Designing and Comparing Automated Test Oracles for GUI-based Software Applications*

Qing Xie and Atif M Memon
Department of Computer Science,
University of Maryland, College Park, MD 20742, USA

Abstract

Test designers widely believe that the overall effectiveness and cost of software testing depends largely on the type and number of test cases executed on the software. This paper shows that the *test oracle*, a mechanism that determines whether a software executed correctly for a test case, also significantly impacts the fault-detection effectiveness and cost of a test case. Graphical user interfaces (GUIs), which have become ubiquitous for interacting with today's software, have created new challenges for test oracle development. Test designers manually "assert" the expected values of specific properties of certain GUI widgets in each test case; during test execution, these assertions are used as test oracles to determine whether the GUI executed correctly. Since a test case for a GUI is a sequence of events, a test designer must decide (1) what to assert, and (2) how frequently to check an assertion, *e.g.*, after each event in the test case or after the entire test case has completed execution. Variations of these two factors significantly impact the fault-detection ability and cost of a GUI test case. A technique to declaratively specify different types of automated GUI test oracles is described. Six instances of test oracles are developed and compared in an experiment on four software systems. The results show that test oracles do affect the fault-detection ability of test cases in different and interesting ways: (1) test cases significantly lose their fault-detection ability when using "weak" test oracles, (2) in many cases, invoking a "thorough" oracle at the end of test case execution yields the best cost-benefit ratio, (3) certain test cases detect faults only if the oracle is invoked during a small "window of opportunity" during test execution, and (4) using thorough and frequently-executing test oracles can make up for not having long test cases.

Keywords: Test oracles, oracle procedure, oracle information, GUI testing, empirical studies.

1 Introduction

Testing is widely recognized as a key quality assurance (QA) activity in the software development process. During testing, a set of test cases is generated and executed on an *application*

*An earlier report of this work appeared in the Proceedings of the IEEE International Conference on Automated Software Engineering 2003 [31].

under test (AUT). Test adequacy criteria are used to evaluate the adequacy of test cases and generate more test cases if needed [57]. During testing, *test oracles* are used to determine whether the AUT executed as expected [2, 54]. The test oracle may either be automated or manual; in both cases, the actual output is compared to a presumably correct expected output.

Although research in testing has received considerable attention in the last decade, testing of *graphical user interfaces* (GUIs), which have become nearly ubiquitous as a means of interaction with today's software systems, has remained a neglected research area. GUIs today constitute as much as 45-60% of the total software code [38]. The result of this neglect is that current GUI testing techniques used in practice are *ad hoc* [33] and largely manual ([39] and Chapter 27 of [17]).

In practice, four approaches are used to create test oracles for GUI software, all of which are resource intensive. First, and the most popular, is to use a "manual oracle" [25], *i.e.*, a tester manually interacts with the GUI, performing *events* (actions performed by the user) such as click-on-CUT,[1] click-on-PASTE, and type-in-text-field, and visually checks the GUI for errors. Second is to use capture/replay tools [19] such as WinRunner[2] [30]. A tester uses these tools in two phases: a capture and then a replay phase. During the capture phase, a tester manually interacts with the GUI being tested, performing events. The tool records the interactions; the tester also visually "asserts" that a part of the GUI's response/state be stored with the test case as "expected output." The recorded test cases are replayed automatically on (a modified version of) the GUI using the replay part of the tool. The "assertions" are used to check whether the GUI executed correctly. Third is to use explicit "assert" statements in programmed GUI test cases; a tester programs the test cases (and expected output) by using tools [13,55] such as extensions of *JUnit* including *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*.[3] As these test cases execute, the assertions are used to check the correctness of the GUI. Finally, testers hard-code the expected output into test harnesses that call methods of the underlying business logic as if initiated by the GUI. This approach requires changes to the software architecture (*e.g.*, keep the GUI software "light" and code all "important" decisions in the business logic [26]); moreover, it does not perform testing of the end-user software. Test designers may also use a combination of the above approaches. For example, test designers may use capture/replay tools to capture a session (which is stored as a script) with the GUI and later manually edit the script, inserting method calls, assertions, etc. to enhance the test cases. All the above techniques require a significant amount of manual effort. If assertions are not inserted/checked at proper locations (*i.e.*, relative to events) in the test case, GUI errors may remain undetected during testing.

Since the existing techniques for creating test oracles for GUIs are resource intensive, and software quality assurance has a limited budget (both in terms of time and money), GUI testers are faced with a number of decisions on how to best expend their precious resources, while maximizing their chances of finding defects. The nature of GUI test cases (*i.e.*, sequences of events) raises new questions for testers. Answers to these questions may have an impact on the effectiveness of the overall testing process. Some of the questions relevant to

[1] In the future, for brevity, we will call these events only by their corresponding widget names, *e.g.*, CUT.
[2] `http://mercuryinteractive.com`
[3] `http://junit.org/news/extension/gui/index.htm`

test oracles include: (1) how much information needs to be specified in the expected output, *i.e.*, how many properties/values of widgets need to be checked during testing and (2) since a test case is a sequence of events, is it cost-effective to check intermediate outputs of the GUI after each event, or is it sufficient to check the final GUI output after the last event?

This paper presents a general approach to automate GUI test oracles. Six instances of GUI test oracles are created to answer the above questions empirically. Note that the scope of this paper is restricted to oracles that test the GUI only, not the entire application. For example, consider a GUI with a text-field and a *submit* button, where pressing the submit button retrieves records (related to the contents of the text-field) from a database. The test oracles developed in this paper have no way of checking the correctness of the database operation, except for a few hand-coded instances. The class of GUIs of interest to this research is formally defined in Section 3.1. The automated test oracle creation algorithms mimic test designers by automatically inserting the equivalent of "assert" statements in test cases, which are used to check whether the GUI executed correctly during testing. These test oracle creation algorithms have been incorporated into a GUI testing infrastructure, thereby enabling large empirical studies involving test oracles. An earlier report of this work showed that the type of GUI test oracle has a significant impact on test effectiveness and cost [31]. This paper extends the use of the general GUI test oracle automation approach to create six instances of automated GUI test oracles, which are used in a detailed experiment to determine whether the additional cost of developing/recording more details in expected output, and higher frequency of using the approach yields better fault-detection effectiveness. The results show that test oracles do affect the fault-detection ability of test cases in different and interesting ways: (1) test cases significantly lose their fault-detection ability when using "weak" test oracles, (2) in many cases, invoking a "thorough" oracle at the end of test case execution yields the best cost-benefit ratio, (3) certain test cases detect faults only if the oracle is invoked during a small "window of opportunity" during test execution, and (4) using thorough and frequently-executing test oracles can make up for not having long test cases.

