Which of My Failures are Real? Using Relevance Ranking to Raise True Failures to the Top

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

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

ABSTRACT

GUI reference testing is performed to detect regression errors in a modified or patched GUI software. Test cases are executed on the original and modified GUIs; differences in the states of GUI widgets across versions indicate potential defects. However, various factors (e.g., position, flakiness, resolution) create problems for accurate GUI state collection, leading to spurious state mismatches, and hence false positives; these need to be weeded out manually. In this paper, we show that the problem of false positives is significant, often inundating the tester with a large number of false bug reports, requiring a disproportionate amount of manual effort. We develop an entropy-based approach to rank each GUI widget property, and use it to determine whether a state mismatch (indicative of a bug) is real or a false positive. Our empirical evaluation shows that this ranking helps to percolate real bugs to the top of a set of reported bugs, thereby helping to economize tester time.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Software Testing—*Oracle, System Testing*

General Terms

Software Engineering

1. INTRODUCTION

One popular way to perform regression testing of a GUI-based software application uses "GUI reference testing" [13]. Given two versions of a GUI-based software, v_1 and v_2, test cases—consisting of sequences of GUI events and data inputs—are executed on both v_1 and v_2. The state of some or all GUI widgets is captured from both versions at a certain time instant, and then compared. If there is a mismatch, it is reported as a possible failure. Because some differences are expected (e.g., because v_2 may have new features), all mismatches are examined, typically manually, by a tester. Those due to bugs are recorded; others are discarded.

The above described GUI reference testing scenario would work perfectly if the GUI state could be captured deterministically. Unfortunately, this is not the case. Several factors contribute to the difficulty of capturing GUI state; we note only few. First, GUI test-case execution is known to be flaky [4], i.e., a test case may produce a different outcomes each time it is executed. One reason for this is the test harness and the application under test are two separate pieces of software, designed by different teams of programmers from different organizations, that must necessarily work together so that the harness can execute test cases, one event at a time, and collect GUI state at various time intervals. In previous work [4], we have shown that synchronization issues and artificially inserted delays can cause problems in test execution and state collection.

Second, some parts of the GUI state are tightly coupled with the underlying hardware, e.g., screen resolution. In this case, the *width* and *height* of a widget may be reported differently by the windowing system based on the screen's current resolution. Even if the resolution remains constant, the number of pixels may be misreported by a small number (usually 1). Third, because window placement decisions are typically made by the windowing system, taking into account other running applications, it is possible that the application under test is given a different location on the screen each time it is executed, resulting in a different output of `getLocationOnScreen()`. Finally, because window painting and repainting are resource intensive operations, it is difficult to predict when a window will finish displaying. Consequently, the time instant when the window state gets recorded may yield an incomplete state because the window may not have completed being displayed. In general, because the harness and application under test are disjoint software, it is non-trivial to synchronize them.

The above problems with accurate GUI state recording has implications for GUI reference testing. When a test case is executed on v_1 and v_2, and corresponding GUI states are compared, a mismatch may be a true mismatch (e.g., due to a bug or expected changes between v_1 and v_2) or a false positive because the GUI state was recorded incorrectly. In either case, the mismatch needs to be manually examined by a human tester. If there are too many false positives, the human tester has to spend a disproportionate amount of time weeding them out instead of focusing on true faults. In many cases, GUI test automation is abandoned in the face of an overwhelming number of false positives.

In this paper, we address the issue of false positives by ranking mismatches. We observe all reported values of widget properties and compute an *entropy* score per-widget-property. A property with a high entropy is more likely to lead to a false positive because it indiscriminately jumps between various values. All mismatches with this property are ranked low. The high-ranking mismatches percolate to the top of the reported set of mismatches. These comprise the mismatches associated with low-entropy properties. Because the values of these properties have largely been stable, mismatches are more likely to be due to real differences between versions.

We empirically evaluate our approach on 3 applications with 4 real bugs. Because of mismatches, we see 100% test cases failing because of false positives. The experiment results show that application of our approach helped to percolate real bugs to the top of the prioritized list of reported mismatches. In addition, we analyze the GUI properties that contribute to false positives in our case study.

