Making System User Interactive Tests Repeatable: When and What Should we Control?

Zebao Gao*, Yalan Liang†, Myra B. Cohen†, Atif M. Memon* and Zhen Wang†
*Department of Computer Science, University of Maryland, College Park, MD 20742, USA
Email:{gaozebao,atif}@cs.umd.edu
†Department of Computer Science & Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588, USA
Email:{yaliang,myra,zwang}@cse.unl.edu

***Abstract*—System testing and invariant detection is usually conducted from the user interface perspective when the goal is to evaluate the behavior of an application as a whole. A large number of tools and techniques have been developed to generate and automate this process, many of which have been evaluated in the literature or internally within companies. Typical metrics for determining effectiveness of these techniques include code coverage and fault detection, however, with the assumption that there is determinism in the resulting outputs. In this paper we examine the extent to which a common set of factors such as the system platform, Java version, application starting state and tool harness configurations impact these metrics. We examine three layers of testing outputs: the code layer, the behavioral (or invariant) layer and the external (or user interaction) layer. In a study using five open source applications across three operating system platforms, manipulating several factors, we observe as many as 184 lines of code coverage difference between runs using the same test cases, and up to 96 percent false positives with respect to fault detection. We also see some a small variation among the invariants inferred. Despite our best efforts, we can reduce, but not completely eliminate all possible variation in the output. We use our findings to provide a set of best practices that should lead to better consistency and smaller differences in test outcomes, allowing more repeatable and reliable testing and experimentation.**

I. Introduction

Numerous development and maintenance tasks require the automated execution and re-execution of test cases and often these are run from the system or user interface perspective [1]. For instance, when performing regression testing on a web application [2] or integrating components with a graphical user interface (GUI), a test harness such as Selenium [3], Mozmill [4], or Unified Functional Test (UFT) [5] can be used to simulate system level user interactions on the application by opening and closing windows, clicking buttons and entering text. Re-execution of tests is also needed when performing invariant generation [6], [7], and when experimenting to evaluate the effectiveness of new test generation techniques [8]–[10].

Over the past years, numerous papers have proposed improved GUI and Web automation testing/debugging techniques [8]–[17], and common metrics have been used to determine success such as fault detection [12], [18], [19], time to early fault detection [20], business rule coverage [9], and statement, branch and other structural coverage criteria [21]. Test oracles—mechanisms that determine whether a test passed or failed—for these applications operate at differing degrees of precision [22] such as detecting changes in the interface properties [23], or comparing outputs of specific functions [24], or through the observation of a system crash [25]. The assumption for all of these scenarios, however, is that these *System User Interactive Tests* (SUITs) can be reliably replayed and executed, i.e., they produce the same outcome (code coverage, invariants, state) unless either the tests or software changes.

Recent work by Luo et al. [26] illustrates that many tests do not behave deterministically from the perspective of fault detection – what is commonly called *flaky* behavior in industry [27]. Other research by Arcuri et al. [28] has developed methods to generate unit tests that behave differently in varying environments, controlling this by replacing API calls and classes related to the environment with mocked versions. And there has been attention given to revisiting the expectation of sequence-based testing – that of test order independence [29].

In our own experience with GUI testing and benchmarking [27], [30], [31], we have learned that the ability to repeat others' experimental results with different platforms and/or setups is hard to do. We set out to compile a comprehensive list of factors that should be controlled to ensure testing can be repeated reliably within and across platforms. Unfortunately, the harder we tried, the more we learned that our assumptions about how well we could control tests were incorrect. We found that even within the same platform, simple changes to system load or a new version of Java, or even the time of day that we ran a test, could impact code coverage or the application interface state.

In this paper we have taken a step back to empirically evaluate the set of factors that we have observed have the largest impact on our results, and to examine how and if these can be controlled. We have designed a large empirical study to evaluate for certain classes of interactive applications (those driven by the graphical user interface or GUI), what extent of variation is seen, and which factors matter the most. We use an entropy based metric to determine the stability of test runs and study this for different test metrics (code coverage, GUI state, and invariants). Our results show that the impact is large for many applications when factors are uncontrolled (as many as 184 lines of code coverage differ and more than 95% false positives for fault detection may be observed).

Despite our ability to control factors and reduce variance, there still may exist some that we cannot control - that

are application specific, or sensitive to minor changes in timing or system load, therefore a single run of any testing technique (despite fault determinism) may lead to different code coverage or even different invariants, and that unless one accounts for the normal variance of a subject, a single test run may be insufficient to claim that one technique is better than another, or even that a fault is really a true fault.

The contributions of this work are:
1. The identification of a set of factors that impact the repeatability of testing results at three different layers of user interactive applications;
2. A large empirical study across a range of platforms on a set of open source applications that measures the impact these factors have on different types of test outputs; and
3. A set of practical steps that one can take when comparing different test runs to ensure higher repeatability.

In the following section we provide a motivating example and discuss related work. We follow this with our empirical study in Section III followed by results (Section IV). Finally, in Section V we conclude and present future work.

II. Motivating Examples

We begin by differentiating three layers, illustrated in Figure 1 of a user-interactive software application: (1) *User Interaction Layer:* we typically use this layer to execute SUITs and extract information for test oracles, (2) *Behavioral Layer:* to infer properties, such as invariants, regarding the test execution, and (3) *Code Layer:* for code coverage.

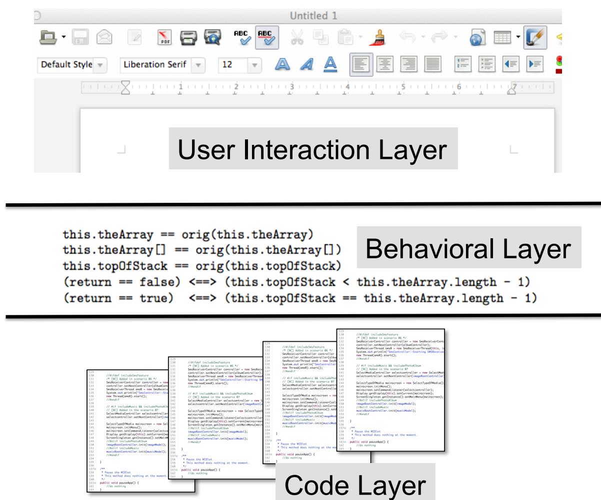


Fig. 1: Layers of a User Interactive Application

User Interaction Layer: We use this to interface with the application and run tests. For example, in the Android system, one could write the following code segment in a system test to programmatically discover and click on the "OK" button.

```
UiObject okButton = new UiObject(new UiSelector().text("OK"));
okButton.click();
```

We also can use this layer to identify faults since this is the layer that the user sees. For instance, if a list widget fails to render or displays the wrong information, then this layer will reveal a fault. From an automation perspective, the properties of the interface widgets can be captured and compared to a correct expected output, e.g., using the following code that (1) gets a handle to the current screen, (2) clicks on a list, and (3) checks whether the list contains the text "Android".

```
solo = new Solo(getInstrumentation(), getActivity());
solo.clickInList(1);
assertTrue(solo.searchText("Android")); // Assertion
```

During our experiments, we find variation in the properties of certain interface widgets between runs, that would appear as a fault to the test harness (and would be a real fault if this happened during manual execution), but that is most likely an artifact of automated test execution – a false positive. For instance in the application, `JEdit`, described later in Section III, we found that the widget *org.gjt.sp.jedit.gui.statusbar.ErrorsWidgetFactory$ErrorHighlight* had empty text in one run, but was filled in during other runs. We believe this had to do with a delay between test steps that was too short when the system load increased, preventing the text from rendering to completion before the next test step occurred. A test case that checks the text in this widget may non-deterministically fail during one run and succeed in another.