This work advances the state-of-the-art in GUI testing by making the following specific contributions:

1. A general method to "declaratively" specify different types of automated GUI oracles.
2. A first experiment comparing GUI test oracles. GUI testers can make better decisions about the types of test oracles that they create.
3. Guidelines for test designers on how to develop test oracles, their relative strengths and weaknesses.
4. A relationship between test case length and test oracles. Testers can decide what type of oracle to use if they have long or short test cases.
5. A new "number of faults detected per comparison" measure that provides a starting point for GUI oracle comparison.
6. Discussion of domain-specific characteristics of GUIs that influence test oracle effectiveness and cost.

Structure of the paper: In the next section, we discuss related work. In Section 3, we first define GUI states and test cases; we then use these definitions to declaratively specify two parts of a GUI test oracle, namely oracle information (Section 3.3) and oracle procedure

(Section 3.4). We also show that our general approach to specify oracle information and procedure may be used to develop six different instances of test oracles. In Section 4, we present details of our experiment comparing the six oracles. Finally, we conclude with a discussion of future research opportunities in Section 5.

2 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 build upon their research. In particular, several researchers have discussed the difficulty of creating test oracles for programs that have a large volume of output [10,12,54]; this is the case with GUI software, where each GUI screen is a part of the program's output. This section describes related work in the following broad categories: *multiple test oracles*, *methods to specify oracles*, *tool support*, *reference testing*, and *GUI oracles*.

Multiple oracles: Some researchers have recognized the need to have multiple types of test oracles. Notable is the work by Richardson in TAOS (Testing with Analysis and Oracle Support) [43] who proposes several levels of test oracle support. One level of test oracle support is given by the `Range-checker` which checks for ranges of values of variables during test-case execution. A higher level of support is given by the `GIL` and `RTIL` languages in which the test designer specifies temporal properties of the software. Siepmann *et al.* in their TOBAC system [49] assume that the expected output is specified by the test designer and provide seven ways of automatically comparing the expected output to the software's actual output. Empirical evaluation of these types of oracles is needed.

Methods to specify oracles: As noted by several other researchers, software systems rarely have an automated oracle [9, 40, 43, 44]. In most cases, the expected behavior of the software is assumed to be provided by the test designer. This behavior may be specified in several ways: (1) as predicates represented in a tabular form to express functional specifications of the software [40], (2) as temporal constraints that specify conditions that must not be violated during software execution [8, 9, 43, 44], (3) as logical expressions to be satisfied by the software [11], and (4) as formal specifications [1, 4, 6, 10, 14, 15, 41], *e.g.*, algebraic specifications to represent abstract data types (ADTs) [4, 10, 15]. This expected behavior is then used by a verifier that either performs a table lookup [40], creates a state-machine model [9,20], or evaluates a boolean expression [11] to determine the correctness of the actual output.

Tools for test oracles: Information specified manually by test designers can be used by several tools to create test oracles. For example, runtime assertion checkers [7,24,46,47] and runtime monitoring tools such as MaC [23], JPaX [18] and Eagle [3] have been used as oracles. Burdy *et al.* describe the *test oracle process* [6] in which the Java Modeling Language (JML) is combined with the unit testing tool JUnit [21] for Java. Several tools (*e.g.*, the `jmlunit` tool, developed at Iowa State University) may be used to generate JUnit test classes that rely on the JML runtime assertion checker. The test classes send messages to objects of the Java classes under test. The generated test classes serve as a test oracle whose behavior is derived from the specified behavior of the class being tested. This approach requires specifications to be fairly complete descriptions of the desired behavior, as the quality of the generated test

oracles depends on the quality of the specifications. Similarly, the DAS-BOOT approach [52] uses class specifications, represented as UML state-chart diagrams, to automatically produce test drivers (with embedded test oracles) to satisfy the testers' criterion and execute the test drivers using a test script. Discrepancies between the software's behavior and the state-chart specification are reported as failures by the test oracle.

Blackburn *et al.* describe the T-VEC system [5] for model-based verification and interface-driven analysis. They combine textual requirement modeling to support automated test generation. The T-VEC system generates test vectors and test drivers, with requirement-to-test traceability information, allowing failures to be traced back to the requirement. The test driver creates a test oracle database in which the test data is derived from the model. Likewise, Robinson proposes the *semantic test process* [45] that generates tests and test oracles using models of the application.

Reference testing: A popular alternative to manually specifying the expected output is to perform reference testing [51, 53]. Actual outputs are recorded when the software is executed for the first time. 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 CAPBAK [50] capture bitmap images of GUI objects. These bitmaps are then used as test oracles to compare against actual output during regression testing. The problem with such tools is that even a small change in the GUI's layout will make the bitmap/test oracle obsolete [34]. Instead of using bitmaps, modern capture/replay tools, discussed earlier, provide sophisticated mechanisms to assert specific widgets and some values of their properties.

GUI oracles: Finally, an earlier report of this research described automated GUI test oracles for the *Planning Assisted Tester for grapHical user interface Systems* (PATHS) system [29, 32]. PATHS uses AI planning techniques to automate testing for GUIs. The oracle described in PATHS uses a formal model (in the form of pre/postconditions of each event) of a GUI to automatically derive the oracle information for a given test case.

This research builds upon the above approaches by developing techniques to declaratively specify and empirically compare multiple types of automated test oracles for GUI-based applications. The general architecture of these oracles is described next.

3 GUI Test Oracles

This section presents a high-level architecture of our automated GUI test oracle. But first, an intuitive overview of its parts is demonstrated via an example. Borrowing terminology from Richardson *et al.* [44], a test oracle is defined 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. Different types of oracles may be obtained by changing the oracle information and using different oracle procedures. For example, for testing a spreadsheet shown in Figure 1, the following two types of oracle information may be used: (1) the expected values of all the cells, and (2) the expected value of a single cell. The choice of oracle information depends on the goals of the specific testing process used. Similarly, as Figure 1 shows, the oracle procedure for a spreadsheet may (a) check for equality between expected and actual cell value, or (b) determine whether a cell value falls within a specified

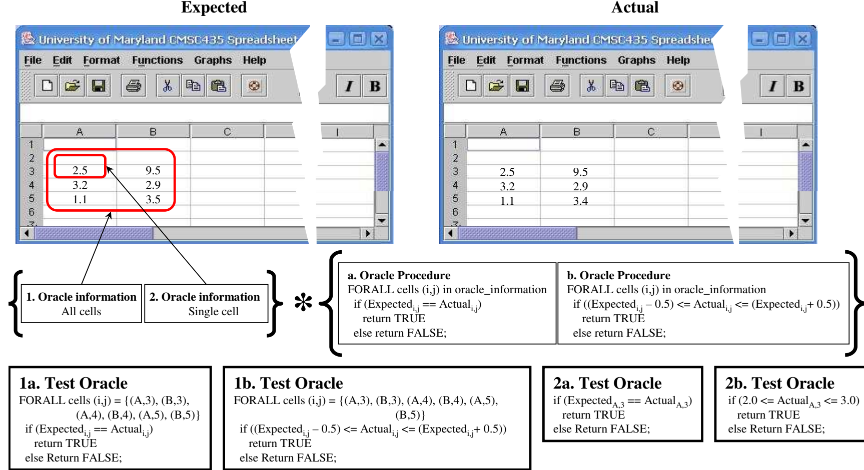


Figure 1: Four different types of oracles for a spreadsheet program.

expected range. Combining the two oracle information types and two procedure types yields four oracles: (1a) check for equality between all expected and actual cells, (1b) check whether all cell values fall within a specified expected range, (2a) check for equality between a single expected and actual cell values, and (2b) check whether a specific cell's value falls within a specified expected range. Note that the cost of maintaining and computing different types of oracle information will differ as will the cost of implementing and executing different oracle procedures.