We present background on GUI testing, GUI test cases, state models, and test oracles next. We also summarize related literature. In Section 3, we present our model of entropy using a simple running example. In Section 4, we present the design of our empirical study, with its results in Section 5. Finally we conclude with a discussion of future work in Section 6.

2. BACKGROUND & RELATED WORK

Automatic regression testing [19] has long been a research topic in the area of software testing. Also, as Graphical User Interfaces (GUIs) are playing a crucially important role in modern softwares, automatic testing of GUI applications [10, 8] are attracting more and more research efforts. This paper presents a technique, *autoOrac*, which can be used to identify real faults from false positives in regression GUI testing. Our technique is based on our previous work on automatic regression testing of GUI applications, with test case generation and test oracle as two key components. In this section, we will introduce the techniques that we have been using for automatic test generation and oracle. We will also explain why false positive is a major issue in this setting.

2.1 GUI Test Cases

We use the GUITAR [7, 15] system as our testing harness for automatic test generation. GUITAR is an automatic GUI Testing frAmewoRk which supports automatic *GUI ripping*, *Event Flow Graph (EFG) construction*, *test generation* and *test execution*.

GUI Ripping [11, 9] is a technique to automatically reverse engineer the GUI structure of an Application Under Test (AUT). The GUI ripper starts from the initial state of the application, and tries to traverse all the events available in the GUI in a depth-first manner. During the process, the ripper extract the AUT's *GUI structure* information including the hierarchal structure of windows, widgets, as well as their properties such as *title*, *width*, *height*, etc. The ripper also keeps track of the *invoking* events of each new window. This information, together with the GUI structure information will be used to construct the event model of the AUT.

In the EFG construction stage, a directed graph model is constructed to depict the possible interactions between the GUI events. Inside an EFG, a vertex represents an event (e.g., Click-on-Button, Check-a-RadioBox) and a directed edge from event x to y means that event y *may-follow*, i.e., can be executed immediately after, event x in some condition.

Then GUITAR will be able to generate test cases to satisfy a certain type of event coverage criteria [5, 14]. Each test case is a sequence of events that follows a certain path on the EFG. We call a test suite "*length-k*" test suite if it covers all *k-paths* (i.e., paths of length k) on the EFG. For example, a length-1 test suite covers all events and a length-2 test suite covers all edges on the EFG. The greater k is, the more test cases will be required to cover all k-paths. Thus it is often preferable to apply random test generation techniques to obtain long test cases.

2.2 GUI State Model & GUI Test Oracle

The test cases generated can be automatically executed, or replayed, by our testing harness. During replay, the GUI structure information can be captured after each test step. To be more specific, all properties of all widgets in all windows of the application after execution of each GUI event can be recorded in our GUI state model. And the GUI state information can be used to design automatic GUI oracle in regression testing.

Test oracle is an important part in testing as it is the mechanism to determine whether a test case execution passes or fails. Thus effective automatic GUI test oracles are necessary parts of the automatic regression system. Different from traditional applications which generally only use the final output values to determine the correctness of a software, GUI test oracles are much more complex in terms of input and output of the application. GUI testing may accept arbitrary combination of events as test inputs, and the outputs include the whole GUI states and other forms of outputs throughout the execution sequence. These problems make automatic GUI testing more complex and challenging.

An automatic GUI test oracle technique is proposed by Xie and Memon [20] which utilize the GUI state information. In their technique, they execute the same test suite on two versions, one correct version and one buggy version, of the AUT, and compare the values of GUI properties obtained from the two versions: a mismatch means the test case reveals a bug without considering about noises caused by false positives. An important contribution in their work is that it allows multiple options regarding the set of widget properties to check and the frequency of state checking. The tester may select to check the GUI properties associated with the event, or the current active window or all windows. Also the tester is allowed to check for mismatches after each step or only at the last step of the test case.

2.3 Why False Positives?

Even though some techniques are proposed to solve the automatic GUI oracle problem, they still tend to report a large number of *false positives* in real scenarios. A false positive is the case when mismatches are detected between executions without presence of a bug.