Behavioral Layer: This layer presents behavioral data, such as runtime values of variables and function return values, and returning data to an external location or database, which may be obtained from the running program via instrumentation hooks. Such data can also be mined/analyzed to infer invariants. Invariant detection involves running a set of test cases and inferring from this, a set of properties that hold on all test cases. Invariants represents high level functionality of the underlying code and should be consistent between runs.

In our experiments, we found differences in the invariants reported. For instance, in the application `Rachota`, Section III, we found that approximately in two of every ten runs, the application started faster than normal and generated the following two extra invariants related to the startup window:

```
this.lbImage == orig(org.cesilko.rachota.gui.StartupWindow.
                                    startupWindow.lbImage)
this.loading == orig(org.cesilko.rachota.gui.StartupWindow.
                                    startupWindow.loading)
```

not found in the other eight runs. In each run we used exactly the same set of test cases. The "correct" set of invariants is dependent on the speed at which the window opens and again, may be an artifact of the system load or test harness factors.

Code Layer: At the lowest level we have the source code. It is common to measure code coverage at the statement, branch or path level to determine coverage of the underlying business logic. It is very commonly used in experimentation to determine the quality of a new testing technique.

In our experiments, we found numerous instances of the same test case executing different code. In fact, this was a large motivating factor for our work. The harder we tried to deterministically obtain ten runs with the exact code coverage,

the more we learned that this may not be possible. We also ruled out factors such as concurrency and multi-threading. The following code shows an example of memory dependent code from the application DrJava that we cannot deterministically control with our test harness.

```
if (whole == 1) {
    sb.append(whole);
    sb.append(' ');
    sb.append(sizes[i]);  ...}
```

In this code (lines 571-580 of StringOps.java, the *memSizeToString()* method) we see code that checks memory usage so that it can create a string object stating the memory size. It checks for whole block boundaries, e.g., 1B, 1KB, 1MB, via `whole == 1` and constructs the string content accordingly. Because the actual memory allocated to the executing JVM may vary from one run to the next, in our experiments one in ten executions covered this code because by chance the space was not equal to a block.

The next code segment is an example of code that we can control by making sure the environment is mimicked for all test cases.

```
public static int parsePermissions(String s) {
    int permissions = 0;
    if(s.length() == 9) {
      if(s.charAt(0) == 'r')
          permissions += 0400;
      if(s.charAt(1) == 'w')
          permissions += 0200;
      ....
      else if(s.charAt(8) == 'T')
          permissions += 01000;}
    return permissions;}
```

In this code (JEdit, *MiscUtilities* class lines 1318-1357) the application checks file permissions. We found that the files did not always properly inherit the correct permissions when moved by the test script and this caused 9 different lines being covered between executions, or when time and date were involved as has been described in [28]. Other examples of differing code coverage occurred when opening screens of windows have code that waits for them to settle (based on a time).

These are just some examples of the types of differences we found at the various layers when we examined our results. Our aim in this work is to identify how much these impact our testing results and provide ways to minimize these differences. We next present the set of factors that we believe cause the largest impact on test differences in our applications.

A. Factors Impacting Execution

We categorize the factors we believe have the greatest impact on running SUITs deterministically into four groups:

1. **Test Execution Platform.** Many of the applications that we test can be run on different execution platforms which includes different operating systems (e.g., Windows Mac OS, Linux) or on different versions of the same operating system (e.g. Ubuntu 12, Ubuntu 10). Different operating systems may render system interfaces differently and could have different load times, etc. In addition, applications often contain code that is operating system specific.

2. **Application Starting State/Configuration.** Many applications have preferences (or configuration files, registry entries) that impact how they start up. Research has shown that the configuration of an application impacts test execution [32] therefore we know that this starting point is important. Even if we always start with a default configuration at the start of testing, test cases may change the configurations for future tests. Therefore the program configuration files should be located and restored before each test is run.

3. **Test Harness Factors.** Test harnesses such as Selenium contain parameters such as *step delays* or *startup delays* to ensure that the application settles between test steps, however this is often set to a default value and/or is set heuristically by the tester. Long delays may mean that the application pauses and runs additional code, but short delays may not allow completion of some functionality, particularly when system load or resources vary between executions. In early experiments we ran tests on a VM where we can change CPU and memory and have found that reducing memory and/or CPU has a large impact on the ability to repeat test execution – primarily due to the need for tuning these parameters.

4. **Execution Application Versions.** If we are running web applications using different web browsers or Java programs using different version of Virtual machines (e.g. Java 7 versus 6 or OpenJDK versus Oracle Java), we may run into differences due to support for different events and threading policies.

We select a subset of these factors in our experimentation to evaluate them in more detail.

III. Empirical Study

We now evaluate the impact of factors that we have identified (summarized in the previous section) on test repeatability in SUITs via the following research questions.

RQ1: To what extent do these factors impact code coverage?

RQ2: To what extent do these factors impact invariant detection?

RQ3: To what extent do these factors impact GUI state coverage?

Note that our questions include one for each of the layers. We start with the lowest code layer (code coverage) and end with the highest user interaction layer (GUI state).[1]

A. Objects of Analysis

We selected five non-trivial open source Java applications with GUI front ends from sourceforge.net. All of these have been used in prior studies on GUI testing. Table I shows the details of each application. For each we show the version, the lines of code, the number of windows and the number of events on the interface. Rachota is a time tracking tool allowing one to keep track of multiple projects and create customized time management reports. Buddi is a personal financial tool for managing a budget. It is geared for those with little financial background. JabRef is a bibliography reference manager.

[1] We have included supplemental data of our experiments at http://cse.unl.edu/∼myra/artifacts/Repeatability/.

`JEdit` is a text editor for programmers. Last, `DrJava` is an integrated development environment (IDE) for Java programs.

TABLE I: Programs used in Study.