In subsequent sections, three types of GUI oracle information in increasing level of detail and cost are declaratively specified: *widget*, *active window*, and *all windows*. The oracles are specified using six levels of complexity and cost: "check for `equality` of *widget*, *active window*, *all windows* after each event" and "check for `equality` of *widget*, *active window*, *all windows* after the last event" of the test case. Details and examples are provided in Sections 3.3 and 3.4.

We now describe the architecture of GUI test oracles. The automated test oracle creation algorithms mimic a human test designer by automatically computing "assert" statements that are used to verify the correctness of the GUI during test case execution. These assert statements contain two parts: (1) the expected values of certain properties of specific GUI widgets, and (2) the comparison process (which we have hard-coded to "equality") and its frequency of execution. Hereafter, the former is referred to as "oracle information" and the latter as "oracle procedure."

As shown in Figure 2, 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 [32] or by using a "golden" (assumed correct baseline) version of the software [51, 53] (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 in [32], 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 determines if the GUI is executing as expected. As described later in Section 3.1, the state of a GUI is represented as a set of all triples (w_i, p_j, v_k), where w_i is a GUI widget, p_j is its property, and v_k is its

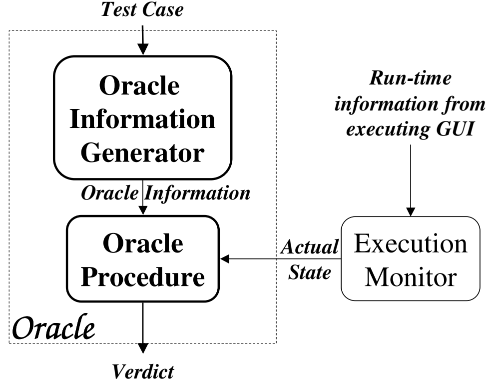


Figure 2: An overview of the GUI oracle.

value.

The above partitioning of functionality allows the definition of a simple algorithm for test execution. Given a test case, the algorithm executes all its events, computes the expected state, obtains the GUI's actual state, compares the two states, and determines if the actual is as expected. This algorithm is shown in Figure 3. The algorithm `ExecTestCase` takes four parameters: (1) a test case **T** (LINE 1) of the form $< S_0, e_1; e_2; \ldots; e_n >$, where S_0 is the state of the GUI before the test case is executed and $e_1; e_2; \ldots; e_n$ is an event sequence; (2) a set of integers **OPF** (LINE 2) that determines how frequently the oracle procedure is invoked (details in Section 3.4); (3) $\mathbf{C}_{OI}$ (LINE 3), a boolean constraint used to obtain relevant triples for the oracle information (examples of some constraints are shown in Section 3.3); and (4) $\mathbf{C}_{AS}$ (LINE 4), a similar boolean constraint but used by the oracle procedure to obtain relevant triples for the actual state.

The algorithm traverses the test case's events one by one (LINE 5) and executes them on the GUI (LINE 6). The oracle information $\mathbf{OI}_i$ is obtained for the event e_i (LINE 7); i, the index of the current event, ties the event e_i to the corresponding oracle information $\mathbf{OI}_i$. The constraint $\mathbf{C}_{OI}$ is used to select a subset of the complete expected state. This constraint is discussed in Section 3.3. Similarly, the actual state $\mathbf{AS}_i$ of the GUI, also constrained by $\mathbf{C}_{AS}$, is obtained (LINE 8). The oracle procedure is then invoked (LINE 9) that determines whether the software's execution was correct for the event.

Having outlined how a GUI test oracle is used during test case execution, we now describe details of GUI state, and oracle information and procedure.

3.1 GUI Model

Some basic terms that are needed to understand the design of the test oracles are defined first. The techniques and tools used for this research require that a GUI be modeled in terms of the *widgets* (basic building blocks) the GUI contains, their *properties*, and the *values* of the properties. We also require GUI *events* to be deterministic so as to be able to predict their outcome. Hence, to provide focus, the discussion in this paper is limited to a particular subclass of GUIs, defined next.

```
ALGORITHM :: ExecTestCase(
  T: Test case; /* T = < S_0, e_1; e_2; ...; e_n > */                       1
  OPF ⊆ {1, 2, 3, ..., n}; /* oracle procedure freq. */                     2
  C_OI: Boolean Constraint; /* on oracle information */                     3
  C_AS: Boolean Constraint; /* on actual state */){                         4

  FOREACH e_i in T DO { /* for all events */                                5
   Execute(e_i); /* perform the event on the GUI */                         6

   /* obtain the expected state for event e_i */
   OI_i ← GetOracleInfo(i, C_OI);                                           7

   /* extract the GUI's actual state */
   AS_i ← GetActualState(i, C_AS);                                          8

   /* invoke the oracle procedure */
   IF !(Op(AS_i, OI_i, C_AS, OPF, i)) THEN {                                9
     Return(FAIL)}} /* test case fails */                                   10
  Return(PASS)} /* if no failure, report success */                         11
```

Figure 3: Test execution algorithm.

Definition: A *Graphical User Interface* is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events, and produces deterministic graphical output. A GUI contains graphical *widgets*; each widget has a fixed set of *properties*. At any time during the execution of the GUI, these properties have discrete *values*, the set of which constitutes the state of the GUI. □

Note that this definition would need to be extended for other GUI classes such as web-user interfaces that have synchronization/timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible (or not practical/feasible) to model the state of the software in its entirety and hence the effect of an event cannot be predicted. Since the software's back-end is not modeled in this research, GUIs that are tightly coupled with the back-end code, *e.g.*, ones whose content is created dynamically using a database, are also excluded.

A GUI is modeled 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* $V = \{v_1, v_2 ..., v_n\}$ (*e.g.*, red, bold, 16pt) associated with the properties. Note that the discrete values may depend on certain system parameters (*e.g.*, the font-size may be a function of the font-size being used by the windowing system). However, at any point during the GUI's execution, these values are instantiated. We also assume that the system parameters that may effect certain values

remain unchanged. Hence, the GUI can be fully described in terms of the specific widgets that it currently contains and the values of their properties.

