Using Transient/Persistent Errors to Develop Automated Test Oracles for Event-driven Software

Atif Memon† and Qing Xie
Department of Computer Science (†and Institute for Advanced Computer Studies)
University of Maryland, College Park, Maryland 20742, USA
{atif,qing}@cs.umd.edu

Abstract

Today's software-intensive systems contain an important class of software, namely event-driven software *(***EDS***). All EDS take events as input, change their state, and (perhaps) output an event sequence. EDS is typically implemented as a collection of event-handlers designed to respond to individual events. The nature of EDS creates new challenges for test automation. In this paper, we focus on those relevant to automated* test oracles*. A test oracle is a mechanism that determines whether a software executed correctly for a test case. A test case for an EDS consists of a sequence of events. The test case is executed on the EDS, one event at a time. Errors in the EDS may "appear" and later "disappear" at several points (e.g., after an event is executed) during test case execution. Because of the behavior of these transient (those that disappear) and persistent (those that don't disappear) errors, EDS require complex and expensive test oracles that compare the expected and actual output multiple times during test case execution. We leverage our previous work to study several applications and observe the occurrence of* persistent/transient *errors. Our studies show that in practice, a large number of errors in EDS are transient and that there are specific classes of events that lead to transient errors. We use the results of this study to develop a new test oracle that compares the expected and actual output at strategic points during test case execution. We show that the oracle is effective at detecting errors and efficient in terms of resource utilization.*

1 Introduction

Motivation: Today's large, complex software-intensive systems contain an important class of software, namely *event-driven software* (**EDS**). Several researchers have modeled the event-driven nature of different type of software including simulation software [5], component-based software [34], web applications [38], graphical-user interfaces (GUI) [12], visualization software [19], network protocols [33], device drivers [16], database applications [18], and embedded and middleware software [31]. What distinguishes an EDS from conventional software is the EDS's event-driven model. All EDS take user-generated and/or system-generated events (e.g., simulation control events, messages, mouse-clicks) as input, change their state, and (perhaps) output an event sequence. EDS is typically implemented as a collection of event-handlers designed to respond to individual events. The nature of EDS creates new challenges for quality assurance activities such as test automation.

Challenges: Testing EDS is complex because of several factors; we focus on two relevant to *test oracles*. A test oracle is used to determine whether the application under test (AUT) executed as expected [2]. The test oracle may either be automated or manual; in both cases, the actual output is compared to a presumably correct expected output. The first challenge stems from the representation of an EDS's test-case [25]; a test case for an EDS consists of a sequence of events. The test case is executed on the EDS, one event at a time. Errors in the EDS may "appear" and later "disappear" at several points (e.g., after an event is executed) during test case execution, requiring the need for complex test oracles that compare the expected and actual output multiple times during test case execution. Second, it is difficult to determine how much expected output to use during testing [23].

Preliminary observations: In previous work, we developed several types of test oracles for GUIs and empirically showed their relative strengths, weaknesses, and costs [23, 24]. Borrowing terminology from Richardson et al. [30], we defined a test oracle to contain two parts: *oracle information* that is used as the expected output and an *oracle procedure* that compares the oracle information with the actual output. We showed that the test oracle contributes significantly to test effectiveness and cost. In [23] we created different test oracles primarily by varying the oracle information. We almost ignored the frequency of comparison except at two extreme points during test case execu-

tion: (1) O_{all} – "check for `equality` of the oracle information and actual output after each event" and (2) O_{last} – "check for `equality` of oracle information and actual output after the last event" of the test case. Since then, we have developed algorithms to generate different types of long test cases. We have found that a large number of test cases detect transient errors (*informally*, these errors "disappear" during test case execution). In order to detect such errors, we invoke the oracle procedure after each event in the test case, i.e., we use O_{all}. Not only is the comparison process expensive, it also requires that we maintain the oracle information after each event; for long test cases, this is significantly large. O_{last}, although cheaper, would miss such errors. What we need is a test oracle that is comparable to O_{all} in terms of fault detection effectiveness and O_{last} in terms of cost.

Approach: In this paper, we leverage our previous work to study several applications and observe the occurrence of persistent and transient errors.[1] We then use the results of our study to develop a new test oracle that is effective at detecting errors and efficient in terms of resource utilization. Our empirical studies show that (1) in practice, a large number of errors in GUIs are transient, (2) there are specific classes of GUI events that lead to transient errors, and (3) we can create a new automated test oracle (O_{new}) that is comparable to O_{all} in terms of fault-detection effectiveness and O_{last} in terms of cost.

Contributions: The contributions of this paper include:

1. Model of persistent and transient errors for GUIs.
2. Empirical study showing characteristics of errors.
3. Relationship between GUI faults and errors.
4. New automated test oracle for GUIs.

Structure of the paper: In Section 2, we present a brief overview of our previous work. In Section 3 we define the terms persistent and transient errors. Our empirical studies are described in Section 4. In Section 5 we briefly discuss related work and finally conclude in Section 6 with a discussion of current and future research.