Name	Version	LOC	No. Windows	No. Events
Rachota	2.3	8,803	10	149
Buddi	3.4.0.8	9,588	11	185
JabRef	2.10b2	52,032	49	680
JEdit	5.1.0	55,006	20	457
DrJava	20130901-r5756	92,813	25	305

B. Experiment Procedure

Having selected the applications, our experiment procedure involves executing a number of test cases on these applications multiple times on various platforms and collecting information at the 3 layers discussed earlier. For each application we run 200 test cases randomly selected from all possible length 2 test cases generated by the GUITAR test case generator that are executable (and complete) on all three different platforms, Ubuntu 12.04, Redhat Scientific Linux 6.4 and Mac OSX 10.8. The Ubuntu machine is a stand-alone server with an Intel Xenon 2.4GHZ CPU and 48 GB of memory. The MacOS machine is a laptop with a 2.5 GHZ Intel Core and 8GB of memory and the RedHat Scientific Linux machine is an Opteron cluster server running at 2000MHz with 8GB of memory on each node. All test cases are short, which reflects what might be done for overnight regression or *smoke testing* [33]. We decided that using the shortest possible (reasonable) tests would keep us from unduly biasing this data. We view this as a baseline – if we can't control length 2 tests, then longer tests should be even harder. For each application and experiment configuration, we run the tests 10 times using the GUITAR replayer [34], [35].

We instrumented each application with the Cobertura code coverage tool [36] to obtain the line coverage. We obtain the GUI state from the GUITAR oracle generator. For the oracle, we excluded common patterns that are known to be false positives (see [35]). For the code coverage, we parse the Cobertura report files to determine if we cover the same (or different) lines of code in each run. We share the test cases and scripts between platforms to ensure consistency.

C. Independent Variables

We selected our factors from each of the four categories based on those identified in Section II.

1) **Platform:** We use the three operating systems that are described above (Ubuntu, Mac and Scientific Linux).
2) **Initial Starting State/Configuration:** We control the initial configuration of each application, input data files that are loaded or written to and when possible (on the Ubuntu and Mac OS stand-alone machines) we control the date and time of the machine running the tests (note that we could not control the time on our Redhat cluster, but try to run tests within the same day when possible).
3) **Time Delay:** These are the delays that are used in the test harness to control the time that GUITAR waits to stabilize during replay of each step in the test case.
4) **Java Version:** We use three different Java versions in our experiments: Oracle JDK 6 and 7, and OpenJDK 6.

Our experiments vary each of the factors above. We do not vary all combinations of factors, but have designed a set of experiments that we believe is representative. The experiment configurations are shown in Table II. Each configuration (row) represents a set of conditions.

TABLE II: Configurations of our Experiments

Runs	Config Fixed	Input Files	Date& Time	Delay	JDK
1. Best	Y	Y	Y	Best	Oracle 6
2. Unctrl	N	N	N (rand)	0ms	Oracle 6
3. No Init	N	N	N (actual)	best	Oracle 6
4. D-0ms	Y	Y	Y	0ms	Oracle 6
5. D-50ms	Y	Y	Y	50ms	Oracle 6
6. D-100ms	Y	Y	Y	100ms	Oracle 6
7. D-200ms	Y	Y	Y	200ms	Oracle 6
8. Opn-6	Y	Y	Y	Best	OpenJDK 6
9. Orc-7	Y	Y	Y	Best	Oracle 7

Best means the best configuration (our gold standard for the experiments). For this we use the same configuration setup and use the same initial input files for the applications so that its starting state is the same. We also control the time (when possible) and fix the Java version to Oracle 6. To obtain the best configuration, we first tried to control as many factors as possible, and heuristically selected the best delay value for each different platform (where best shows the smallest variation based on a visual inspection of a sample of the test cases). We then fixed this configuration as our *best* configuration and created variants of these for study.

Unctrl means uncontrolled. This is expected to be our worst configuration. We do not control any of the factors mentioned. We just run our test cases with the default tool delay value (0ms), and do not reset the starting configuration files or provide a fixed input file. We use a random date (on the two platforms where we can control this).

Starting from our best configuration, we removed the initial configuration/starting state files. We call this **No Init** (see configuration #3). We then varied the delay values (shown as **D-0** through **D-200ms**, while keeping the configuration files and inputs fixed. Our last configurations (**Opn-6** and **Orc-7**) use the best delays and control all other factors but use different versions of Java (Open JDK 6 and Oracle 7) instead of the default version (Oracle 6).

D. Dependent Variables: Metrics

We gather code coverage, invariants, and interface properties (oracles) as we run the tests. For code coverage we use statement coverage as reported by Cobertura. For interface oracles we use the states returned by GUITAR. To avoid false positives and compare only meaningful properties of GUI state, we filter out properties, such as the ID of widgets given by the testing harness, minor differences in coordinates of widgets, etc. The remaining differences should be meaningful such as the text in a text field, or missing widgets. For invariant detection we use the Daikon Invariant Detector [37]. Due to the

large number of invariants generated we only selected three classes for study, those with the highest code coverage. To calculate the *stableness* of our runs, we use entropy as our measure [38], [39]. We illustrate this metric next.

Assume we have the test cases with the coverage metrics as shown in Table III. The coverage can be line or branch or some other unit (in our experiments we use line coverage). The test suite (TS) includes 4 test cases (rows TC1- TC4), is executed 4 times on a subject application that has 6 lines of code (cols 1..6 within each run). A dot in the table means that the line is covered during a single execution of the test case. For example, In the first run of test case 1 (TC1), line 2 and 3 are covered whereas all other lines are not.

TABLE III: Example Test Case/Suite Coverage Variance

#Cov	Run 1						Run 2						Run 3						Run 4						Const Groups
	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6	
TC1		●	●					●	●					●	●					●	●				4
TC2	●	●		●			●	●		●	●		●	●		●	●		●	●		●	●		3/1
TC3		●			●			●			●		●	●		●	●		●	●		●	●		2/2
TC4		●	●	●				●	●	●				●		●				●	●	●		●	2/1/1
TS	●	●	●	●	●		●	●	●	●	●		●	●	●	●	●		●	●	●	●	●	●	3/1

Definition 1 (Consistently Covered Lines): A line is *Consistently Covered* in two runs *iff* the line is covered in both runs or not covered in either of the runs.

In our example, Line 1 is consistently covered in Run 1 and Run 2 of TC1 because Line 1 is not covered in either run. Line 2 is also consistently covered, because it is covered in both runs.

Definition 2 (Consistent Coverage): Two runs of a test case or test suite have *Consistent Coverage iff* all lines of the subject application are consistently covered in both runs.

Both Run 1 and Run 2 of TC1 cover line set {#2, #3} and thus have consistent coverage.

Because the consistent coverage relationship is *reflexive*, *symmetric*, and *transitive* and is hence an *equivalence relation*, we can divide all runs of a test case/suite into equivalence groups based on the consistent coverage relationship, and measure **stableness** of a test case/test suite as the *entropy* (or H) of the group based on the following formula [38]:

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_e(p(x_i))$$

where n is the number of groups and $p(x_i)$ is the probability that a certain run falls into the ith group.