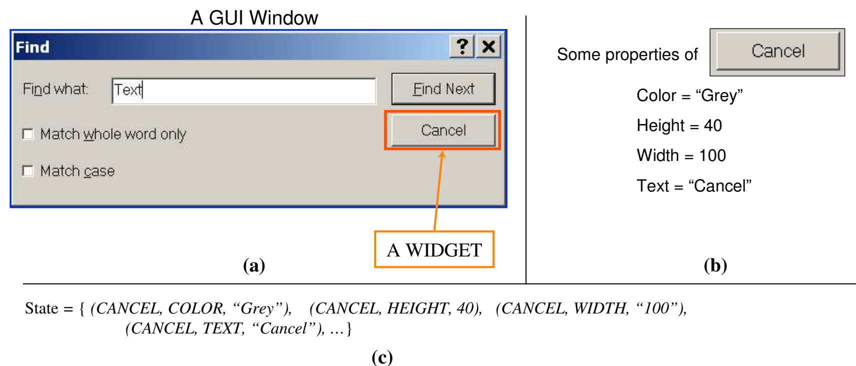


Figure 4: (a) `Find` GUI, (b) some properties of the `CANCEL` button, and (c) the partial state of the `Find` window.

For example, consider the `Find` GUI window shown in Figure 4(a). This GUI contains several widgets; one is explicitly labeled, namely `CANCEL`; a small subset of its properties is shown in Figure 4(b). Note that all widget types have a designated set of properties and all properties can take values from a designated set.

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

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

A description of the *complete state* would contain information about the types of *all* the widgets currently extant in the GUI, as well as *all* of the properties and their values for each of those widgets. The state of the `Find` GUI, partially shown in Figure 4(c), contains all the properties of all the widgets in `Find`.

In this research, the definition of the state of a GUI is extensively used to develop the oracle information and procedure. As will be seen later, oracle information is associated with each test case. Hence, a GUI test case is formally defined next.

3.2 GUI Test Cases

With each GUI is associated a distinguished set of states called its *valid initial state set*:

Definition: A set of states S_I is called the *valid initial state set* for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked. □

The state of a GUI is not static; *events* performed on the GUI widgets change its state. States that result from applying sequences of events to valid initial states are called the *reachable states* of the GUI. The events are modeled as functions from one state to another.

Definition: The **events** $E = \{e_1, e_2, \ldots, e_n\}$ associated with a GUI widget are functions from one state to another state of the GUI. □

Events may be strung together into sequences. Of importance to testers are sequences that are permitted by the structure of the GUI [35]. These event sequences are called *legal* and are defined as follows:

Definition: A *legal event sequence* of a GUI is $e_1; e_2; e_3; ...; e_n$ where e_{i+1} can be performed *immediately* after e_i. □

The definitions of events, widgets, properties, and values can be used to formally define a GUI test case:

Definition: A **GUI test case** T is a pair $< S_0,\ e_1;\ e_2;\ \ldots;\ e_n >$, consisting of a state $S_0 \in S_I$, called the *initial state for T*, and a legal event sequence $e_1; e_2; \ldots; e_n$. □

Having defined the basic GUI concepts (the interested reader is referred to [28,29,33,36] for details and examples), we now describe test oracle information.

3.3 Test Oracle Information

The oracle information is a description of the GUI's expected state for a test case.

Definition: For a given test case $T = < S_0,\ e_1;\ e_2;\ \ldots;\ e_n >$, the **test oracle information** is a sequence $< S_1, S_2, \ldots, S_n >$, such that S_i is the (possibly partial) expected state of the GUI immediately after event e_i has been executed on it. □

Recall from Section 3.1 that the GUI's state is a set of triples of the form (w_i, p_j, v_k), where w_i is a widget, p_j is a property of w_i, and v_k is a value for p_j. Hence the oracle information for a test case is a sequence of these sets. Note that oracle information has been deliberately defined in very general terms, thus allowing the creation of different instances of oracles. The least descriptive oracle information set may contain a single triple, describing one value of a property of a single widget. The most descriptive oracle information would contain values of all properties of all the widgets, *i.e.*, the GUI's complete expected state. In fact, all the non-null subsets of the complete state may be viewed as a spectrum of all possible oracle information types, with the single triple set being the smallest and the complete state being the largest. A declarative mechanism (in the form of a boolean constraint called $\mathbf{C}_{OI}$ on Line 6 in Figure 3) is used to specify a point in this spectrum. Three instances of oracle information are presented next; for every triple that is included in the state description, the constraint $\mathbf{C}_{OI}$ must evaluate to `TRUE`.

1. **widget (LOI1):** the set of all triples for the single widget w associated with the event e_i being executed. The constraint is written as $(\#1 == w)$, where $\#1$ represents the first element of the triple. If applied to a triple with "w" as its first element, the constraint would evaluate to `TRUE`; in all other cases, it would evaluate to `FALSE`. Figure 5 shows an example of the oracle information. The test case contains the `Cancel` event in the `Find` window. The complete expected state S_i of the GUI after `Cancel` has been executed is also shown. For the *widget* level test oracle information, only the (boxed) triples relevant to `Cancel` are stored.

2. **active window (LOI2):** the set of all triples for all widgets that are a part of the currently active window W. The constraint is written as $(inWindow(\#1, W))$, where $inWindow(a, b)$ is a predicate that is `TRUE` if widget a is a part of window b.

3. **all windows (LOI3):** the set of all triples for all widgets of all windows. Note that the constraint for this set is simply `TRUE` since it is the complete state of the GUI.

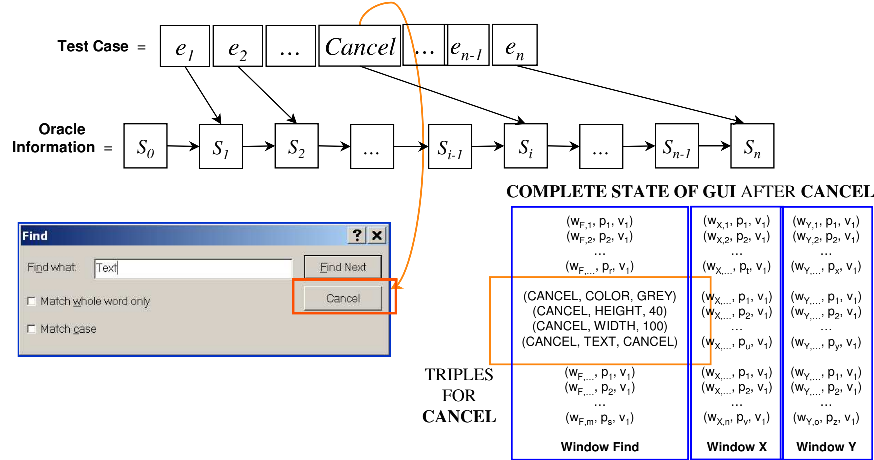


Figure 5: Oracle information for the `Cancel` event.