2 GUI Test Oracles Overview

A high-level overview of our GUI test oracle is shown in Figure 1. The *oracle information generator* automatically derives the *oracle information* (expected state) using either a formal specification of the GUI as described in our earlier work [24] or by using a "correct" version of the software [36, 37] (as described in Section 4). Likewise, the *actual state* is obtained from an *execution monitor*. The execution monitor may use any of the techniques described

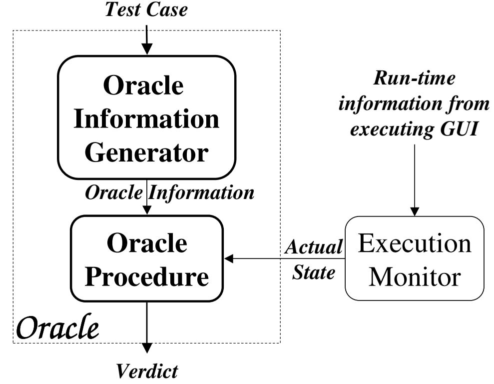


Figure 1. An Overview of the GUI Oracle.

in [24], such as screen scraping and/or querying to obtain the actual state of the executing GUI. An *oracle procedure* then automatically compares the two states and reports GUI errors.

Intuitively, a *GUI error* is a mismatch between the actual GUI state and oracle information. We now briefly[2] describe the representation of a GUI state.

We model a GUI as a set of *widgets* $W = \{w_1, w_2, ..., w_l\}$ (e.g., buttons, panels, text fields) that constitute the GUI, a set of *properties* $P = \{p_1, p_2..., p_m\}$ (e.g., background color, size, font) of these widgets, and a set of *values* $V = \{v_1, v_2, \ldots, v_n\}$ (e.g., red, bold, 16pt) associated with the properties. Each GUI will contain certain types of widgets with associated properties. At any point during its execution, the GUI can be described in terms of the specific widgets that it currently contains and the values of their properties.

The set of widgets and their properties is used to create a model of the *state* of the GUI.

Definition: The *state* of a GUI at a particular time t is the set S of triples $\{(w_i, p_j, v_k)\}$, where $w_i \in W$, $p_j \in P$, and $v_k \in V$. □

In earlier work [23], we used the state definition to develop four types of oracle information in increasing level of detail and cost: *widget*, *active window*, *visible windows*, and *all windows*. The oracle procedure too had several increasing levels of complexity and cost: "check for equality of *widget*, *active window*, *visible window*, *all windows* after each event" and "check *all windows* after the last event" of the test case. Combining the oracle information and oracle procedures gave us 11 different types of oracles. In this paper, for lack of space we consider two of the 11 oracles; we define oracles O_{all} and O_{last} as follows.

Definition: Test oracle O_{all} compares the set of all triples

[1] Note that the same (and similar) terms have been used by other researchers [3, 7, 10, 14, 20, 21, 35]. Later, we formally define them for GUIs in this paper.

[2] The reader is referred to [23, 24] for details.

for all widgets of all visible GUI windows with the corresponding expected state after each event in the test case. Note that visibility is a property of a window, which can be set, for example, by invoking the `SetVisible()` method in Java. Windows that are partially or fully hidden by other overlapping windows are also considered to be visible as long as this property is set. □

Definition: Test oracle O_{last} compares the set of all triples for all widgets of all visible GUI windows with the corresponding expected state after the last event in the test case. □

In our empirical studies, we compare our new test oracle to O_{all} and O_{last}. To help create the test oracle, we study the relationship between classes of GUI events and GUI errors. We now briefly describe the event classification. At all times during interaction with the GUI, the user interacts with events within a modal dialog. This modal dialog consists of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly by X. The modal dialog remains in place until X is explicitly terminated. The first class of events, called **restricted-focus events** open *modal windows*. For example, `Set Language` in MS Word is a restricted-focus event. The second class, called **unrestricted-focus events** open *modeless windows*. For example, `Replace` in MS Word is an unrestricted-focus event. **Termination events** close modal windows; common examples include `Ok` and `Cancel`.

The GUI contains other types of events that do not open or close windows but make other GUI events available. These events, called **menu-open events** are used to open menus. They expand the set of GUI events available to the user. Menu-open events do not interact with the underlying software. Note that the only difference between menu-open events and unrestricted-focus events is that the latter open windows that must be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. For example, in MS Word, `File` and `SendTo` are menu-open events. Finally, **system-interaction events** interact with the underlying software to perform some action; common examples include the `Copy` event used for copying objects to the clipboard.

The above classification will be used in the empirical study to identify strategic points in the test case at which comparison between expected and actual states would lead to GUI error detection. We formally define GUI errors next.

3 GUI Errors

Intuitively, a GUI error is a mismatch between the actual and expected states.

Definition: A *GUI error* occurs during execution of a test case, if $A_i \neq S_i$, for any event e_i in the test case. S_i is the oracle information and A_i is the actual state obtained after the execution of e_i. □

During test case execution, the mismatch may persist until after the last event in the test case. Such errors are called persistent and may be detected by O_{last}.

Definition: A *persistent GUI error* occurs during execution of a test case, if $A_i \neq S_i$, for all events e_i in a test case of length n ($j \leq i \leq n$ for some j). □

On the other hand, many mismatches may not persist until the last event in the test case, i.e., they disappear after the execution of some event in the test case. Note that such events would be missed by O_{last} but would be detected by O_{all}.

Definition: A *transient GUI error* occurs during execution of a test case, if $A_i \neq S_i$, for an event e_i in the test case, where S_i is the oracle information and A_i is the actual state and $A_j == S_j$, for some $i < j \leq n$. □

O_{all} and O_{last} may be viewed as extreme points in a spectrum of test oracles ordered by frequency of comparison. Although O_{all} is able to detect the maximum number of errors, it performs expensive and redundant comparisons. O_{last}, on the other hand, contains only one comparison, causing it to miss transient errors. The key to designing an efficient and effective test oracle is to compare the expected and actual states at strategic points during test case execution. Our empirical study, described next, will help us identify such strategic points.