As introduced in previous sections of this paper, during execution of test cases, GUI test oracles are checking a rich set of GUI state information, including those properties that are very sensitive to internal or external runtime environment. The value of such properties are difficult to control even on the same machine and OS, and with the same libraries and inputs. In regression testing scenario where two versions of applications are involved, cases are even more complex. For example, some features from the previous version may be changed or even removed in the new version, and new added functional or GUI modules are also likely to create mismatches between versions.

Before our technique, *autoOrac*, human testers may have to manually go through all reported "failure" a big portion of which are false positives. This makes the testing procedure tedious and ineffective. Our technique proposed in this paper may help alleviate this problem. The novel idea is based on the observation that different GUI properties associated with different widget have a different chance to change during test execution. We can thus build an entropy model to depict the stability of different widget-properties. And with this model, we will be able to predicate more precisely which mismatches are real bugs.

2.4 Related Work

Regression testing has been extensively studied especially in fields such as test selection and minimization [18, 19]. These works aim to minimize cost of regression testing by eliminating redundant test cases. They assume the existence of automatic oracle whereas we do not have this ideal assumption in our work and target to solve the problem of false positives with automatic oracles.

There exist a number of techniques that target on generating or supporting oracles. Some approaches [3, 12] make use of specifications to support oracles. Richardson [17] presented a toolkit named TAOS (Testing with Analysis and Oracle Support) which allows users to write down expected outputs or to specify the ranges, etc. The major difference between our technique and theirs is that our tool works automatically at a lower level.

Some other techniques are proposed to generate oracle without specifications. JCrasher [2] generates oracles based on crash and exception information from execution on previous version. Eclat [16] learns a model from assumed correct executions to assert test results. The difference between this technique and ours is that it use the existing execution data to predict and pick test cases that are likely to reveal faults in future versions, whereas our model targets on measuring reliability of actual mismatches between previous and current versions. Harrold [6] and Xie [22] proposed spectra comparison approaches which capture internal program execution information to expose faults. Our technique is different from theirs in that we do not use internal program information and we build our entropy model based on execution information on the previous version only. Another tool called Orstra is developed by Xie [21] to assert object states and return values of methods. Orstra generates assertions based on observed behaviors and adds assertions to check future runs. Orstra is able to augment a unit-test suite with regression oracles. This tool is different from ours as well because we are asserting behaviors at the system level. In addition, although both Orstra and our technique make some kind of predications based on existing behavior, they are making predications on future runs whereas we are measuring the possibility of mismatches being real faults.

Finally, as mentioned in the Background section of this paper, Xie and Memon [20] proposed an automatic GUI oracle technique based on different levels of GUI state information. Their work laid the basis of *autoOrac*, but there are great differences between their technique and *autoOrac*. First, they are evaluating their technique on the same version of AUTs with instrumented faults, whereas *autoOrac*targets to make automatic GUI oracles usable on real regression scenarios with real applications and bugs where false positives tend to be massive. As a result, instead of purely relying on the GUI state information, the major contribution of our technique is to predicate the stability of widget-properties based on history execution data. This is a novel approach as far as we know.

3. MODELING WIDGET RANKING

We now describe how we compute the ranking of mismatches so that false positives are assigned lower importance compared to real bugs. To help explain the mathematical concepts, we use a running example to present our approach and computations. The example uses an open source application, jEdit[1], from Sourceforge[2] and a real regression bug reported on Sourceforge[3]. More information of the subject application is provided in Section 4. The example is simplified to make the illustration clearer.

In the example, after executing a sequence of GUI events on jEdit, a dialog "Edit Status Bar Entry" shows up allowing users to choose and add a new widget type. Figure 1 shows the screenshots of GUI state prior to adding the widget type: the top part is the correct version v_1 and bottom is the buggy version v_2. The top screenshot shows the GUI state of v_1 where the dropdown list titled "Choose a widget" is enabled; whereas the bottom screenshot shows the GUI state of v_2 where the dropdown list is erroneously grayed and hence a bug.