For brevity, the terms LOI1 to LOI3 will be used for the above three *levels of oracle information*. Note that although only three instances of test oracle information have been specified, the specification mechanism is general and may be used to specify many other instances. In Figure 3, the subroutine GetOracleInfo(i, $\mathbf{C}_{OI}$) is used to compute the oracle information. There are several different ways to realize GetOracleInfo, three of which are outlined next:

1. As discussed in Section 2, using **capture/replay tools** is a popular method to obtain the oracle information for GUIs [22]. Recall that capture/replay tools are semi-automated tools used to record and store a tester's manual interaction with the GUI; the goal is to replay the interactions and observe the GUI's output. Testers manually select some widgets and some of their properties that they are interested in storing during a capture session. This partial state is used as oracle information during replay. Any mismatches are reported as possible defects.

2. In the experiment presented in this paper, we have automated the above approach by developing a technique that we call **execution extraction**, a form of reference testing discussed in Section 2. The key idea of using execution extraction is to designate the current version of an application as "correct" and use it as a specification of the software. During the execution extraction process, oracle information is collected via reverse engineering [27] from the "golden" version of the application. Platform-specific technologies such as Java API,[4] Windows API,[5] and MSAA[6] are used to obtain this information. The oracle information is then used to test future versions of the software or ones that have been artificially seeded with faults.

3. We have used **formal specifications** in earlier work [32] to automatically derive oracle information. These specifications are in the form of pre/postconditions for each GUI

[4]java.sun.com
[5]msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp
[6]msdn.microsoft.com/library/default.asp?url=/library/en-us/msaa/msaaccrf_87ja.asp

event.

3.4 Oracle Procedure

The oracle procedure is the process used to compare the oracle information with the executing GUI's actual state. It returns `TRUE` if the actual and expected states match, `FALSE` otherwise. Formally, an oracle procedure is defined as:

Definition: A **test oracle procedure** is a function $\zeta(\mathbf{OI}, \mathbf{AS}, \mathbf{C}_{OI}, \mathbf{C}_{AS}, \Phi) \longrightarrow \{$`TRUE`, `FALSE`$\}$, where **OI** is the oracle information, **AS** is the actual state of the executing GUI, $\mathbf{C}_{OI}$ is a boolean constraint on **OI**, $\mathbf{C}_{AS}$ is a boolean constraint on **AS**, and Φ is a comparison operator. ζ returns `TRUE` if **OI** and **AS** "match" as defined by Φ; `FALSE` otherwise. □

ALGORITHM :: `OP`(
  $\mathbf{AS}_i$: Actual state; /* *for event* e_i */ 1
  $\mathbf{OI}_i$: Oracle information; /* *for event* e_i */ 2
  $\mathbf{C}_{AS}$: Boolean Constraint; /* *on actual state* */ 3
  $\mathbf{OPF} \subseteq \{1, 2, 3, \ldots, n\}$ /* *oracle procedure freq.* */ 4
  i: event number; /* *current event index* $1 \leq i \leq n$ */) { 5

  `IF` ($i \in \mathbf{OPF}$) `THEN` /* *compare?* */ 6
   RETURN(FILTER($\mathbf{OI}_i$, $\mathbf{C}_{AS}$) == $\mathbf{AS}_i$) 7
  `ELSE` RETURN(TRUE)} 8

Figure 6: Oracle procedure algorithm.

The oracle procedure may be invoked as frequently as once after every event of the test case or less frequently, *e.g.*, after the last event. The algorithm for the oracle procedure is shown in Figure 6. *Note that our specific implementation* `OP` *of* ζ *takes extra parameters* i *and* **OPF** *that account for this frequency;* i *is the event number in the test case and* **OPF** *is a set of numbers that specify when the comparison is done. Also note that* Φ *is hard-coded to "set equality", hence omitted from* `OP`*'s parameters (Line 7 of Figure 6).* $\mathbf{C}_{OI}$ is also omitted since **OI** has already been filtered before `OP` is invoked from LINE 9 of Figure 3. `OP` takes five parameters described earlier. The comparison process is straightforward – if the GUI needs to be checked at the current index i of the test case (LINE 6), then the oracle information is filtered[7] using the constraint $\mathbf{C}_{AS}$ to allow for *set equality* comparison. The constraint $\mathbf{C}_{AS}$ (not $\mathbf{C}_{OI}$) ensures that the result of the filtering is compatible with $\mathbf{AS}_i$. The oracle procedure returns `TRUE` if the actual state and oracle information sets are equal.

Note that it is important to provide the constraint $\mathbf{C}_{AS}$ and the set **OPF** to completely specify the oracle procedure. The definition of `OP` is now used to specify six different instances of test oracles.

- **L1:** *After each event of the test case*, compare the set of all triples for the single widget w associated with that event. The constraint $\mathbf{C}_{AS}$ is written as $(\#1 == w)$ and **OPF**

[7] Note that this filtering is unnecessary if `OP` is invoked by our test case executor, since it already filters the oracle information. We include the filtering step here for completeness.

$= \{1, 2, 3, \ldots, n\}$. Note that $\mathbf{C}_{AS}$ is first used to select relevant triples for the actual state (LINE 8 of Figure 3) and then later to filter the oracle information (LINE 7 of Figure 6). We show **L1** in Figure 7; it compares the state triples relevant to the widget $\mathbf{W_x}$.

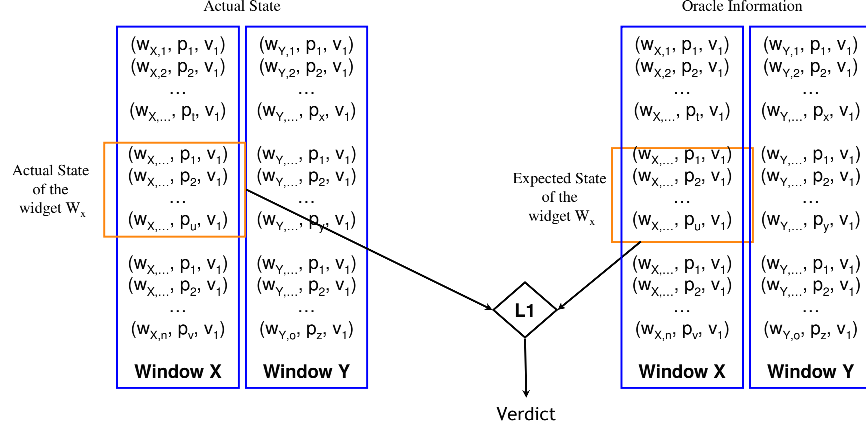


Figure 7: **L1** compares widget-relevant triples after each event in the test case.