4 Empirical Studies

In this section, we present details of empirical studies that demonstrate the existence and characteristics of persistent/transient errors in GUI programs[3]. We show that certain classes of GUI events contribute to transient errors, and that we can use these classes to design effective automated test oracles. We use oracles O_{all} and O_{last} as controls for these studies. More specifically, we are interested in answering the following questions:

1. In practice, do persistent/transient errors occur in GUIs? What percentage of errors are transient?
2. Are there specific classes of GUI events that lead to transient errors?
3. Can we build a new automated test oracle (O_{new}) that is comparable to O_{all} in terms of fault-detection effectiveness and O_{last} in terms of cost?

[3] All the tools, artifacts, and data, along with the study process diagram are available for download at the authors' web-site.

Subject Application	Windows	Widgets	LOC	Classes	Methods	Branches
TerpWord	11	132	4893	104	236	452
TerpSpreadSheet	9	165	12791	125	579	1521
TerpPaint	10	220	18376	219	644	1277
TOTAL	30	517	36060	448	1459	3250

Table 1. Our Subject Applications.

4.1 Study 1: Investigating Error Characteristics

To answer Questions 1 and 2, we perform the following steps in this study:

1. Choose software subjects with GUI front-ends,
2. Generate test cases,
3. Generate oracle information,
4. Use fault seeding techniques to artificially seed faults in the software subjects,
5. Execute all test cases on the software subjects. During execution, compare the actual GUI state to the oracle information. (Note that state is represented by all the widgets in the visible windows of the GUI.) Measure the number of faults detected, and the position in the test case when the fault was detected. Measure the space required to store the oracle information and the time required for each comparison.

Step 1, Subject Applications: The subject applications for our studies are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[4] and consists of six applications out of which we use three – TerpPaint (an image editing/manipulation program) TerpSpreadSheet (a spreadsheet application), and TerpWord (a small word-processor), They have been implemented using Java. Table 1 summarizes the characteristics of these applications. The number of widgets listed in the table are the ones on which user events can be executed (e.g., text-labels are not included). Note that these applications are fairly large with complex GUI front-ends.

Step 2, Test Cases: We generated GUI test cases of varying lengths using *event-flow graphs* (EFG) [23]. A detailed discussion of this approach is beyond the scope of this paper. Intuitively, the EFG models all possible GUI event sequences. Nodes in the EFG represent events. An edge from node n_x to n_y is used to show that the event represented by n_y *may be* executed by a user immediately after the event represented by n_x. Test cases are generated by traversing the EFG using various graph-walking algorithms and enumerating all the nodes encountered. We performed a random walk of the EFG, which was designed to give uniform coverage of all the events in the GUI. Figure 2 shows the event distribution of all the test cases showing that we had good event coverage. The figure shows multiple column graphs on one axis. The columns are grouped by application. The x-axis shows all the events in each application. The y-axis shows the number of times a particular event was executed by a test case. To allow visual comparison of the total number of events in each application, we have used the same fixed-width for each application. Since TerpPaint has the maximum number of events, its column is filled completely. The other applications have missing areas. However, we noted that all the events in each application were covered. We generated 819 test cases of varying lengths. Figure 3 shows the length distribution of our test cases.

[4]www.cs.umd.edu/users/atif/TerpOffice

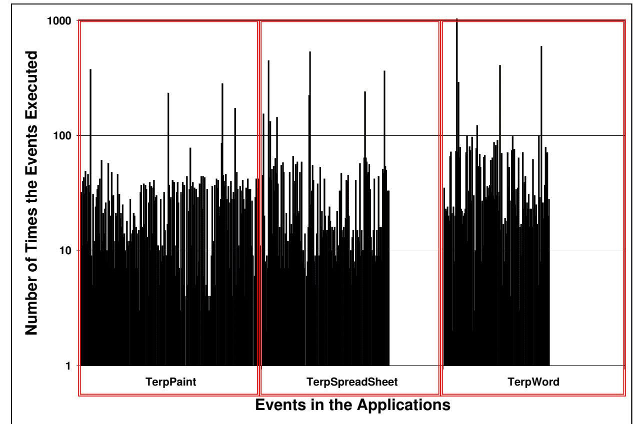


Figure 2. Event Distribution.

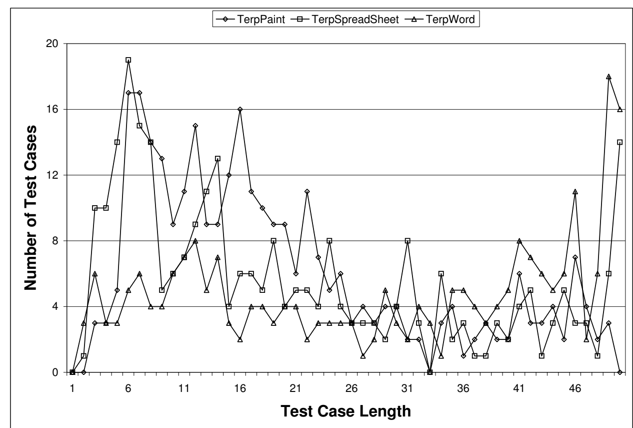


Figure 3. Test Case Length Distribution.