For example, the 4 runs of TC1 are divided into a single equivalence group of consistent coverage; thus we have $H(X)_{TC1} = -(4/4 * log_e(4/4)) = 0$. The 4 runs of TC4, however, are divided into 3 groups with sizes 2, 1 and 1, and the corresponding entropy is $H(X)_{TS} = -(2/4 * log_e(2/4) + 1/4 * log_e(1/4) + 1/4 * log_e(1/4)) = 1.04$.

To measure the impact of the variance, we further measure the *range of difference* of all runs of a test case/suite.

Definition 3 (Range of Difference): The *Range* of all runs of a test case or test suite is the total number of lines that are not consistently covered in any two runs.

For example, in the 4 runs of TC1, the same set of lines are covered in each run, thus the range of difference is 0, while the test suite has a range of 1 (at line #6). We show the ranges and entropies calculated for test cases and the test suite in Table IV. The average range and entropy of test cases are shown in the last row of the table. Note that the more unstable the groups are, the greater the entropy value is. TC1 has perfect stability and thus has an entropy value 0. TC4 is the most unstable and thus has the greatest entropy value for test cases. The entropy of a test suite is generally smaller than the average entropy of all test cases. This is because some lines that are not covered by one test case may be covered by another and the individual differences are erased (we see this phenomenon in our study). If however, tests do not have this property, then the entropy of test suite will be higher than the average entropy of test cases.

TABLE IV: Entropy Metrics for Table III

Metrics	Inconsistent Lines	Range	Groups	Entropy
TC1	{}	0	4	0
TC2	{#5}	1	3/1	0.56
TC3	{#1, #4}	2	2/2	0.69
TC4	{#3, #6}	2	2/1/1	1.04
TS	{#6}	1	3/1	0.56
Average		1.25		0.57

E. Threats to Validity

One threat to validity of our study is that some of our individual results may be test harness (GUITAR) specific, but we believe that the lessons learned should be transferable to other tools. To reduce this threat we have performed some additional studies using Selenium and discuss this in Section IV-D. We used only 5 applications in these studies, but we selected ones that are different and all have been used in prior studies on GUI testing. For state comparison, we use the GUI state of the first run as the oracle. For invariant detection we only selected a small subset of classes to instrument and obtain invariants due to scalability issues. This means that the size of both our state and invariant variances may differ if we modify the selection. However, our results are still relevant since we have found differences; the study conclusion still holds. Finally, we may have used the wrong metrics for our different layers, but believe that code coverage, invariants and GUI properties are the most common types of information that would be obtained during testing of this type, and entropy has been used in prior work to show stability.

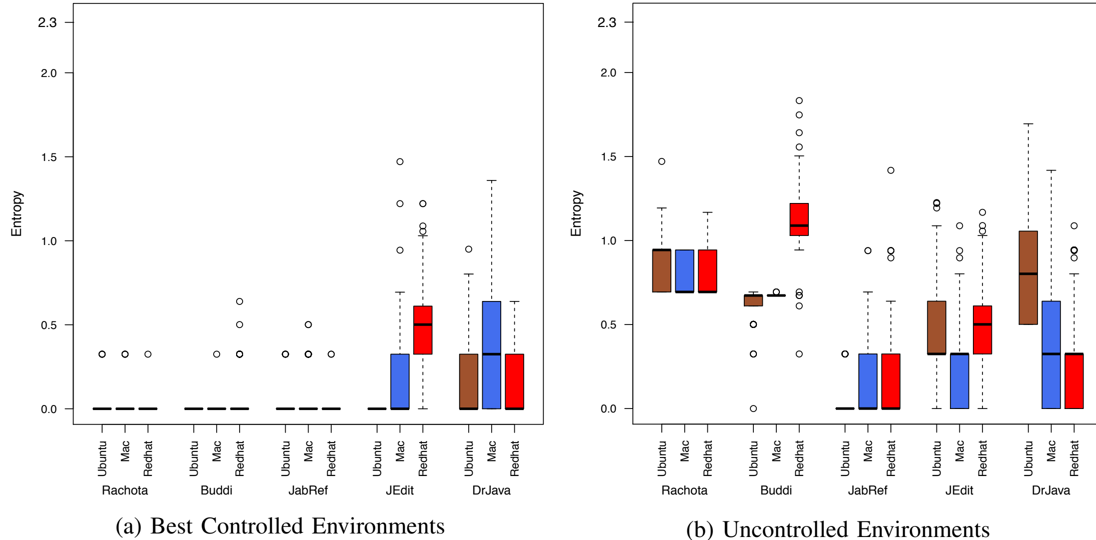
IV. Results

A. RQ1: Code Coverage

We begin by looking at the data for code coverage using the first and second configurations from Table II. These configurations include the *best* with everything controlled and the potentially worst configuration (where we control nothing). Table V shows the Test suite (TS) entropies for each application on each operating system. The rows labeled TC

TABLE V: Comparison Between Best & Uncontrolled Environments for Code Coverage and GUI False Positives.

#Metrics		Rachota			Buddi			JabRef			JEdit			DrJava		
		Ubuntu	Mac	Redhat	Ubuntu	Mac	Redhat	Ubuntu	Mac	Redhat	Ubuntu	Mac	Redhat	Ubuntu	Mac	Redhat
TS	Best	0.67	0.69	0	0	0	1.23	0	0	0	0	0.33	0.33	0	0	0.64
	Unctrl	1.28	0.94	0.94	0	0.94	2.16	0	0	1.61	1.64	1.01	0.90	1.83	0.90	1.61
TC	Best	0.00	0.00	0.00	0	0.00	0.15	0.00	0.01	0.00	0	0.09	0.24	0.06	0.20	0.05
	Unctrl	0.42	0.40	0.41	0.32	0.34	0.57	0.01	0.06	0.10	0.22	0.10	0.23	0.41	0.18	0.13
Range	Best	0.05	0.05	0.03	0	0.02	39.56	0.03	0.17	0.02	0	0.88	6.21	0.32	3.82	2.70
	Unctrl	83.85	80.22	83.09	4.98	5.00	90.83	2.72	2.72	181.62	72.63	5.03	1.99	184.22	13.19	13.37
FP	Best	4.44	1.44	2.11	0.06	0	1.56	0.50	0.06	0.50	0.11	3.17	1.50	0	5.67	0.17
	Unctrl	96.61	96.44	69.39	66.56	76.67	7.50	1.44	19.00	14.56	34.67	19.00	5.33	24.67	38.17	14.78



(a) Best Controlled Environments

(b) Uncontrolled Environments

Fig. 2: Entropies of Line-Coverage Groups of 3 Platforms in Best & Uncontrolled Environments

are the average entropy of the individual test cases within the test suite. In these tables we have 10 runs of test cases. Note that 0's without decimal places have true 0 entropy, while 0.00 indicates a very small entropy that is rounded to this value. The highest entropy occurs when all 10 runs differ – in this case we have an entropy of 2.3. When only a single run out of 10 differs the entropy is 0.33 and when half of the runs differ we have an entropy of 0.69. We see lower (close to zero), but not always zero, entropy when we control the factors, and higher entropy when we don't. We see differences between the applications and between platforms. We also show the range of coverage (in lines) which is the average variance across test cases. We can see that in the uncontrolled configuration we see as high as 184 lines on average differing and in the best we see closer to zero. However, we still have a few platforms/applications (such as Redhat running `Buddi`) where there is a large variance (90+ lines). This is because we were not able to control the time/date on this server and `Buddi` uses time in its code.