- **L2:** *After each event of the test case*, compare the set of all triples for all widgets that are a part of the currently active window W. The constraint $\mathbf{C}_{AS}$ is written as $(inWindow(\#1, W))$ and **OPF** $= \{1, 2, 3, \ldots, n\}$.
- **L3:** *After each event of the test case*, compare the set of all triples for all widgets of all windows. The constraint $\mathbf{C}_{AS}$ is written as `TRUE` and **OPF** $= \{1, 2, 3, \ldots, n\}$.
- **L4, L5, L6:** *After the last event of the test case*, compare the set associated with the current widget, active window, and all windows, respectively. **OPF** $= \{n\}$ for all these oracles.

Even though only six instances of oracles have been developed, the definition of `OP` is very general and may be used to develop a variety of test oracles.

4 Experiment

Having presented the design of GUI test oracles and our ability to specify multiple oracles, we now present details of an experiment using actual software and test cases to compare the different oracles.

4.1 Research Questions

We introduce two notations: $C(T, L)$ – the cost of executing a test case T with oracle L; $F(T, L)$ – the number of faults that test case T detects when using test oracle L.

The way we have defined test oracles L1 to L6 leads to some immediate observations. First, we note that L1 to L3 have been defined with increasing complexity (as have L4 to

L6), which will have a direct impact on their relative cost (*i.e.*, time to generate/execute); L3 will be most expensive and L1 the least expensive (hence, for all test cases T, $C(T, L3) \geq C(T, L2) \geq C(T, L1)$). Similarly, $C(T, L6) \geq C(T, L5) \geq C(T, L4)$. We can also safely say that $C(T, L4) \leq C(T, L1)$, $C(T, L5) \leq C(T, L2)$, and $C(T, L6) \leq C(T, L3)$. Second, T with oracle L3 is expected to reveal more faults than T with oracle L1 or L2, simply because L3 "looks at" a larger set of GUI widgets during T's execution (*i.e.*, $F(T, L3) \geq F(T, L2) \geq F(T, L1)$); it can certainly do no worse. Similarly, T with L6 is expected to reveal more faults than with either L4 or L5 (*i.e.*, $F(T, L6) \geq F(T, L5) \geq F(T, L4)$). It is, however, not clear how L1 compares to L4 in terms of T's fault-detection effectiveness, *i.e.*, is $F(T, L4) < F(T, L1)$ or is $F(T, L4) = F(T, L1)$? (similar questions can be asked about the pairs (L2, L5) and (L3, L6)). Also, even though the above relationships have been presented as "obvious," the magnitude of these relationships needs further study to determine practical significance. For example, even though, in theory, the relationship $C(T, L5) \leq C(T, L2)$ holds, *how much more* does L2 cost? Is the additional cost worth the extra faults that may be found (if any) when using L2? Answers to these questions will demonstrate the practical significance of using different test oracles.

In particular, the following questions need to be answered to show the relative strengths of the test oracles and to explore the cost of using different types of oracles.

- **Q1:** What effect does the oracle information have on the fault-detection effectiveness of a test case? Is the additional effectiveness worth the cost?
- **Q2:** What effect does the invocation frequency of a test oracle have on the fault-detection effectiveness of a test case? Is the additional effectiveness worth the cost?
- **Q3:** What combination of oracle information and procedure provide the best cost-benefit ratio?

While answering the above questions, we will also informally study situations in which generating/using a complex (more expensive) oracle are justified. For example, if a tester has only short test cases (and/or a small number of test cases), will the test results improve if complex oracles are used? We will refer to this question as **Q4**.

4.2 Modeling Cost and Fault Detection Effectiveness

One factor of cost is the time needed to execute a test case with a given oracle; this time is directly proportional to the number of comparisons of $(widget, property, value)$ triples during test case execution. Another factor of cost is the effort needed to create the test oracle, which as explained earlier is a manual process. This factor is also directly related to the number of triples specified by a tester. For example, a tester using a capture/replay tool needs to specify each triple that must be stored and compared. Hence, we use the number of *widget comparisons* done (during execution of test case T) by test oracle L as a measure of cost. We use the notation $\mathcal{C}(T, L)$ for this measure. For example, $\mathcal{C}(T, L4) = 1$ for all test cases, since L4 involves comparing the triples for a single widget.

Since we are interested in studying the impact of using different test oracles on each test case, we model fault-detection effectiveness on a per test case basis. We define $\mathcal{F}(T, L)$ of a test case T as the number of faults it detects with test oracle L. Obviously, a higher value

of $\mathcal{F}$ is desirable but at a reasonable cost. A more appropriate measure called the "number of faults detected per comparison" (ξ) is computed as:

$$\xi(T,O) = \begin{cases} \frac{\mathcal{F}(T,L)}{\mathcal{C}(T,L)} & \text{if } \mathcal{C}(T,L) > 0, \\ undefined & \text{if } \mathcal{C}(T,L) = 0. \end{cases}$$

The second case of the definition is included only for completeness; as long as T is a non-empty sequence, $\mathcal{C}(T,L)$ will be positive. The ξ value gives us a good measure of the relative cost and benefit of test oracles. A test oracle that performs very few comparisons yet reveals a large number of faults will have a high ξ value, which is desirable due to the larger number of faults that it detects. However, ξ has several weaknesses. First, a test oracle that performs very few (say x) (*e.g.*, $x = 1$ for L4) comparisons and reveals too few faults (say y) will have a higher ξ value than one that performs more comparisons (*e.g.*, $10x$) but detects more faults (*e.g.*, $5y$). However, the latter oracle may be more desirable. In practice, the cost of missing a fault may be extremely high. Indeed, in particular domains, a tester may be willing to spend considerable resources to detect even a single fault. In such domains a test oracle with a high average $\mathcal{F}$ value is clearly desirable. Second, all faults are given equal weight in this model. The ξ formula can be easily modified if the "severity" of faults is to be considered; in this experiment we consider all faults to be of equal severity. Although ξ suffers from some of these problems, it provides us with an adequate starting point for oracle comparison. Recognizing the weaknesses of the cost/benefit model, we present details of the actual number of faults detected; readers can interpret the results for their particular domains/situations. We also refer the interested reader to related literature that provides an excellent discussion on such cost-benefit models and their advantages/disadvantages [16,48].

To answer Q1, we will compare the $\mathcal{F}$ and ξ values for oracles L1–L3 and L4–L6. To answer Q2, we will compare the $\mathcal{F}$ and ξ values for the oracle pairs (L1,L4), (L2,L5), and (L3,L6). For Q3, we will compare the average ξ values of all oracles. Finally, for Q4, we will study the impact of test case length and their number on the ξ values for each oracle.