In a standard regression testing scenario, we will compare the GUI states after execution on v_1 and v_2 and identify the mismatches as bugs. In this real example, multiple mismatches can be observed in the GUI properties captured on the GUI states. For example, the width of the text field and the x-coordinate of the radio button "widget" are reported as different in two versions, and the dropdown list titled "Choose a widget" is enabled in v_1 but grayed out in v_2. Do all these mismatches show real bugs? And if not, how

[1]http://sourceforge.net/projects/jedit/
[2]http://sourceforge.net/
[3]http://sourceforge.net/p/jedit/bugs/3645/

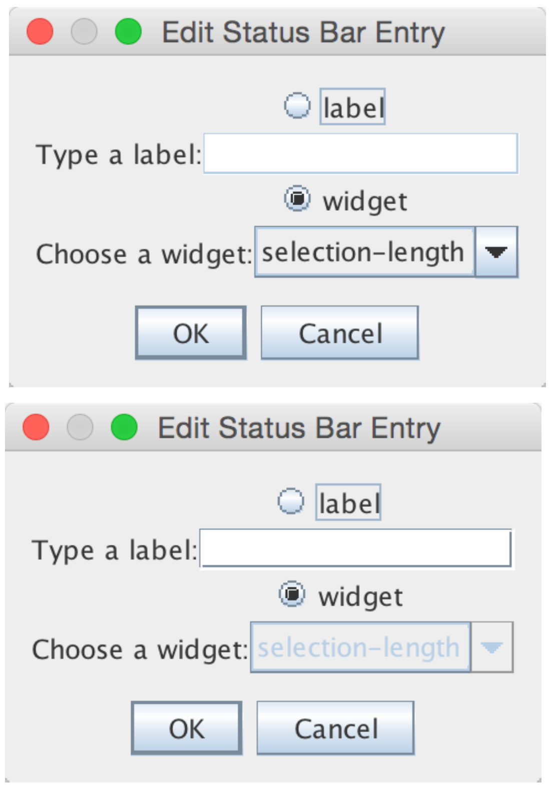

Figure 1: Screenshots of the Example Application

can we automatically identify mismatches related to the real bug? In the following parts of this section, we will apply our computations of entropy and ranking to show how they can help percolate to the top real bugs.

3.1 Stability of GUI Properties

In this step, we measure the stability of a GUI property. *Entropy* is widely used for measurement of stability [1]. It is also used in previous work [4] to measure the stability of test execution. In general, the higher the entropy value is, the less stable the property set is. We continue with our example to show how this metric works on GUI property data.

For each property and for each widget, we extract the set of values of this property that shows up during the execution of the test suite and calculate its entropy to finally obtain a value for stability based on the following definition.

Definition 1 (St(widget, property)). *The stability of a GUI property with regard to a certain widget is measured by the entropy of the set of property values (X) of the widget observed during the execution of the test suite on a certain version of AUT.*

$$St(property, widget) = H(X) = -\sum_{i=1}^{n} p(X_i) log_e(p(X_i))$$

where X_i stands for an identical property value and $p(X_i)$ stands for the probability of a property having value X_i.

The stability is defined as entropy of the set of values of a widget-property. Thus the smaller *St(widget, property)* is, the more stable the widget-property is.

Now assume that we have a pool of regression test cases on v_1 of our example AUT. After execution of all these test cases, we are able to collect the GUI state information: including all GUI properties of all widgets after the execution of each test step. During this process, we have observed different property values of different widgets for a certain times. We pick three example GUI widgets and the values observed regarding three properties: the *width* property measures the width of a window, container or widget in pixels; the *X-coord* property is the relative position of the top left corner of a widget in relation to the top left corner of the parent window; and the *isEnabled* is a *boolean* property which indicates whether a widget is usable or not. We show these sample widgets and properties in the first two columns of Table 1. In the third column, we show the total times the widget property is observed during the execution of the test suite. In Colums "Value 1" through "Value 3", we show the different values and the times these values are observed.