We show this data in an alternative view in Figure 2. The entropy of all 200 test cases by application and operating system within the application. Figure 2(a) shows the best while (b) shows the uncontrolled factors. Entropy is lower in the best environment (but not zero) across most of the applications and that it varies by platform. `Rachota` and `Buddi` have almost zero entropy in their test cases while `JEdit` and `DrJava` have a non-zero entropy even when we control its factors. Only the Ubuntu platform with the best configuration shows a zero entropy.

We next turn to the configurations that do not control the initial environment (No Init), vary the delay time (D-$x$ms) and vary Java versions (Opn-6 and Orc-7). We show this data in Tables VI through VIII. Table VI shows the test suite entropy, while Table VII shows the average test case entropy. Table VIII shows the variance in line coverage. We see that the initial starting state and application configuration (No Init) has an impact on some of our applications, but not as large as we expected, when we control the other factors.

A boxplot of the entropies by application when we don't control the initial state and configurations is seen in Figure 3. We show a boxplot of the entropies by application for different delays in Figure 4. We see that the different delays also impact the entropy and the delay value varies by application and platform. Finally, looking at the boxplots of the Java Versions in Figure 5, we see differences between Java versions, the largest being with `DrJava` using Oracle 7.

B. *RQ2: Invariants*

To examine the results at the behavioral or invariant level we examine the invariants created by Daikon. Two runs of a test case have the same behavior if all the invariants that hold in these two runs are exactly the same. We use

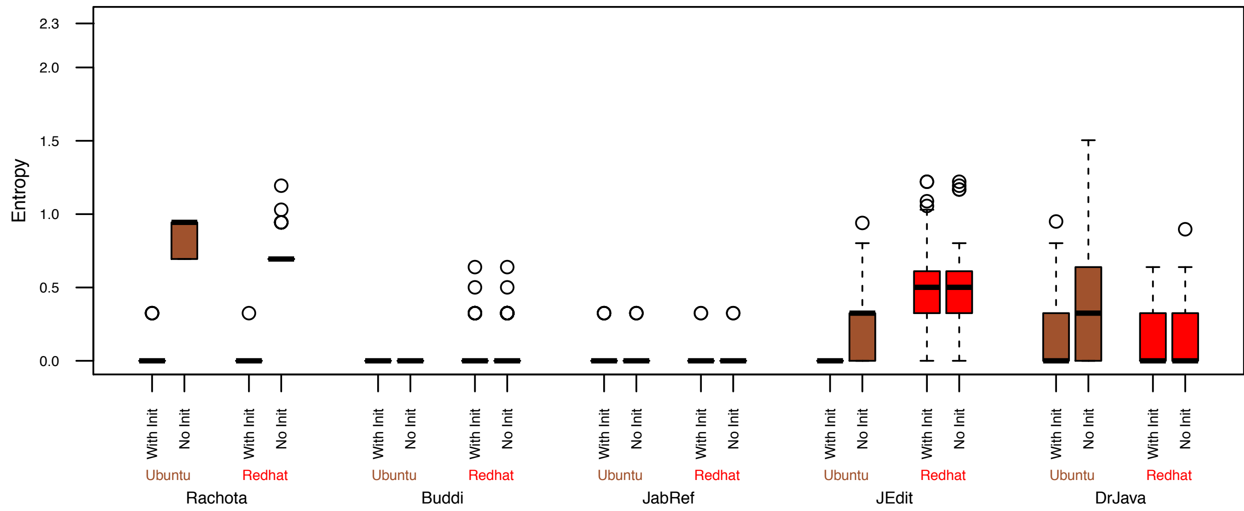


Fig. 3: Entropies of Test Cases of 2 Platforms with/without Initial State and Configuration Control