Step 3, Generating Oracle Information: We used an approach that we call *execution extraction* to obtain the oracle information. During this process, each test case was executed on the correct version (with no seeded faults) of the subject applications and the GUI state was extracted

and stored as oracle information. We employed platform-specific technology such as Java API to obtain this information.

Step 4, Fault Seeding: Fault seeding is a common technique to introduce known faults into programs. It has several applications; in this paper, we use it to create mutants of code. We create test cases and execute them on the mutants. We then study the properties of the test cases that were successful at killing the mutants. Care is taken so that the artificially seeded faults are similar to faults that occur in programs due to mistakes made by developers [15, 26].

We adopted an observation-based approach to seed GUI faults, i.e., we observed "real" GUI faults in real applications. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[5] was used by the developers to report and track faults in TerpOffice version 1.0 while they were working to extend its functionality and developing version 2.0. The reported faults are an excellent representative of faults that are introduced by developers during implementation.

We created 200 faulty versions for each software. Only one fault was introduced in each version. This model is useful to avoid fault-interaction, which can be a thorny problem in these types of studies and also simplifies the computation of the number of faults detected; now we can simply count the faulty versions that led to an error.

Since this is a controlled study, we were careful to define classes of faults for seeding. We now summarize the these classes in one short statement and provide an example of each in Table 2. Note that the row number in the table corresponds to the numbering below.

1. Modify relational operator (>, <, >=, <=, ==, !=);
2. Invert the the condition statement;
3. Modify arithmetic operator (+, -, *, /, =, ++, –, +=, -=, *=, /=);
4. Modify logical operator (&&, ||);
5. Set/return different boolean value (`true`, `false`);
6. Invoke different (syntactically similar) method;
7. Set/return different attributes;
8. Modify bit operator (&, |, ∧, &=, !=, ∧=);
9. Set/return different variable name;
10. Set/return different integer value;
11. Exchange two parameters in a method;
12. Set/return different string value.

Figure 4 shows the distribution of the 600 faults that were seeded in the subject applications. The x-axis shows the type/class of fault, as defined earlier, and the y-axis shows the number of faults seeded. Note that while every effort was made to seed faults of all types uniformly, some faults were easier to seed than others because of the opportunities in the code. For example, fault Type 1, i.e., *modify relational operator* was seeded 323 times because of the large number of relational operators in the code.

[5] `http://bugs.cs.umd.edu`

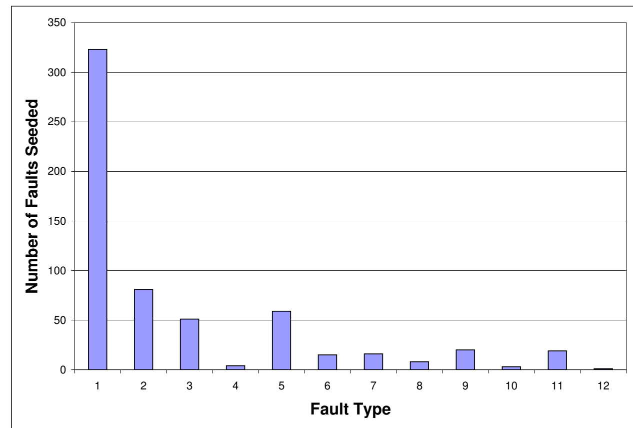


Figure 4. Different Type of Faults Frequency Distribution.

Step 5, Test Execution: We designed a test executor that is capable of executing an entire test suite automatically on the AUT. It performs all the events in each test case and compares the actual output with the expected output. Events are triggered on the AUT using the Java API *doClick*.

The execution included launching the application under test, replaying GUI events from a test case on it and analyzing the resulting GUI states. The analysis consisted of recording the actual GUI states of the faulty version and determining the result of the test case execution. The test cases executed on four machines (Pentium 4, 2.2GHz, each with 256MB RAM) simultaneously for more than a week. Although much of the execution was automated, we had to restart some machines (and test scripts) because of problems with the JVM.

4.1.1 Threats to Validity

Threats to external validity are conditions that limit the ability to generalize the results of our studies to industrial practice. We have used four Java applications are our subject programs. Although they have different types of GUIs, this does not reflect a wide spectrum of possible GUIs that are available today. Although our model of the GUI maintains uniformity between Java and Win32 applications, the results may vary for Win32 applications.

Threats to internal validity are conditions that can affect the dependent variables of the study without the researcher's knowledge. We have used an observation-based approach for seeding faults in the GUI applications. This may have affected the detection of faults by the test cases. Faults not exercised by any test case will go undetected. We made an

<table>
<tr><th>Fault Type</th><th>Original Code</th><th>Mutated Code</th></tr>
<tr><td>1</td><td>if (this.row > y.row)</td><td>if (this.row < y.row)</td></tr>
<tr><td>2</td><td>if (newValue)</td><td>if (!newValue)</td></tr>
<tr><td>3</td><td>prev = index+1;</td><td>prev = index-1;</td></tr>
<tr><td>4</td><td>if (done || border == null ||</td><td>if (done && border == null ||</td></tr>
<tr><td>5</td><td>if(contentArea.closeDocument(true))</td><td>if(contentArea.closeDocument(false))</td></tr>
<tr><td>6</td><td>int rowLimit = model.getRowCount() - 1;</td><td>int rowLimit = model.getColumnCount() - 1;</td></tr>
<tr><td>7</td><td>int style = Font.ITALIC;</td><td>int style = Font.BOLD;</td></tr>
<tr><td>8</td><td>style |= Font.BOLD;</td><td>style &= Font.BOLD;</td></tr>
<tr><td>9</td><td>buttonPanel.add(okButton);</td><td>buttonPanel.add(cancelButton);</td></tr>
<tr><td>10</td><td>int size = 12;</td><td>int size = 15;</td></tr>
<tr><td>11</td><td>tmp = data.substring(0, i2);</td><td>tmp = data.substring(i2,0);</td></tr>
<tr><td>12</td><td>if(findString.equals("")) { return; }</td><td>if(findString.equals(" ")) { return; }</td></tr>
</table>

Table 2. Classes of Seeded Faults.

effort to make the faults as close as possible to naturally occurring faults. Some of these faults might not manifest themselves through the GUI.