Based on the definition of *St(widget, property)*, we can calculate the stability of widget properties as entropy values. For example, the *width* property of the *Main Window* is observed with 3 different values showing up for 80, 50, and 70 times respectively. Thus its stability can be calculated as

$$-(\tfrac{80}{200} * \log_e(\tfrac{80}{200}) + \tfrac{50}{200} * \log_e(\tfrac{50}{200}) + \tfrac{70}{200} * \log_e(\tfrac{70}{200})) = 1.08.$$

As another example, the *isEnabled* property of the *dropdown list "Choose..."* has 2 different values occurring for 180 and 20 times respectively. Thus its stability can be calculated as

$$-(\tfrac{180}{200} * \log_e(\tfrac{180}{200}) + \tfrac{20}{200} * \log_e(\tfrac{20}{200})) = 0.33.$$

The second widget property has a smaller entropy value, thus it has greater stability than the former widget property. The results show that, in this example, properties such as *width* and X-coord are less stable and may lead to "spurious" mismatches, and properties such as *isEnabled* are much more stable and should thus be more reliably used in an automatic oracle in GUI regression testing.

3.2 Weighted Distance of GUI Widgets

GUI properties with greater stability (smaller *St* values) are considered more reliable in regression, and thus should be assigned greater weight in the distance calculation when such properties have mismatches. We assign a weight to each property that is proportional to *St(property)*.

Hence, for a GUI widget with property p and two property values (may be the same or different) in two executions, we calculate the weighted distance regarding to the widget and property as follows:

Definition 2. $Wd(widget, property) = St(p) * mis(p)$

where $mis(p)$ is 1 if the values of property p in two replays mismatch and is 0 otherwise.

Table 1: Sample Widgets and Property Values Observed During Execution on v_1

Widget	Property	Total Times	Value 1		Value 2		Value 3		Entropy
			Value	Times	Value	Times	Value	Times	
TextField	Width	**200**	335	**80**	336	**50**	337	**70**	1.08
	X-coord	**200**	249	**100**	251	**100**	–	–	0.69
	isEnabled	**200**	True	**140**	False	**60**	–	–	0.61
Radio Button "widget"	Width	**200**	110	**120**	112	**80**	–	–	0.67
	X-coord	**200**	350	**100**	352	**50**	351	**50**	1.04
	isEnabled	**200**	True	**60**	False	**140**	–	–	0.61
Dropdown List "Choose..."	Width	**200**	260	**120**	262	**60**	258	**20**	0.90
	X-coord	**200**	300	**90**	302	**50**	298	**60**	1.07
	isEnabled	**200**	True	**180**	False	**20**	–	–	0.33

The weighted distance is used to determine the overall importance of mismatches of property values of a certain widget in two executions. When we compute the weighted distance of a certain widget based on the GUI states captured from executions on v_1 and v_2 of the application, we quantify the chance this mismatch is a real fault because the stability of different properties are taken into consideration in the metrics.

In Table 2, we show the calculation for weighted distance based on our previous example. Column "Stability" shows the entropies of widget properties obtained from previous step. Column "Mis" will have a value 1 if the property of this widget is observed to have different values when the same test case is executed on v_1 and v_2 of the AUT. And finally Column "WD" show the value of weighted distance calculated based on our definition. The results show that 5 non-zero weighted distances in our example even though only 1 mismatch is expected to be related with the real bug.

Table 2: Weighted Distances of Sample Widget and Properties Between v_1 and v_2

Widget	Property	Stability	Mis	WD	Rank
TextField	Width	1.08	1	1.08	5
	X-coord	0.69	0	0	–
	isEnabled	0.61	0	0	–
Radio Button "widget"	Width	0.67	1	0.67	2
	X-coord	1.04	1	1.04	4
	isEnabled	0.61	0	0	–
Dropdown List "Choose..."	Width	0.90	1	0.90	3
	X-coord	1.07	0	0	–
	isEnabled	0.33	1	0.33	1

3.3 Widget Ranking

In the last step of *autoOrac*, we apply our metrics to the scenario of regression testing. To be specific, we obtain two GUI states by executing the same test cases on two regression versions, v_1 and v_2, of the AUT. For each widget-property observed in the two executions, we calculate the weighted distance *Wd(widget, property)*. We then rank the distances in ascending order to obtain the top K mismatches.

