136	490	1. Click on toolbar icon "new" to create a new database.	1	The search panel on the left side of the main window is shown in V_{X-1} but missing in V_X.

Table II shows the fundamental characteristics of these suites, including their sizes ($|T_E|$ and $|T_D|$) and the number of events (steps) they execute ($|events(T_E)|$ and $|events(T_D)|$). For the test oracle, we used "*Subset of properties of all widgets in all windows extant in the GUI after execution of the last event in the test case*" (SP-LW). We excluded certain properties that we know *a priori* will vary between runs: *ID* (a dynamic hash code generated from a set of properties), *x-coordinate*, *y-coordinate*, *width* and *height*.

Having configured our algorithms, for each bug, we took its respective pair of versions (V_{X-1}, V_X) generated test plans to satisfy the coverage criteria, and executed the resulting test suites on the pair in lockstep manner. During the execution of each test case, *testPatch* captured the GUI states, including properties of all widgets in all the windows after each step of the test case. Then, as determined by our test oracle, a subset of this captured state was used to determine whether a test case passed or failed. Because of the nature of our reference testing oracle, a mismatch in the state is a failure of the test case; otherwise it passes.

The second column in Tables III through V shows the total number of test cases that detected the respective bug. We see that 3 bugs were detected by hundreds of test cases; the remaining were detected by numbers much beyond our expectation given that these bugs have been in the systems for months to years. For each bug, we also show a sample test case that automatically revealed the bug as well as the cause of this revelation, i.e., the properties that mismatched, and hence triggered the test oracle to report a failure. The columns "Steps" and "TC length" show the steps and length of the test cases. The "Oracle" column shows the mismatch.

As can be seen, all the 16 bugs are in fact detectable by using GUI state. In our case, we missed Bug T1 of Jmol and Bug 160 of JabRef. This is because T1 requires us to compare logo images, a capability that we currently do not have in *testPatch*. Bug 160 requires that we examine a property that allows us to determine whether a column is expandable or not; we currently do not know how to do this. Hence, we missed these bugs because of our current inability to examine certain aspects of the GUI state. We consider this weakness a limitation of our current implementation, not of the overall approach.

This study showed us that with *testPatch* we can actually reveal almost all of our selected bugs at the very version when the bugs are first introduced, thereby making the window of exposure 0. This answers RQ1.

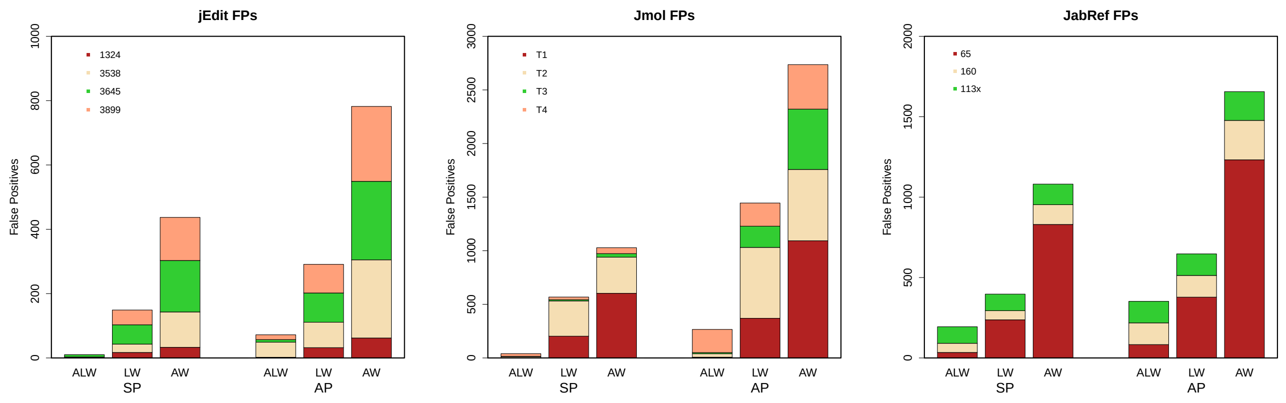


Fig. 4. False Positives of Bug Revealing Test Cases of 3 Applications Using Different Oracles

TABLE VI. SELECTED PROPERTIES OF LAST ACTIVE WINDOW – DIFFERENCES AND REASONS OF FPS

jEdit								
Bug Id	Total	Bug Related	isSelected	Class	Icon	Enabled	Foreground	
1324	3	2	1					
3538	22	20		1	1			
3645	11	4	2	1		1	2	1
3899	2	2						

JMol											
Bug Id	Total	Bug Related	isSelected	Class	Icon	Foreground	Text	Index	Background	Font	Other
T1	3	0		2			1				
T2	12	1	1	1		1	2		5	1	
T3	2	1									1
T4	29	4		17	1		2	2			