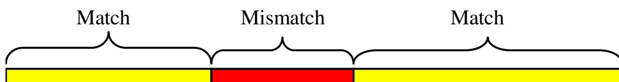
Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this study one of our measures of cost is time. Since GUI programs are often multi-threaded, and interact with the windowing system's manager, our experience has shown that the execution time varies from one run to another. One way to minimize the effect of such variations is to run the studies multiple number of times and report average time.

The results of our studies, presented next, should be interpreted keeping in mind the above threats to validity.

4.2 Study 1 Results

Number of Persistent/Transient Errors: We immediately noticed the large number of transient errors found during execution of our test cases. Figure 5 shows a column graph with the total number of persistent/transient errors that our test cases found. The x-axis shows two columns per application. The two columns represent persistent and transient errors. The y-axis shows the number of errors found.

Detailed View of Errors: We next wanted to see parts of test cases that caused the expected and actual states to mismatch. For compactness, we will use one graph per application. Figure 6 shows the results for TerpPaint. The x-axis shows the event number (i.e., its position in the sequence) in the test case. The y-axis represents test cases that successfully killed a mutant. For each test case, we have a line, with 2 levels of shading. The dark band shows the events after which the actual and expected states mismatched. The light band shows the event number after which the actual and expected states matched:



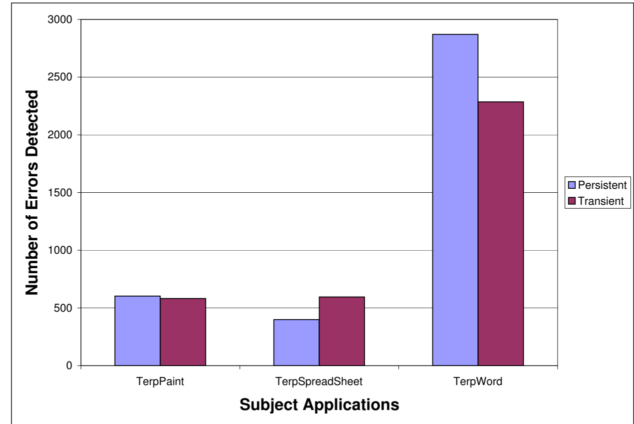


Figure 5. Number of Errors.

If a test case killed more than one mutant, it is counted more than once.

We have sorted the test cases to show the dark/light bands clearly. We note that many test cases have small areas of mismatch. In all cases where the test case ends in a light band (i.e., a match) , the test oracle O_{last} would have failed to report an error. Similar results are seen for TerpSpreadSheet and TerpWord in Figures 7 and 8 respectively.

Events that Reveal/Conceal Errors: Having observed that long test cases, during execution, can transit frequently between matching and mismatching, we wanted to see if there were certain classes of events that caused the transitions. We mined our execution data to find events that led from a match to a mismatch and vise versa. We summarize our results in Figure 9. The figure shows a column graph with event types on the x-axis. The y-axis shows the number of times an event type led to a transition. As seen in the graph, termination, restricted-focus, and system-interaction events cause the maximum number of transitions. A test oracle that compares the expected and actual states of the GUI at these events is most likely to report transient errors.

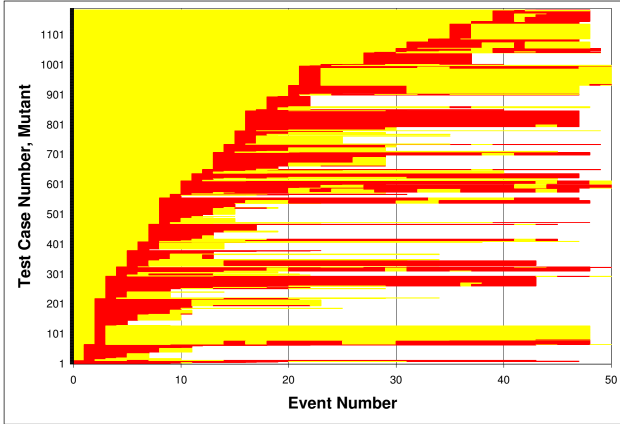


Figure 6. Errors for TerpPaint.

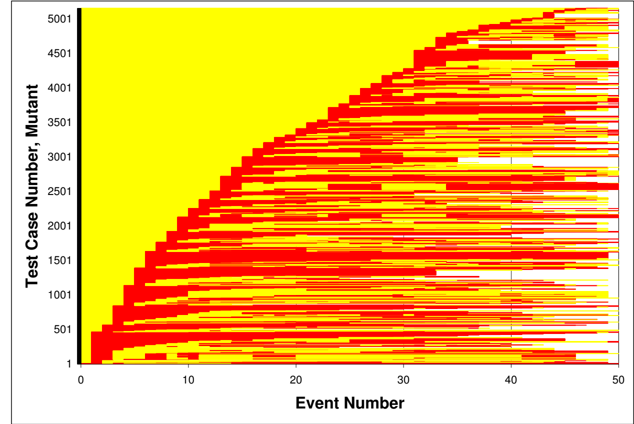


Figure 8. Errors for TerpWord.