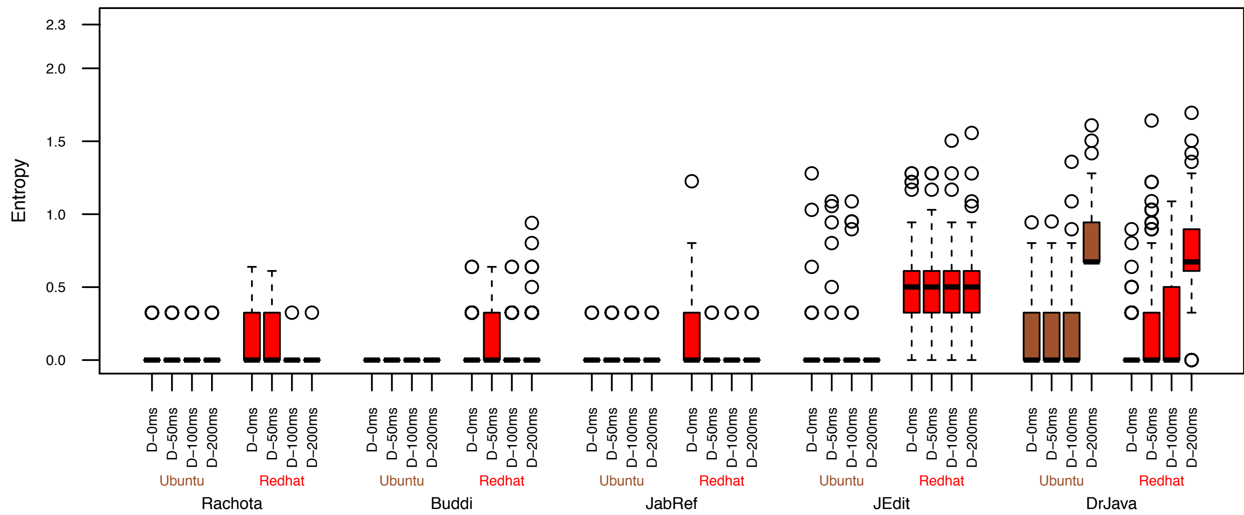


Fig. 4: Entropies of Test Cases with Different Delay Values

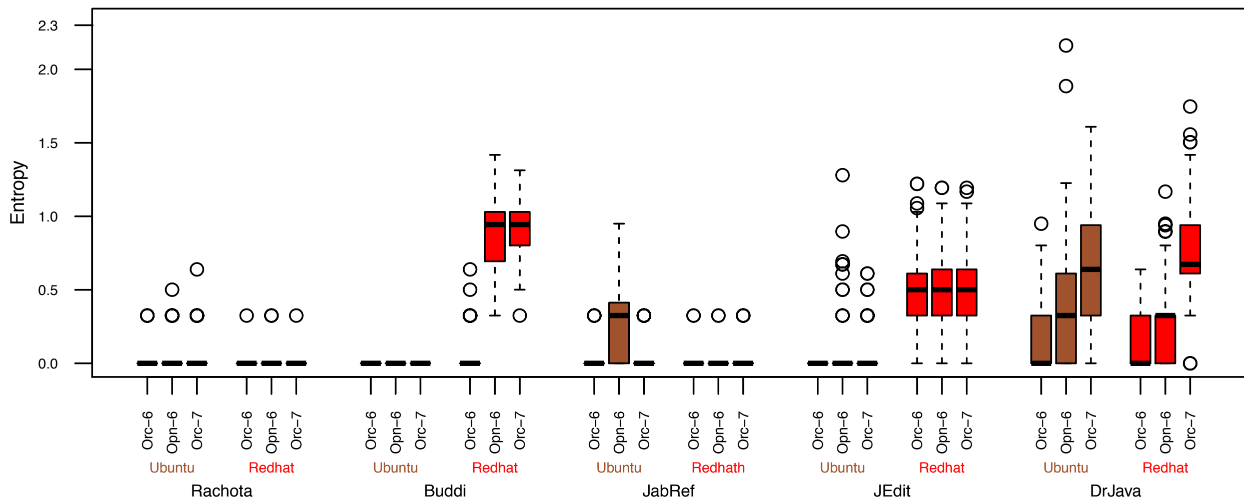


Fig. 5: Entropies of Test Cases with Different JDK Versions

TABLE VI: Entropies of Test Suite Line Coverage for Initial State, Delays and Java Version.

Entropy	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	1.42	0.69	0	0.94	0	0	0.33	0.67	0.80	1.09
D-0ms	0.69	0	0	0.80	0	0.64	0.50	0.64	0.64	1.28
D-50ms	0.94	0	0	1.47	0	0.33	0.33	0	0	0.64
D-100ms	0.69	0	0	1.19	0	0.33	0	1.06	1.17	1.09
D-200ms	0.67	0	0	0.50	0	0.33	0	1.03	0.94	1.36
Opn-6	0.94	0	0	1.47	0.33	0	0.50	1.36	0.95	0.90
Orc-7	1.36	0	0	1.23	0	0.33	0.80	1.61	0.64	0.94

TABLE VII: Average Entropies of Test Case Line Coverage.

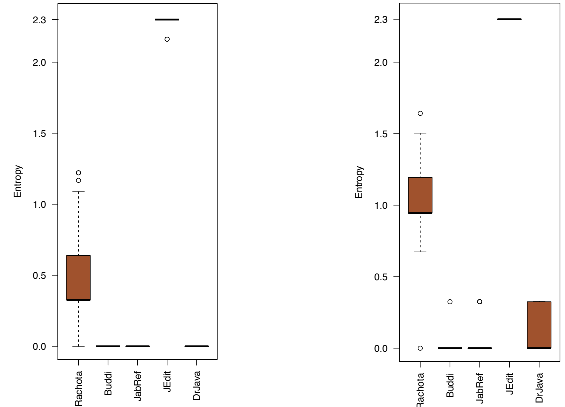
Entropy	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	0.41	0.35	0	0.03	0.00	0.00	0.13	0.24	0.20	0.05
D-0ms	0.01	0.06	0	0.03	0.00	0.08	0.01	0.24	0.06	0.04
D-50ms	0.01	0.06	0	0.05	0.01	0.00	0.01	0.23	0.06	0.13
D-100ms	0.01	0.00	0	0.04	0.01	0.00	0.02	0.23	0.07	0.14
D-200ms	0.00	0.00	0	0.05	0.00	0.00	0	0.24	0.39	0.36
Opn-6	0.02	0.00	0	0.05	0.13	0.00	0.02	0.26	0.16	0.13
Orc-7	0.02	0.00	0	0.05	0.01	0.00	0.02	0.27	0.36	0.40

TABLE VIII: Average Variance-Ranges of Line Coverage

Range	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	85.19	63.41	0	84.59	0.02	4.06	10.06	1.92	14.06	4.89
D-0ms	0.10	0.72	0	94.18	5.69	168.46	0.19	7.63	0.93	1.49
D-50ms	0.12	0.71	0	150.65	2.61	6.46	0.13	1.77	0.32	12.70
D-100ms	0.09	0.03	0	146.83	0.05	5.89	0.14	1.92	0.97	14.12
D-200ms	0.05	0.03	0	114.62	0.03	5.18	0	2.00	82.77	14.63
Opn-6	0.25	0.08	0	43.08	0.80	3.56	0.22	56.75	1.55	13.94
Orc-7	0.38	0.03	0	13.68	0.06	11.29	7.36	61.01	97.45	15.56

TABLE IX: Average/Max/Min Entropies of Invariants across 200 test cases. 0 without decimal places have true 0 entropy; 0.00 indicates a very small entropy rounded to this value.

Entropy	Buddi	Rachota	JEdit	JabRef	DrJava
Best					
Avg	0	0.45	2.30	0	0
Max	0	1.22	2.30	0	0
Min	0	0	2.16	0	0
Unctrl					
Avg	0.00	1.04	2.30	0.00	0.15
Max	0.33	1.64	2.30	0.33	0.33
Min	0	0	2.30	0	0



(a) Best Controlled Environment (b) Uncontrolled Environment

Fig. 6: Entropies of Invariant Groups on Ubuntu

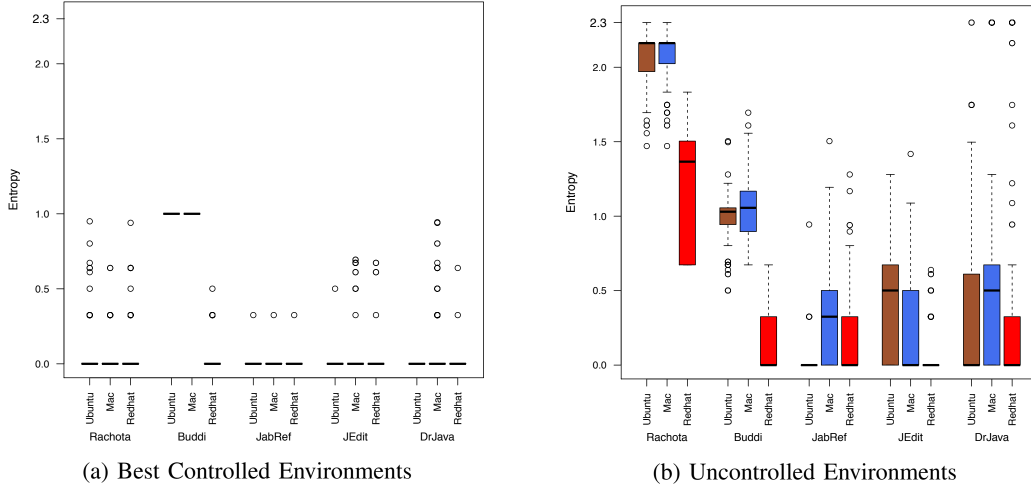
entropy again for this purpose. The average, maximum and minimum entropies of the 200 test cases using the best and uncontrolled configurations are shown in Table IX. As we can see, the invariants seem to be more sensitive to application than to the factors that we are controlling. We can see this if we examine Figure 6. `Rachota` seems to have internal variation not related to these factors, while three of the other applications appear to have almost zero entropy for both the best and uncontrolled runs. For `DrJava`, the factors impact the invariants. In general though the variation is lower than at the code level of interaction.