The list of top K widget-properties is presented to testers to confirm if they reveal real bugs or are false positives. When the test case actually fails, if the bug revealing mismatch is included in the top list, then *autoOrac* helps successfully distinguish bugs and save manual inspection effort.

To continue with our previous example, the ranking is shown in Column "Rank" of Table 2. Multiple non-zero weighted distances are obtained in the previous stages of our example. Now we rank the weighted distances in an ascending order to obtain the rank of each mismatch. As shown in the table, the *isEnabled* property of widget *Dropdown List "Choose..."* has the smallest weighted distance, and is thus considered as the mismatch that is most relevant with the bug. Thus, our technique *autoOrac* is able to alleviate human inspection effort by raising the most related ones to the top of list.

4. EMPIRICAL STUDY

We pose the following research questions regarding our approach, *autoOrac*:

RQ1: How effective is *autoOrac* in distinguishing mismatches related with real failures?

RQ2: What GUI properties are the major reasons for false positives?

To answer the first research question, we evaluate *autoOrac* on real applications and bugs. We measure the rankings of failure related mismatches of widget-properties in the scenario of regression GUI testing. In the second research question, we determin those mismatching widget-properties that are unrelated with the failures, i.e., false positives. To address this research question, we firstly provide a list of properties ranked by stability which can provide us with a general idea on characteristics of different GUI properties. Further, we group false positives identified in our case study by property to figure out flaky (unstable) widget-properties in the real world.

4.1 Subjects of Study & Bugs

The following three Java applications are used in our empirical study:

1. jEdit[4] is a text editor for programmers.
2. Jmol[5] is a molecular viewer for three-dimensional chemical structures.
3. JabRef[6] is a bibliography reference manager.

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

Table 3: Bugs of AUTs

Bug Id	$\lvert TC \rvert$	Steps	Oracle
3645	8	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 "+".	The "Enabled" property of the Dropdown List "Choose a Widget" is True in v_1 whereas false in v_2.
T3	8	1. Click on menu "Help"; 2. Click on menu item "About Jmol".	The logo of Jmol is shown in v_1 whereas not shown in v_2 in dialog "About Jmol".
65	6	1. Click on menu "Options"; 2. Click on menu item "Customize entry types"; 3.Click on the help icon in the new window.	The widget "JabRef Help" contains help content in v_1 but is empty in v_2.
1130	6	1. Click on toolbar icon "new" to create a new database.	The "Enabled" property of the menu item "Close database" has is true in v_1 whereas is false in v_2.

Table 4: Bugs in jEdit, Jmol and JabRef

AUT	Bug Id	v_1	v_2	v_3
jEdit	3645	4.4.2	4.5.1	5.0-pre1
Jmol	T3	12.2.34	13.0.1	14.2.11
JabRef	65	1.1	1.2	1.3.1
	1130	1.7.1	1.8-beta	2.0-beta

Table 5: Stability of Widget Properties

AUT	Max	Min	Avg
jEdit	1.94	0.00	0.02
Jmol	2.07	0.00	0.02
JabRef	1.61	0.00	0.04

For each AUT, we inspect multiple versions and select 4 reported regression bugs from these applications' bug reporting sites (usually SourceForge). These 4 bugs are picked because all of them are reflected on the GUI of the AUTs with different values of a certain widget-properties and thus suitable for our study. In Table 4, we show the versions in which the bugs were introduced and fixed. We use v_2 to notate the version in which the bug was first introduced. v_1 is used to notate the past version in which the broken feature still worked. Finally, we use v_3 to notate the future version in which the bug was first fixed.