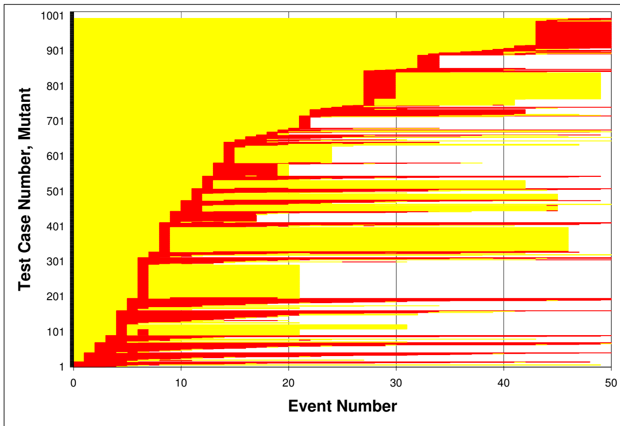


Figure 7. Errors for TerpSpreadSheet.

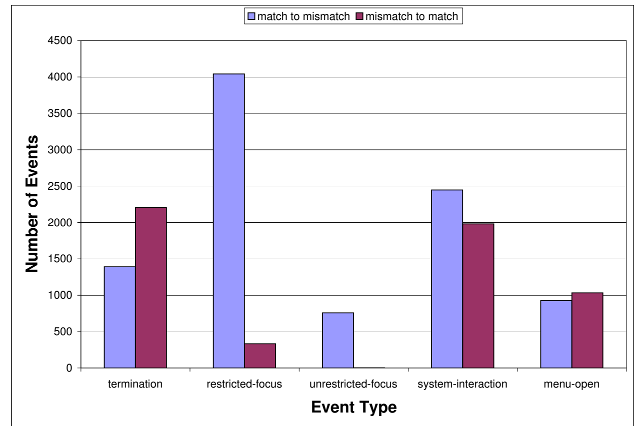


Figure 9. Event Classes and Error Types.

We manually examined the test case execution reports and observed that object creation and destruction play an important role in transient errors. Examples of such objects for GUIs include windows, menus, widgets, etc. A restricted-focus event that opens a erroneous window will cause an error to be detected by the test oracle. On the other hand, a termination event that destroys a window will close the erroneous window, resulting in a match between expected and actual states.

4.3 Study 2: Creating and Studying O_{new}

We used the results of Study 1 to extend our research and design a new oracle called O_{new}. We specifically wanted to compare O_{new} with O_{last} and O_{all}in terms of error detection and cost. We simulated three testers, each equipped with one of the oracles and a set of test cases that were executed on all mutants.[6] A test case was terminated as soon as it detected an error. The time and storage was measured.

O_{new} was created by modifying O_{last}. Oracle information was generated for termination and restricted-focus events. The oracle procedure was modified to compare the expected and actual states at these points. We chose not to use system-interaction events because our test cases contain a large number of these events, i.e., had we compared at system-interaction events, O_{new} would have degraded to O_{all} in terms of cost because it would have required frequent comparison.

4.4 Study 2 Results

Error detection: Figure 10 shows three colums for O_{new}, O_{all}, and O_{last} respectively for each application. The y-axis shows the number of errors reported. As seen from the

[6]Note that, to reduce threats to validity, we performed Study 2 on two applications that were not part of Study 1 – the results are not presented in this paper due to lack of space.

graph, O_{new} is able to report almost as many errors as O_{all}. O_{last} performed worst because it missed all the transient errors.

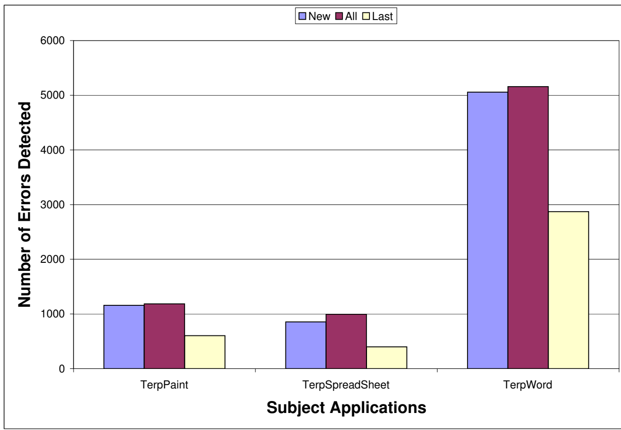


Figure 10. Error detection of O_{new}.

We also wanted to determine how O_{new} compared to O_{all} and O_{last} with respect to storage space and time.

Space: Figure 11 shows three colums for O_{new}, O_{all}, and O_{last} respectively for each application. The y-axis (log scale) shows the space required in MB for all the test cases. As seen from the graph, O_{new} requires significantly less space than O_{all} and (as expected) more space than O_{last}.