C. RQ3: GUI State

For this layer, we use *false positives* as our high-level measure of stableness. Our reasoning is that if one were to use the state as an oracle for fault detection, any change detected would indicate a fault. We use the state that is captured during the first (of 10) runs as the oracle, and then then calculate the false positives based on the following formula:

$$\sum FalsePositives/(\sum \#testcases * (\#runs - 1)) * 100.$$

The results of false positives are shown in Table V in the last rows as **FP**. We see as high as a 96% chance (`Rachota` on Ubuntu) for obtaining a false positive. In general in the best configuration we see a very low false positive rate (no



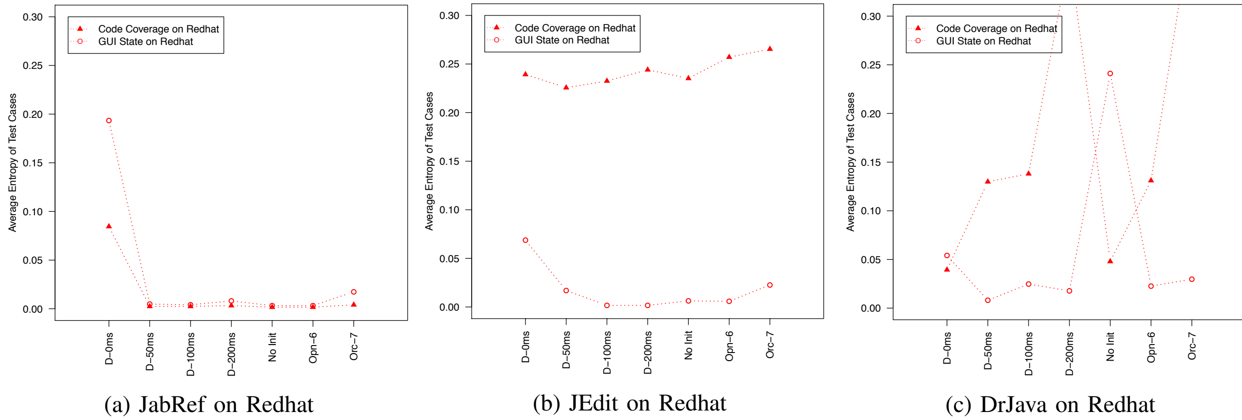
(a) Best Controlled Environments

(b) Uncontrolled Environments

Fig. 7: Entropies of GUI-State Groups of 3 Platforms in Best & Uncontrolled Environments

TABLE X: (Potential) False Positives detected by GUI State Oracle (%)

FP(%)	Rachota		Buddi		JabRef		JEdit		DrJava	
	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat	Ubuntu	Redhat
No Init	48.17	38.78	0.22	2.94	0	1.00	2.22	0.28	10.22	11.33
D-0ms	2.22	3.06	0.06	4.22	1.83	12.28	2.50	4.61	0.22	3.44
D-50ms	2.44	5.94	0.50	1.56	27.33	1.06	0.72	1.17	0	1.61
D-100ms	1.22	9.33	0.22	1.57	1.00	0.61	0.22	0.06	0	3.72
D-200ms	4.44	1.28	0	2.56	0.50	0.72	0.11	0.06	65.22	2.39
Opn-6	3.39	2.61	1.78	22.22	0	0.56	0.39	0.44	0.83	3.44
Orc-7	2.83	3.83	26.61	23.89	0	1.83	17.67	1.22	0.17	3.11



(a) JabRef on Redhat

(b) JEdit on Redhat

(c) DrJava on Redhat

Fig. 8: Average Entropy: Code Coverage vs GUI State

more than 6%), however it is only 0 in a few cases (such as `Buddi` on Mac). The high false positive rate is concerning for experiments that report on new techniques and finding faults. This data also concurs with other recent work on flakiness which uses fault detection as the metric (see Luo et al. [26]).

Figure 7 shows the distribution of entropies for interface state for the 200 test cases. Figure 7(a) shows the entropy rate is very low for all 5 applications in the best controlled configuration, but that in 7(b) there is a higher median entropy when we leave our factors uncontrolled. Finally we show the false positives for the other experimental configurations in Table X. The results show that this level of information is sensitive to the initial state, delay values, and Java version.

To study a possible correlation between the stableness of code coverage and GUI state layers, we plot curves of average entropies in Figures 8a - 8c, for 3 applications on the RedHat platform. We see that the unstableness in code coverage is generally greater than the GUI state. But we don't see a correlation between the two. Figure 8b shows a big difference in code coverage and GUI state entropy; sometimes code coverage is unstable when GUI states are stable. Figure 8c shows this trend.

D. Discussion and Guidelines

We have seen some interesting results during these experiments. First, in almost none of our layers were we able to completely control the information obtained. We saw instances of an application with zero entropy for one or two of the testing configurations, but in general most of our results had a positive entropy meaning at least one test case varied. Some of the greatest variance appears to be related to the delay values and issues with timing which are system and load dependent. Since this might also be an artifact of the tool harness that we used (GUITAR), we wanted to understand the potential threat to validity further, therefore we ran an additional set of experiments using the Selenium test case tool which automates test execution on web applications.

We selected a web application that has existing test cases called `schoolmate` version: 1.5.4. It was used as one of the subjects by Zou et al. in [40]. `SchoolMate` is a PHP/MySQL solution for elementary, middle and high schools. We randomly selected 20 of the test cases and modified the delay values of the test cases to be 0ms, 100ms and 500ms. We ran each 10 times as we did in our other experiments. Six of the twenty test cases failed at least once (and 5 failed in all 10 runs) when the delay value was 0ms or 100ms. This shows us that the delay value is relevant in other tools as well.

One might expect monotonic behavior with respect to the delay, but we did not observe this. Since tools such as GUITAR cannot use human input to advise them about which properties to wait for, they use heuristics to detect an application steady state. The delay value says how long to wait before checking for a steady state. In some applications, there are system events that check periodically for a status, such as timing or reporting events. Since these run at intervals (and the intervals may vary), they create a complex interaction with the delay value, resulting in unpredictable behavior.