JabRef												
Bug Id	Total	Bug Related	isSelected	Class	Icon	Text	Index	Background	Font	Color	Accelerator	Other
65	34	0/3*	2						31			1
160	57	0/0†		2	6	2	18			1	3	
1130	105	3‡		2	1	1	19	1				
1132	105	1‡		2	1	1	19	1				
1133	105	3‡		2	1	1	19	1				
1134	105	3‡		2	1	1	19	1				
1135	105	3‡		2	1	1	19	1				
1136	105	2‡		2	1	1	19	1				

∗diff in last step but not active window of last step
† the bug is related to the minor diff of the width of a column
‡ there are totally 80 differences related to the bugs, but for most of the reported bugs, only a few differences are related with them.

D. Addressing RQ2

We start by comparing the FPs for the six oracles. The results are shown in Figure 4. We see that oracle SP-ALW has the lowest number of false positives and oracle AP-AW has the largest number. In fact, for jEdit and Jmol, SP-AWL has very few false positives. We note that these oracles did successfully detect bugs.

To further understand the differences that compose the FPs, we study the GUI properties that contribute to FPs for each bug on each application when applying oracle SP-ALW. The results, shown in Table VI, show that varying sets of GUI properties contribute to FPs in different applications. Columns "Total" and "Bug Related" show the number of all/bug related mismatches. For jEdit, the bug related mismatches make up the majority of all mismatches. The bug 113x is an exception, because the test case is able to report 80 mismatches that are bug related, but for each single bug, there are only a small number of related mismatches. The following columns show the GUI properties that contribute to the non-bug related mismatches. Among them, a few properties, such as *Index*, *Class*, *Font* and *Background* contribute the most.

TABLE VII. FALSE NEGATIVES & TEST ORACLES

Application	Bug ID	SP			AP		
		ALW	LW	AW	ALW	LW	AW
Jmol	T1	1	1	1	1	1	1
JabRef	65	3	0	0	0	0	0
JabRef	160	1	1	1	1	1	1

Next, we consider the FNs of the oracles in Table VII. Only 3 bugs are shown in the table because there are no FNs for the remaining bugs. It is interesting to observe that the oracle SP-AWL lead to more FNs than other oracles on Bug 65 of JabRef. This is because the bug-related difference is included in one of the windows shown after the last step of the test case, but not in the active window. And as mentioned earlier, two bugs, Bug T1 on Jmol and Bug 160 on JabRef, require a stronger set of GUI properties to be examined, thus they report

TABLE VIII. FAULT DETECTION ABILITY OF DIFFERENT TEST GENERATION TECHNIQUES

	1324	3538	3645	3899	T1	T2	T3	T4	65	160	103x
T_E	1	1	1	1	1	1	1	1	1	1	1
R_E	0.02	0.01	0.00	0.00	0.11	0.09	0.03	0.04	0.00	0.03	1.0
T_D	1	1	1	1	1	1	1	1	1	1	1
R_D	0.38	0.01	0.05	0.04	0.74	0.36	0.90	0.92	0.26	0.48	1.00
T_E	19	1	1	1	1	1	3	3	14	1	1
R_E	0.02	0.01	0.00	0.00	0.11	0.10	0.03	0.04	0.00	0.03	15.62
T_D	303	30	41	41	19	19	23	20	369	24	490
R_D	0.49	0.01	0.05	0.05	1.38	0.54	2.38	2.07	0.29	0.61	1240.81

one FN of on all of our 6 oracles.

To conclude, oracle SP-AWL, has relatively fewer FPs and one FNs. This addresses RQ2.

E. Addressing RQ3

For this research question, we executed a fuzzing *Monkey* algorithm that essentially performs a bounded number of random actions on the GUI. In our case, we bounded this by the number of steps (events) that our workflow-based test plans covered. Equivalent to our event-coverage based plans, we obtained R_E suites; for our edge-coverage based plans, we obtained R_D suites. We repeat the random test generation process for 100 times.

We obtain the probability of the random sequence to detect the bug as the ratio of sequences in the 100 sequences that detect the bug. The results are shown in Table VIII. Each column shows one bug and each row show the fault detection ability of one test generation technique. Test suites R_E and R_D are the random test suites that have the same cost as T_E and T_D, respectively, in terms of test steps. Table VIII are separated into two parts: in the upper part, we show the probability of a certain technique to detect the bug. Since the T_E and T_D test suites can detect all bugs we have, they always have a probability value 1. Whereas the randomly generated test sequences may not be able to detect the bug. And for a certain bugs, the random technique is not able to detect the bug after repeating the random process for 100 times. The lower half of the table show the expected times of the bug triggering events to be executed. For most of the bugs, the technique based on event-coverage criteria outperforms the random techniques except the bugs 103x that requires only clicking on one of the initial events of the application. This addresses RQ3.