We make use of our automatic testing harness to generate length 2 test cases that cover all edges of the event-flow graph (EFG). Among these test cases, a sample fault revealing test case is picked out for each bug and listed in Table 3. Column "$\lvert TC \rvert$" of the table shows the number of test cases picked for each bug. The necessary steps to trigger each bug are listed in the third column of the table, and the last column shows the oracles that can be used to detect the occurrence of each bug. In our experiments, all test cases are replayed automatically on both v_1 and v_2 of the AUT, and the GUI states, including all properties of all widgets after each test step are captured during the execution.

4.1.1 Addressing RQ1

We firstly show the stability of each GUI property based on execution of all test cases on v_1 in Table 5. The results show that there is a big variance between the greatest and smallest entropies for widget-properties on all 3 AUTs. But the average entropies are close to 0, meaning a big portion of the widget-properties show stable values.

In the next step we calculate and rank the weighted distances of widget-properties between v_1 and v_2 and the results are shown in Table 6. For each test case, we show the total number of widget properties collected by executing it on v_1 in Column "Total #". Then we pair up the GUI elements with widgets collected from execution of the same test case on v_2, and the number of paired widget-properties are shown in Column "Paired #". Then among those paired widget properties, we find pairs that have different property values, i.e., possible indications of failures. The number of mismatching pairs are shown in Column "Mismatching #". Finally, we rank all mismatching pairs of widget properties in ascending order of their weighted distance. To evaluate the effectiveness of our technique, we manually identify the mismatch that reveals the failure, and find out its position in the ranking which is shown in Column "Ranking" of the table.

As shown in Table 6, all test cases generate a number of mismatches most of which are false positives. With 100% test cases generating false positives, it is specially challenging and important to effectively percolate to the top mismatches related with real bugs. The results in Table 6 show that *autoOrac* successfully percolate real bugs to the top 30 of all the lists of mismatches, especially for Bug 3645 and 65 where real bugs are placed at top 5 among tens or hundreds of false positives. The mismatch related with Bug T3 is not located in the very top of the list because the buggy property "Text" involves long HTML contents. Since many elements in HTML such as properties and class names are unordered, thus semantically equivalent HTML contents are often observed to have different texts which lead to a low stability score of the property. The test case that triggers Bug 1130 can also trigger about 80 similar bugs at the same time. As observed from the ranked list, the mismatches related to these bugs, including Bug 1130, are located at the 6th-83th place and have very close weighted distances (ranging from 0.65 to 0.69).

4.1.2 Addressing RQ2

To fully address the research question, we firstly study the average stability of each property across widgets to grasp a overview of their characteristics.

Table 6: Widget Rankings

Bug Id	Total #	Paired #	Mismatching #	Ranking
3645	4147	3818	23	3
T3	2004	142	10	9
65	1555	1212	202	4
1130	2128	1814	92	30

Table 7: Sample Flaky and Stable Properties

Property	jEdit	Jmol	JabRef
IconImage	0.97	1.04	1.61
X-coord	0.34	0.38	0.44
Width	0.07	0.06	0.05
isEnabled	0.0	0.01	0.21
isEditable	0.0	0.0	0.0
Text	0.0	0.0	0.0

Table 7 shows the average entropies of some of the most stable and flaky properties in the 3 AUTs. As a general trend, GUI related properties such as *iconImage*, *X-coord* and *width* are more flaky with higher entropy values, and function related properties such as *isEnabled*, *isEditable* and *text* are more stable with lower entropy values, with property *isEnabled* as the only exception observed to have a higher average entropy in one AUT.

Further, to clarify how much each property (associated with its widget) contributes to the false positives, we group the false positives in the final mismatches by property. In Table 8, we show the properties that are major reasons of false positives. These properties can be divided into 2 general groups:

- **GUI rendering properties** including *icon* (the icon image of a widget), *width/height* (the width/height of a window or a widget in pixels), *X-/Y-coord* (the X-/Y-coordinate of the top-left corner of a window or a widget), *font* (the font type and size of the textual contents of a window or widget), *foreground/background* (the foreground/background color of a window or a widget), etc.
- **Functional properties** including *isSelected* (a *boolean* value indicating whether a widget is selected/focused or not), *isEnabled* (a *boolean* value indicating whether a widget is usable or not), *text* (the text shown on a widget or container), *locale* (a locale stands for a specific country, region or native), *accelerator* (an accelerator is the shut-cuts for a certain widgets), etc.