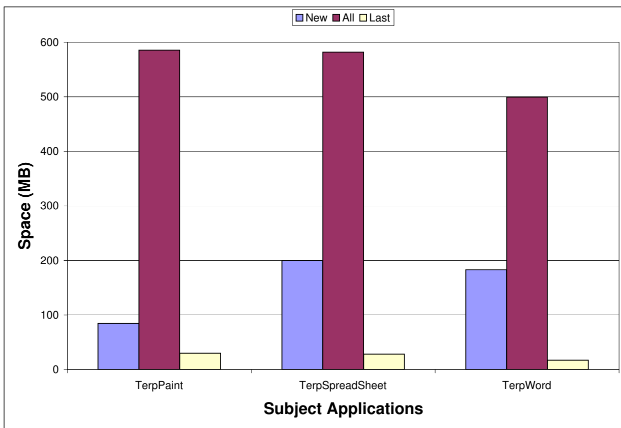


Figure 11. Space Required for O_{new}.

Time: Figure 12 shows three colums for O_{new}, O_{all}, and O_{last} respectively for each application. The y-axis shows the time required in seconds for all the test cases. As seen from the graph, O_{new} requires significantly less time than O_{all} and more time than O_{last}.

The results of this study showed that O_{new} was almost as effective as O_{all} in terms of error detection. However, it was much cheaper to execute and store.

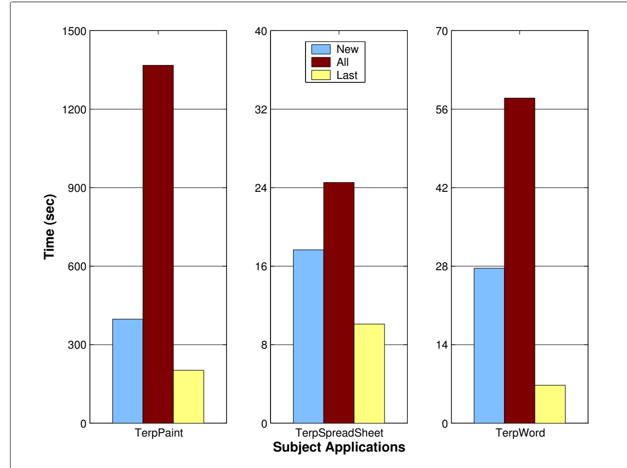


Figure 12. Time Required for O_{new}.

4.5 Faults and Errors

In a study such as ours, the artificially seeded faults play an important role in determining its outcome. We wanted to see if particular classes of seeded faults lead to persistent/transient errors, i.e., whether we has inadvertently favored any error type. For all three applications, a column graph in Figure 13 shows for our seeded fault types the total number of times each led to a persistent or transient error. The x-axis shows all fault classes. For each fault class we have two columns, representing persistent and transient faults respectively. The y-axis is the total number of times, for all test cases, the fault manifested itself as an error.

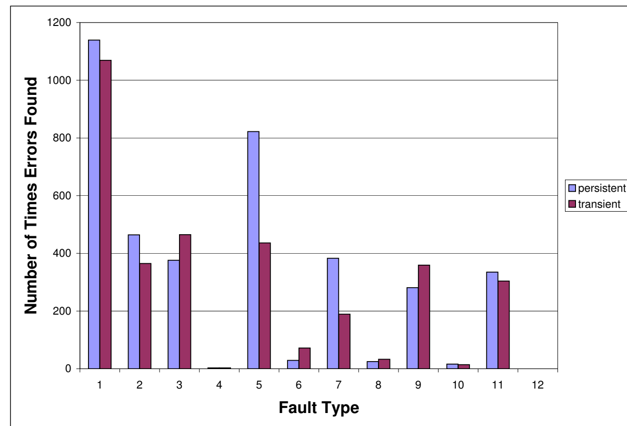


Figure 13. Faults classes and Error Types.

We note that each fault type led to both persistent and transient errors. Hence our seeded faults did not influence the results of this study.

5 Related Work

Although there is no prior work that directly addresses the research presented in this paper, several researchers and practitioners have discussed concepts that are relevant to its specific parts. We discuss the following broad categories: *persistent and transient errors*, *methods to specify oracles*, *reference testing*, and *GUI oracles*.

Persistent and transient errors: Very few researchers have applied the concepts of persistent and transient errors for software testing. Cheynet et al. present an automated approach to detect transient errors for safety-critical software [7]. However, many researchers have used persistent and transient errors (also called persistent and transient faults) for fault-tolerant software [3, 10, 35]. Schreyjak defined two types of faults for component-based software [32] – persistent faults are those that are reproducible in certain system states; transient faults occur occasionally during execution [14]. Similarly, Laprie et al. [20, 21] proposed fault classes according to their persistence. If the faults are permanent, they are classified as persistent/solid; otherwise if they are temporary, they are classified as transient/intermittent/soft/volatile.

Methods to specify oracles: Software systems rarely have an automated oracle [9, 27, 29, 30]. In most cases, the expected behavior of the software is assumed to be provided by the test designer. It can be specified in several ways: (1) the form of a table of pairs *(actual output, expected output)* [27], (2) as temporal constraints that specify conditions that must not be violated during software execution [8, 9, 29, 30], and (3) as logical expressions to be satisfied by the software [11]. This expected behavior is then used by the verifier by either performing a table lookup [27], FSM creation [9, 17], or boolean formula evaluation [11] to determine the correctness of the actual output. In few cases, formal specifications have also been used to specify and generate test oracle information [1,4,13,28]. A runtime assertion checker is used as the oracle procedure [6].