4.3 Experimental Procedure

We select different software subject applications with GUI front-ends and, for each application, perform the following steps:

Step 1: generate test cases,

Step 2: generate different levels of oracle information (using execution extraction),

Step 3: execute the test cases on the application using different oracle procedures. Measure the following variables:

- **Number of Faults Detected:** A "fault is detected" if the expected and actual states mismatch.
- **Number of Comparisons:** This is the number of widget comparisons between the expected and actual states for each oracle.

Step 4: from the execution results, eliminate test runs that were affected by factors beyond our control, *e.g.*, those that crash the subject application irrespective of the test oracle used.

We discuss details of these steps in subsequent sections.

Subject Application	LOC	Classes	Windows	Widgets	Properties
TerpPaint	9,287	42	8	173	2076
TerpPresent	4,769	4	5	95	1140
TerpSpreadSheet	9,964	25	6	124	1488
TerpWord	1,747	9	8	86	1032
TOTAL	25,767	80	27	478	5736

Table 1: Our Subject Applications.

4.3.1 Subject Applications

We had to satisfy several requirements when choosing our subject applications. First, we wanted to have access to the source code, CVS development history, and bug reports (for oracle creation, described later). Second, we wanted applications that were "GUI-intensive," *i.e.*, ones without complex back-end code. The GUIs of such applications are typically static, *i.e.*, not generated dynamically from back-end data. Finally, we wanted non-trivial applications, consisting of several windows and widgets.

The subject applications for our experiment 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[8] and includes the applications TerpWord, TerpPresent, TerpPaint and TerpSpreadSheet. TerpWord is a word-processor with drawing capability; TerpPresent is used to prepare slides and present them online; TerpPaint is an imaging tool; TerpSpreadSheet is a compact spreadsheet program. They have been implemented using Java. Table 1 summarizes the characteristics (lines of code, number of classes, windows, widgets, and the properties of widgets) of these applications. The widget counts shown include only those widgets on which events can be performed. Most of the code written for the implementation of each application is for the GUI. None of the applications have complex underlying "business logic."

4.3.2 Step 1: Generate Test Cases

We used an automated tool (GUITAR[9]) to generate 600 test cases for each application. We chose the number 600 since we could execute them in a reasonable amount of time; we later describe that we created 100 fault-seeded versions of each application; with 600 test cases, 100 versions, and 4 applications, we would need 240K test runs; since an average test run takes 30 seconds, our experiment would run for months. The number 600 allowed us to keep the experiment within the realm of practicality.

GUITAR employs previously developed graph structures (*event-flow graphs* and *integration trees* [35]) to generate test cases. A detailed discussion of the algorithms used by GUITAR is beyond the scope of this paper (see [29] for additional details and analysis). The key idea is that once the graph structures have been created, a simple graph traversal algorithm is used to generate sequences of events, each of which is a test case.

Since we wanted to study the role of test case length in GUI testing (Q4), we used an

[8]Available at `http://www.cs.umd.edu/users/atif/TerpOffice`

[9]guitar.cs.umd.edu

algorithm that allowed us to control the length of the test case by specifying a limit on the graph traversal. Hence, we created buckets of test cases by length. One of the problems with automated GUI testing is the creation and execution of long test cases. Our experience with GUI testing tools has shown that test cases longer than 20 events typically run into problems during execution, mostly due to timing issues with windows rendering. As events in a test case are executed, the test case replayer keeps track of GUI state information for each event. For long sequences, the overhead of keeping track of this information significantly affects the performance of the JVM, which is also responsible for executing the subject application. After 20 events, window rendering becomes so slow that events are executed even before the corresponding widget is available, resulting in uncaught exceptions. Because of this limitation of our tool, we capped the GUI length at 20, *i.e.*, we had 20 buckets, one for each length. Since we did not want to favor any one bucket, we generated an equal number of test cases, *i.e.*, 30, per bucket. In all we had 600 test cases per application.

4.3.3 Step 2: Generate Oracle Information

The next step was to obtain the oracle information for each test case. Since approaches (1) and (3) discussed in Section 3.3 are extremely resource intensive, we chose to use approach (2). We used an automated tool (also a part of GUITAR) that implements execution extraction. This tool automatically executes a given test case on a software system and captures its state (widgets, properties, and values) by using the Java Swing API. Due to the limitations of this API, we were able to extract only 12 properties for each widget. By running this tool on the four subject applications for all 600 test cases, we obtained the oracle information. Note that the tool extracted all three levels of oracle information.

Reported Fault in Bug Database	Corrected Code	Fault #1	Fault #2

for(row = 0 ; row < 1024 ; ++row)
for(col = 0 ; col < 26 ; ++col)
display_cell(**col**, **row**);
	for(row = 0 ; row < 1024 ; ++row)
	for(col = 0 ; col < 26 ; ++col)
	display_cell(row, col);
		for(row = 0 ; row < **26** ; ++row)
		for(col = 0 ; col < 26 ; ++col)
		display_cell(row, col);
			for(row = 0 ; row < 1024 ; ++row)
			for(col = 0 ; **row** < 26 ; ++col)
			display_cell(row, col);

(a)	(b)	(c)	(d)

Table 2: Seeding GUI Faults.

There are several ways to obtain different versions of an application. One popular way is to use CVS history and bug reports; the latest version at the head of the CVS is treated as the "correct" version; the previous versions may contain (now known) faults that may have been fixed over time. Indeed the latest version may have been the result of numerous bug fixes. During the development of TerpOffice, a bug tracking tool 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. We studied the CVS logs and bug reports for our subject applications. We were careful to identify only *GUI faults*, *i.e.*, those that were manifested on the visible GUI at some point of time during the software's execution. Unfortunately, we were unable to find a sufficient number of faults relevant to the GUI. However, recognizing that the small number of reported faults that we did find are an excellent representative of faults that are introduced by developers during implementation, we used a

variation of this approach. We used the real faults as "templates" and created instances of similar faults and seeded them in the latest version of each application. Table 2(a) shows an example of a fault reported in our bug database and Table 2(b) shows the (later) corrected segment of the same code. Table 2(c) and 2(d) show examples of faults seeded into this code.

One common issue with fault-seeding is that some will never be manifested on the GUI. Hence, we chose a large enough number to seed so that we could still get useful results, even if some of them were not manifested. We seeded 100 faults in each application. Four graduate students seeded the faults independently. These students had taken a graduate course in software testing and were familiar with popular testing techniques.

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 experiments and also simplifies the computation of the variable "Number of Faults Detected"; now we can simply count the faulty versions that led to a mismatch between actual state and oracle information. Note, however, that this approach may overestimate fault detection; in cases where multiple faults would actually mask each other's effects, causing no GUI errors to be manifested. Other researchers have also studied issues of such masking [48]; their studies have shown that there are low incidents of masking, causing no significant impact on results. We use these results to side-step the issue of fault masking, except for faults that cause sofware crashes; we eliminate these faults as discussed in Section 4.3.5.