The results show that rendering properties associated with corresponding widgets make up 93% of all false positives and are thus much more flaky than functional widget-properties. This makes functional properties more reliable as automatic oracles. The rendering properties are purely related to the rendering of graphical interface and less related to the correctness of functionalities of AUTs. Unless a bug which fails to properly render a GUI object due to problems of the OS or the standard GUI library which is very unlikely to happen, we can safely exclude those properties to reduce false positives in *autoOrac*. When a bug is related with a rendering property, it could be more difficult to be identified. But fortunately the biggest advantage of our technique is that it never treats a property in isolation – a property is always bound with a GUI widget. And by associating widget and properties, our ranking model based on entropy will automatically figure out the most reliable widget-properties.

5. CONCLUSIONS & FUTURE WORK

Much prior work on GUI reference testing and test oracles based on object/GUI state assumes that the GUI state can be obtained reliably. In this paper, we showed that this is not a reasonable assumption. For a given set of test cases and widget-properties, we showed that if all elements of the GUI state are considered during reference testing, all test cases would fail. Each of the failures would need to be examined manually, requiring a significant amount of work, perhaps eliminating the benefits of test automation.

While recognizing the limitations in our computing platforms and tools to effectively and predictably capture GUI state, we presented a new solution that ranks state mismatches based on the “flakiness” or “instability” of a widget property. We used entropy to model this instability and used it to rank a widget property – the more unstable, the less it is trusted for a test oracle. We showed that such a ranking helps to percolate true mismatches to the top (they are highly ranked), thereby helping the human tester to focus on a prioritized list of failures.

We have identified several directions for future work. First, we need to extend our experiments to additional subjects, bugs, and widget properties, so that we can start to refine our classification of flaky properties. Second, we need to add different types of bugs, especially those in which widgets may be missing in one version, and hence, comparison for reference testing may be impossible. Finally, we will examine non-GUI applications to determine whether similar problems happen in their reference testing.

Our study shows that the ranking scheme ranks faulty properties higher. However, we need to determine if this will actually help software engineers in practice, for which we will conduct a study. We need also to determine a proper cutoff point of mismatches for engineers to check.

Acknowledgments

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.

6. REFERENCES

[1] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 257–267, Nov 2013.

[2] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

Table 8: False Positives Grouped by Property

Bug ID	Total	Font	Width	Height	Icon	isEnabled	isSelected	Y-coord	Text	Locale	Foreground	Accelerator
3645	22		3	3	5	5	5					
T3	9			8				1				
65	201	103	37	33	23			1	2			
1130	14		8	3						2		1
Total	246	103	48	47	28	5	5	2	2	2	1	1

[3] L. K. Dillon and Y. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 106–117. ACM, 1996.

[4] 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.

[5] D. Hackner and A. M. Memon. Test case generator for GUITAR. In *ICSE '08: Research Demonstration Track: International Conference on Software Engineering*, Washington, DC, USA, 2008. IEEE Computer Society.

[6] 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*, 10(3):171–194, 2000.

[7] A. M. Memon. *A comprehensive framework for testing graphical user interfaces.* PhD thesis, 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware).

[8] A. M. Memon. *Automated GUI Regression Testing Using AI Planning*, volume 56, pages 51–100. World Scientific Publishing Co., 2004.

[9] A. M. Memon. *Using Reverse Engineering for Automated Usability Evaluation of GUI-Based Applications.* Springer-Verlag London Ltd, 2009.

[10] A. M. Memon, I. Banerjee, and A. Nagarajan. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the International Conference on Software Maintenance 2003*, Sept. 2003.

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

[12] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 30–39, New York, NY, USA, 2000. ACM Press.

[13] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM Press.

[14] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM Press.

[15] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.

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

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

[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[19] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[20] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

[21] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006–Object-Oriented Programming*, pages 380–403. Springer, 2006.

[22] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *Software Engineering, IEEE Transactions on*, 31(10):869–883, 2005.