F. Limitations

In its present form, our implementation and evaluation suffered from a few limitations that we now discuss. First, to make our method more practical, the number of false positive needs to be further decreased. In its present form, the false positives may overwhelm the tester, thereby making our technique less desirable for practical deployment. Second, our weight-based widget matching scheme, to find the closest match, needs to be enhanced to handle the case of ties. Consider the case when widget w_a (e.g., a text box) in version V_{x-1} matches with two widgets, also text boxes, w_b and w_c in version V_x. This may happen if we ignore the coordinates of these widgets, which we usually do because the screen coordinates change frequently across runs, and only consider their type. If w_b and w_c are in different *containers* (e.g., window, panel, fragment), then we are able to use w_a's container to break the tie. However, if w_b and w_c are in the same container, a case we never encountered in our study subjects, we may need to enhance our matching scheme, or require human input. Third, we completely sidestepped the issue of obsolete or unexecutable test cases. Consider our example of Figure 1 and 2, in which *Print* has been deleted. Test case t_3 is no longer executable on the modified software because the test case uses *Print* but the GUI no longer contains that event. Fourth, even though we did not encounter this issue in our work, it is quite possible that our matching scheme yields an incorrect match, i.e., an event from V_{x-1} is incorrectly matched to an event in V_x. Execution of test cases involving these events will likely yield a large number of mismatches in GUI state, which will increase the number of false positives. Finally, we have evaluated only length-2 test cases in our experiment. Longer test cases will likely increase execution cost because the number of test cases will go up.

V. RELATED WORK

Our work attempts to address two classical problems in the context of regression testing for GUI applications: test generation and test oracles, aiming at finding regression faults with workflow-based criteria and GUI properties as oracles.

Regression testing has been extensively investigated during the past decades, especially in minimization, selection and prioritization [17]. These works aim to maximize the value of the accrued test suite by eliminating redundant test cases, identifying relevant the test cases and ordering test cases. In other words, they work to find subsets of regression test suites and they do not change the codes of test cases. For these works, information about the cost of verifying the output observed with the existing test suite may be collected across versions, since regression testing techniques seek to efficiently re-use existing test cases. This can be incorporated. New versions may change some original features, not just adding new features, and some test cases cannot be run against the new versions. Our work aims to make full automated regression testing with test oracles. On the one hand, our techniques can automatically find differences between versions that reveal bugs and transform the previous test suites into ones that can work on the new versions as well as automatically differences between versions that reveal bugs. On the other hand, test oracles can be automatically created by our techniques.

Richardson [18] developed a toolkit called TAOS (Testing with Analysis and Oracle Support) toolkit to provide different levels of test oracle support. In lower levels, developers can write down expected outputs for a test input, specify ranges for variable values, or manually inspect actual outputs. The oracle support we provide is in TAOS' lower levels: generating expected outputs (widgets) for test inputs. In higher levels, developers can use specification languages to specify temporal properties. There exist a number of proposed approaches for providing oracle supports based on different types of specifications [19]. Different from these specification-based oracle supports, we can enhance oracle checking only for exposing regression faults without any specifications.

When specifications do not exist, automatic test-generation tools such as JCrasher [20] use program crashes or uncaught exceptions as symptoms of the current program version's faulty behavior. Randoop [21] allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra [22] generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. Eclat [23] can generate assertions based on a model learned from assumed correct executions. EvoSuite[24] uses mutation testing to select a subset of assertions. Both Harrold et al's spectra comparison approach [25] and Xie et al's valuespectra comparison approach [26] focus on exposing regression faults. Program spectra usually capture internal program execution information and these approaches compare program spectra from two program versions in order to expose regression faults. Xie et al. [22]developed an automatic approach and its supporting tool, called Orstra, for augmenting an automatically generated unit-test suite with regression oracle checking. Orstra creates assertions for asserting behavior of the object states, as well as return values of methods. Zaeem et al. [27] applied user-interaction features, which is implicated in a significant fraction of bugs and for which oracles can be constructed, based on their common understanding of how apps behave. We model a GUI state in terms of the widgets that the GUI contains, their properties, and the values of the properties.

Orso and Kennedy [28] developed techniques for capturing and replaying interactions between a selected subsystem (such as a class) and the rest of the application. Orso et al. proposed [29] behavioral regression testing approach by (1) generating test cases that focus on the changed parts of the code, (2) running the generated test cases on both the old and new versions of the code and identifying differences in the tests' outcome, and (3) analyzing the identified differences and presenting them to the developers. Their techniques focus on creating fast, focused unit tests from slow system-wide tests. Different from their work, the test suites for new versions are transformed from the previous versions by GUI model matching, which can make the regression test suites run accurately against the new versions. our technique does not only check the differences between the outcomes, but also checks the differences between GUI models. What's more, our technique has a more accurate oracle mechanism and make a automatic regression testing.