We also found some interesting application specific issues such as code which is dependent on the size of memory the application is using, the time of day that the application is run or the time it takes to refresh a screen. For instance, one application had a 30 day update mechanism and we just happened to run one of our early experiments on the 30 day update (and covered new/additional code). Had we been running a real experiment, we might have incorrectly reported that our technique was superior. With respect to operating system differences, we saw three primary causes. Some differences are due to permissions or network connections. For instance, `Rachota` will send requests to a server using the network and these calls were blocked on the Mac experiments. We also saw differences with system load. Certain code is triggered if the application takes longer to perform a particular task or if the resolution is different. This is not necessarily due to differences in the operating systems, but is machine specific. Last, we found code such as in `JabRef` that only runs under specific operating systems enclosed within `if`-blocks.

We did find that we could control a lot of the application starting state and configurations by judicious sharing of the starting configuration files, and that if we heuristically find a good delay value for a specific machine it is relatively stable. The invariant layer was our most stable layer, which indicates that the overall system behavior may not be as sensitive to our experimental factors. Despite our partial success, we believe there is still a need to repeat any tests more than once.

Our overall guidelines are:

1. **Exact configuration and platform states must be reported/shared**. When providing data for experimentation or internal company testing results, the exact configuration information, including the operating system, Java version, harness configurations/parameters and the application starting state needs to be shared.
2. **Run Tests Multiple Times.** For some of the differences observed we do not see an easy solution and expect some variance in our results. Therefore studies should include multiple runs of test cases and report averages and variances of their metrics, as well as sensitivity of their subjects to certain environmental conditions (e.g., resources, date/time).
3. **Use Application Domain Information** to help reduce some variability. For instance, if the application uses time, then clearly this is an environmental variable that needs to be set. But we found others such as memory variances, file permission variances, and simple timing delays within the application that would vary. Knowing what some of these are may allow you to remove that variable code from the evaluation.

V. Conclusions

In this paper we have identified three layers of information that are sensitive to a set of factors when performing automated testing of SUITs. We performed a large study to evaluate their impact. The largest variance was at the bottom and top layer (the code and GUI state layers). When we controlled our factors we saw as little as zero difference in variation, but had entropies as high as 2.3 (where each test run differed) when we did not control these. Despite seeing a lower entropy at the middle, or behavioral level, we did find some cases where invariants differed between runs. Our recommended practice is that all results for testing from the user interface should provide exact configuration and platform as well as tool parameter information. We also recommend running tests more than once and providing both averages and ranges of difference since we are unable to completely control all variation. Finally, we believe application domain information may help to alleviate some parts of the code that are known to be variable based on runtime factors (such as memory allocation or monthly updates) that testing cannot control. In future work we plan to expand our experimentation to other types of applications and look at ways to remove some of the innate application specific variance.

VI. Acknowledgments

We thank M. Roberts for help with Daikon. This work was supported in part by NSF grants CNS-1205472, CNS-1205501 and CCF-1161767.

References

[1] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '04/FSE-12, 2004, pp. 241–251.

[2] A. Marback, H. Do, and N. Ehresmann, "An effective regression testing approach for php web applications," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12, 2012, pp. 221–230.

[3] *Selenium - Web Browser Automation*, 2014 (accessed September 2014). [Online]. Available: http://www.seleniumhq.org/

[4] *Mozmill - Mozilla — MDN*, 2014 (accessed September 2014). [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Mozmill

[5] *Automation Testing Software Tools, QuickTest Professional, (Unified Functional Testing) UFT — HP Official Site*, 2014 (accessed September 2014). [Online]. Available: http://www8.hp.com/us/en/software-solutions/unified-functional-testing-automation/

[6] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, 2009, pp. 69–80.

[7] M. Pradel, "Dynamically inferring, refining, and checking api usage protocols," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, 2009, pp. 773–774.

[8] S. Arlt, C. Bertolini, and M. Schaf, "Behind the scenes: An approach to incorporate context in GUI test case http://comet.unl.edu/generation," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 222–231.

[9] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 162–171.

[10] A. Marchetto and P. Tonella, "Using search-based algorithms for ajax event sequence generation during testing," *Empirical Softw. Engg.*, vol. 16, no. 1, pp. 103–140, Feb. 2011.

[11] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 210–220.

[12] A. Marchetto, F. Ricca, and P. Tonella, "An empirical validation of a web fault taxonomy and its usage for web testing," *J. Web Eng.*, vol. 8, no. 4, pp. 316–345, Dec. 2009.

[13] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing GUI applications," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., 2009, vol. 5885, pp. 69–87.

[14] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an android application," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, March 2011, pp. 377–386.

[15] Z. Yu, H. Hu, C. Bai, K.-Y. Cai, and W. Wong, "GUI software fault localization using N-gram analysis," in *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, Nov 2011, pp. 325–332.

[16] B. U. Maheswari and S. Valli, "Algorithms for the detection of defects in GUI applications." *Journal of Computer Science*, vol. 7, no. 9, 2011.

[17] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of GUI test-case prioritization using weight-based methods," *Journal of Systems and Software*, vol. 83, no. 4, pp. 646 – 659, 2010.

[18] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 1345–1348.

[19] ——, "Coverage and fault detection of the output-uniqueness test selection criteria," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 181–192.

[20] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09, 2009, pp. 901–910.

[21] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, 2011, pp. 3–12.

[22] T. Yu, W. Srisa-an, and G. Rothermel, "An empirical comparison of the fault-detection capabilities of internal oracles," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, Nov 2013, pp. 11–20.

[23] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated replay and failure detection for web applications," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 253–262.

[24] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, and J. Noujaim, "Guicop: Specification-based gui testing," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12, 2012, pp. 747–751.

[25] P. Tramontana, S. De Carmine, G. Imparato, A. R. Fasolino, and D. Amalfitano, "A toolset for gui testing of android applications," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12, 2012, pp. 650–653.

[26] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Foundations of Software Engineering (FSE)*, November 2014, to appear.

[27] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: Models, tools, and controlling flakiness," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 1479–1480.

[28] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2014, to appear.

[29] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 385–396.

[30] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *International Conference on Software Testing, Verification and Validation (ICST)*, April 2010, pp. 245–254.

[31] *COMET - Community Event-based Testing*, 2014 (accessed September 2014). [Online]. Available: http://comet.unl.edu/

[32] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *International Symposium on Software Testing and Analysis, ISSTA*, July 2008, pp. 75–85.

[33] X. Yuan, M. Cohen, and A. M. Memon, "Covering array sampling of input event sequences for automated GUI testing," in *International Conference on Automated Software Engineering*, 2007, pp. 405–408.

[34] "GUITAR – a GUI Testing frAmewoRk," website, 2009, http://guitar.sourceforge.net.

[35] 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.

[36] *Cobertura*, 2014 (accessed September 2014). [Online]. Available: http://cobertura.github.io/cobertura/

[37] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[38] 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*, Nov 2013, pp. 257–267.

[39] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 573–583.

[40] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao, "Virtual DOM coverage for effective testing of dynamic web applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 60–70.