Reference testing: A popular alternative to manually specifying the expected output is to perform reference testing [36, 37]. Actual outputs are recorded the first time the software is executed. The recorded outputs are later used as expected output for regression testing. This is a popular technique used for regression testing of GUI-based software. Capture/Replay tools such as those that are part of GUITAR (`http://guitar.cs.umd.edu`) capture bitmap images of GUI objects into a test script. These bitmaps are then used as test oracles to compare against actual output during regression test cases execution. However, the problem associated with such tools is even a slight change in the GUI's layout will make the bitmap/test oracle obsolete [25].

GUI oracles: Finally, our own earlier work described automated GUI test oracles for the PATHS (Planning Assisted Tester for grapHical user interface Systems) system [22,24]. PATHS uses AI planning techniques to automate testing for GUIs. The oracle described in PATHS uses a formal model of a GUI to automatically derive the oracle information for a given test case.

6 Conclusions

In this paper, we designed a new test oracle for GUI software. We used the observation that GUI errors "appear" and later "disappear" at several points (e.g., after an event is executed) during test case execution. We defined two types of GUI errors – *transient*, those that disappear and *persistent*, those that don't disappear. We leveraged our previous work to study several applications and observe the occurrence of persistent/transient errors. Our studies showed that in practice, a large number of errors in GUIs are transient and that there are specific classes of events that lead to transient errors. We used this study to develop our new test oracle that compares the expected and actual output at strategic points during test case execution. We showed that the oracle is effective at detecting errors and efficient in terms of resource utilization.

We are currently extending our approach to other EDS and object-oriented programs in which objects are created and destroyed during testing. We are also examining the impact of different test oracles on false positives that are reported during GUI testing.

References

[1] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.

[2] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[3] A. Bondavalli, S. Chiaradonna, and F. D. Giandomenico. Efficient fault tolerance: an approach to deal with transient faults in multiprocessor architectures. In *Proceedings of 1994 International Conference on Parllel and Distributed Systemsm*, pages 354–359. IEEE, 1994.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications.

[5] A. Carloganu and J. Raguideau. Claire: An event-driven simulation tool for test and validation of software programs. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 538. IEEE Computer Society, 2002.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01–12, 2001.

[7] P. Cheynet, B. Nicolescu, R. Velazco, M. rebaudengo, M. S. Reorda, and M. Violante. Experimentally evaluating an automatic approach for generating safety-critical software with

respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236, 2000.

[8] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 of *ACM Software Engineering Notes*, pages 106–117, New York, Oct.16–18 1996. ACM Press.

[9] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 140–153, Dec. 1994.

[10] L. Dong, R. Melhem, D. Mossé, S. Ghosh, W. Heimerdinger, and A. Larson. Implementation of a transient-fault-tolerance scheme on DEOS - A technology transfer from an academic system to an industrial system. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, pages 56–67, Washington - Brussels - Tokyo, June 1999. IEEE.

[11] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering*, pages 267–276. ACM Press, May 1999.

[12] M. B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 244–261. Springer / ACM Press, 1997.

[13] P. L. Gall and A. Arnould. Formal specifications and test: Correctness and oracle. In *COMPASS/ADT*, pages 342–358, 1995.

[14] J. Gray and A. Reuter. *Transaction Processing*. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1993.

[15] M. J. Harrold, A. J. Offut, and K. Tewary. An approach to fault modelling and fault seeding using the program dependence graph. *Journal of Systems and Software*, 36(3):273–296, Mar. 1997.

[16] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st international conference on Software engineering*, pages 597–607. IEEE Computer Society Press, 1999.

[17] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 525–537, Berlin - Heidelberg - New York, May 1997. Springer.

[18] G. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, Sept. 2003.

[19] D. Kranzlmuller, S. Grabner, and J. Volkert. Event graph visualization for debugging large applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 108–117. ACM Press, 1996.

[20] J.-C. Laprie. Dependable computing: Concepts, limits, challenges. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Special Issue*, pages 42–57, Pasadena, California, 1995. IEEE Computer Society Press.

[21] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, July 1990.

[22] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[23] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proceedings of the IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, Oct.12–19 2003.

[24] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, Nov. 8–10 2000.

[25] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, Sept. 2003.

[26] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *International Symposium on Software Testing and Analysis*, pages 195–200, 1996.

[27] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 58–65, 1994.

[28] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[29] D. J. Richardson. TAOS: Testing with analysis and oracle support. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA): August 17–19, 1994, Seattle, Washington, USA*, ACM Sigsoft, pages 138–153, New York, NY 10036, USA, 1994. ACM Press.

[30] D. J. Richardson, S. Leif-Aha, and T. O. OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[31] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *Proceedings of the 25th international conference on Software engineering*, pages 694–704. IEEE Computer Society, 2003.

[32] S. Schreyjak. On the aspect of fault tolerance in component-oriented systems. 1998.

[33] A. U. Shankar. Verified data transfer protocols with variable flow control. *ACM Trans. Comput. Syst.*, 7(3):281–316, 1989.

[34] C. Sliwa. Event-driven architecture poised for wide adoption. *COMPUTERWORLD*, May 2003.

[35] J. Sosnowski. Transient fault tolerance in digital systems. *IEEE Micro*, 14(1):24–35, Feb. 1994.

[36] J. Su and P. R. Ritter. Experience in testing the Motif interface. *IEEE Software*, 8(2):26–33, Mar. 1991.

[37] P. Vogel. An integrated general purpose automated test environment. In T. Ostrand and E. Weyuker, editors, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 61–69, New York, NY, USA, June 1993. ACM Press.

[38] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243. ACM Press, 2001.