Xie and Memon [30] developed six instances of test oracles for a general purpose as described in Section II Background. In this paper, to fulfill the task finding regression faults, our techniques determines oracle timing depicted as III Our Approach, as well as introduces test matching to oracle mechanism.

VI. Conclusions & Future Work

This paper presented a new way to think about regression testing – fully automatic so as to empower the end user, who does not have access to source code, specifications, and test expertise. Armed only with the previous and new (or patched) releases of the software, we want to give the end user a tool that can be used to fully automatically regression test the new release. As we are only interested in regression bugs, we can use the previous version as a "correct" version as a basis for a test oracle. Any differences between the version is a potential bug. Our approach described was simple: run test cases on both versions and report mismatches. Because we relied on worflow-based coverage of the software, we ensured a uniform coverage of the application, without considering code coverage. We, of course realize that end users will augment our test cases with those derived from their own use cases so as to conduct a thorough end-to-end testing of their own scenarios..

Our experiment showed that we can automatically detect all 16 bugs that are manifested on the GUI. At the same time, this experiment helped us to better understand limitations and quantify the overall risks associated with using a dynamic approach to reference test patched software. Because it is rooted in dynamic analysis (i.e., based on actual code execution), we expect our solution will have some limitations that may hinder wider applicability; for example, *testPatch* will not find a vulnerability unless it executes the program path that manifests the vulnerability during execution. Although our work-flow based criteria are designed to force execution of all paths via workflows in the input space, we may still miss some.

This research has applicability to any changes. It can be applied to any quality assurance activity that involves *reference testing*, i.e., testing an application version with respect to another. Consider for example, a *memory-footprint optimizer* that applies code-level optimizations to an Android app in order to reduce its memory footprint. There exist no tools to automatically determine whether the optimizations had any unintended side-effects that broke the software. A variant of *testPatch* may be used.

The limitations of our work and results that we raised earlier give us directions for future work. First, the impact of the bounded length of test cases would have an impact on our experiment results. We used length 2 test cases. We suspect that longer test cases may lead to increased false positives but need to confirm experimentally in future work. Second, as seen in our study, all the 16 bugs are in fact detectable using GUI state, but 1 is missed due to limitations of our tool implementation. One future work direction is to improve the implementation to better support our method. Third, we need to address the issue of obsolete or unusable test cases, perhaps by using our previous work on test repair [31]. Finally, we need to study the liklihood and impact of incorrect matching on our false positive rate, and devise approaches to minimize incorrect matches.

Acknowledgments

We thank Dr. Yvan Labiche for his valuable feedback and suggestions. This material is based on research sponsored by the National Science Foundation Grant Number 1205501 and by DARPA under agreement number FA8750-14-2-0039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, 1996.

[2] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, pp. 81–86, 1998.

[3] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 75–86.

[4] J. M. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, no. 6, pp. 53–59, 1998.

[5] T. Mens, *Introduction and roadmap: History and challenges of software evolution*. Springer, 2008.

[6] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[7] L. J. White, "Regression testing of gui event interactions," in *Software Maintenance 1996, Proceedings., International Conference on*. IEEE, 1996, pp. 350–358.

[8] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 408–418. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070540

[9] S. Arlt, I. Banerjee, C. Bertolini, A. M. Memon, and M. Schaf, "Grey-box gui testing: Efficient generation of event sequences," *CoRR*, vol. abs/1205.4928, 2012.

[10] B. N. Nguyen and A. Memon, "An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, p. 1, 2014. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=06714448

[11] M. J. Harrold, "Testing: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 61–72.

[12] A. M. Memon and M. L. Soffa, "Regression testing of guis," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 118–127.

[13] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, pp. 1–41, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10515-013-0128-9

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

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

[16] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *The Proceedings of The 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[18] D. J. Richardson, "Taos: Testing with analysis and oracle support," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 1994, pp. 138–153.

[19] L. K. Dillon and Y. Ramakrishna, "Generating oracles from your favorite temporal logic specifications," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6. ACM, 1996, pp. 106–117.

[20] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[21] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 815–816.

[22] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *ECOOP 2006–Object-Oriented Programming*. Springer, 2006, pp. 380–403.

[23] C. Pacheco and M. D. Ernst, *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

[24] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[25] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[26] T. Xie and D. Notkin, "Checking inside the black box: Regression testing by comparing value spectra," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 869–883, 2005.

[27] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 183–192.

[28] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–7.

[29] A. Orso and T. Xie, "Bert: Behavioral regression testing," in *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. ACM, 2008, pp. 36–42.

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

[31] Z. Gao, Z. Chen, Y. Zou, and A. Memon, "Sitar: Gui script repair," *IEEE Transactions on Software Engineering*, 2015.