4.3.4 Step 3: Oracle Procedure and Test Executor

We executed all 600 test cases on all 100 versions of each subject application (hence we had 60,000 runs per application). When each application was being executed, we extracted its run-time state based on the six oracles and compared it with the stored oracle information and reported mismatches. We used "set equality" to compare the actual state with the oracle information. Note that we ignored widget positions during this process since the windowing system launches the software at a different screen location each time it is invoked.

We noted several points about our test executor. We associated Java methods that would invoke the default event associated with each GUI widget. Whenever a widget was encountered in a test case, we executed the corresponding method. The most common method was `doClick()` associated with widgets such as buttons. If the widget was a text-field, we read values from a database and automatically filled in the text-field. We initialized the database manually with commonly-used values depending on the text-field type.

Each test case required between 10 and 60 seconds to execute. The time varied by application and the number of GUI events in the test case. The total execution time was slightly less than one month for each application. 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 based on the oracle.

Our resulting data can be viewed as a (hypothetical) table (hereafter referred to as the "data table") for each application. Each row of this table represents the result of executing each test case on each fault-seeded version. Hence the table has $600 \times 100 = 60,000$ rows. It has 6 columns, one for each test oracle. Each entry of the table is a boolean

value (`Match`/`Mismatch`) indicating whether at least one mismatch occurred (the fault was detected) during test case execution when using the corresponding oracle.

4.3.5 Step 4: Cleaning up the Data Table

During test execution, two factors that were independent of test oracle caused us to filter out some of the rows in the data table. These factors include the impact of seeded faults on software execution and interactions between test cases and faults. The former is due to the way a fault is manifested during execution. The latter is due to test-case design, whether the test case caused the execution of the program statement in which the fault was seeded, and whether the seeded fault was manifested on the GUI. We list and discuss each of these issues next:

1. *Effect of fault on software execution:* several test cases (during execution) crashed specific fault-seeded versions, irrespective of the test oracle. These test cases executed properly on other versions. We eliminated such crashes from our data. There were 954, 1595, 2302, and 4829 crashes for TerpPresent, TerpWord, TerpPaint, and TerpSpreadSheet respectively. Each of these *(test case, fault-seeded version)* pairs caused the filtering of one row in the table.
2. *Fault design:* several faults were never detected by even a single test case. We call these faults "unobserved." There were 58, 5, 43, and 1 unobserved faults for TerpPresent, TerpWord, TerpPaint, and TerpSpreadSheet respectively. We discarded these faults from our data. For each such fault, we filtered out a maximum of 600 table rows, one for each test case.
3. *Test case design:* one test case in TerpPaint did not detect even a single fault for any oracle. We eliminated this test case, causing the filtering of 57 rows, one for each of the remaining fault-seeded versions of TerpPaint.
4. Finally, a large number of test cases did not detect certain faults for any test oracle. We eliminated these rows from the table.

Subject Applications	**TerpPresent**		**TerpWord**		**TerpPaint**		**TerpSpreadSheet**	
Total Rows		60000		60000		60000		60000
Filtering Steps	*#*	*Rows Filtered*	*#*	*Rows Filtered*	*#*	*Rows Filtered*	*#*	*Rows Filtered*
1 *Crashes*		954		1595		2302		4829
2 *Unobserved faults*	58	34800	5	3000	43	25800	1	600
3 *Test cases not detecting any faults*	0	0	0	0	1	57	0	0
4 *Test cases not detecting specific faults*		20566		54013		31779		52323
Remaining Rows		**3680**		**1392**		**62**		**2248**

Table 3: The Data Table Cleanup Steps.

These "filtering steps" are also shown in Table 3. Note that we executed them in the order presented. Also note that after the last filtering step, some fault-seeded versions may have been filtered out entirely, since test cases either crashed them or did not detect the faults.

The remaining data, which we use for our analysis, are the rows of the data table that contain at least one `Mismatch` entry. These rows represent test runs that yielded a successful

fault detected for at least one test oracle. That is, the test case successfully executed the program statement in which the fault was seeded and the fault manifested as a GUI error. This data is relevant to the results since it helps us to compare test oracles. We note that other entries may be useful for other analyses, *e.g.*, to study characteristics of test cases, which are beyond the scope of this work.

The number of test cases that appeared in at least one row of the resulting data table were 600 for TerpPresent, 424 for TerpWord, 18 for TerpPaint, and 358 for TerpSpreadSheet. Similarly, the number of faults that appeared in at least one row in the table were 25 for TerpPresent, 82 for TerpWord, 18 for TerpPaint, and 83 for TerpSpreadSheet. These numbers will be used in the analyses presented.

4.4 Threats to Validity

Threats to external validity [56] are conditions that limit the ability to generalize the results of our experiment to industrial practice. Our subject applications, types of faults that we seed, and the way we create the oracle information are the biggest threats to external validity. First, we have used four applications, developed by students, as our subject applications. Although they have different types of GUIs, this does not reflect the wide spectrum of possible GUIs that are available today. Second, all our subject programs were developed in Java. Although our abstraction of the GUI maintains uniformity between Java and non-Java applications, the results may vary for non-Java applications. Third, our GUIs are static, in that we do not have widgets that are created on-the-fly from back-end data. We expect that our results do not generalize to more "dynamic" GUIs. Fourth, the faults that were seeded in this experiment represent a small subset of faults that are prevalent in student-developed software. Our test oracles may behave differently for other classes of faults. Fifth, for text-fields, we initialized values manually and stored them in a database. Although we made every attempt to identify categories of text fields (*e.g.*, numbers, file-names, alphanumeric strings) and their choices, the GUIs may have executed differently for other values. Finally, we have used execution extraction to create oracle information. While this provided an efficient mechanism to create different types of oracles, in practice testers use manual techniques to specify oracles. Moreover, they use different types of assertions, *e.g.*, checking ranges, not just state comparisons as test oracles. Our work needs to be extended to handle such assertions.

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher's knowledge. The biggest threats to internal validity are related to the way we create test cases. In order to keep our running time reasonable, we created 600 test cases; due to the limitations of our tools, we capped the length at 20 events. The results may vary for longer test cases. We also used one technique to generate test cases – using event-flow graphs. Other techniques, *e.g.*, using capture/replay tools and programming the test cases manually may produce different types of test cases, which may show different execution behavior. Another threat to internal validity is related to our measurement of fault detection effectiveness; each fault was seeded and activated individually. Note that, as discussed earlier, multiple faults present simultaneously can lead to more complex scenarios that include fault masking.

Threats to construct validity arise when measurement instruments do not adequately

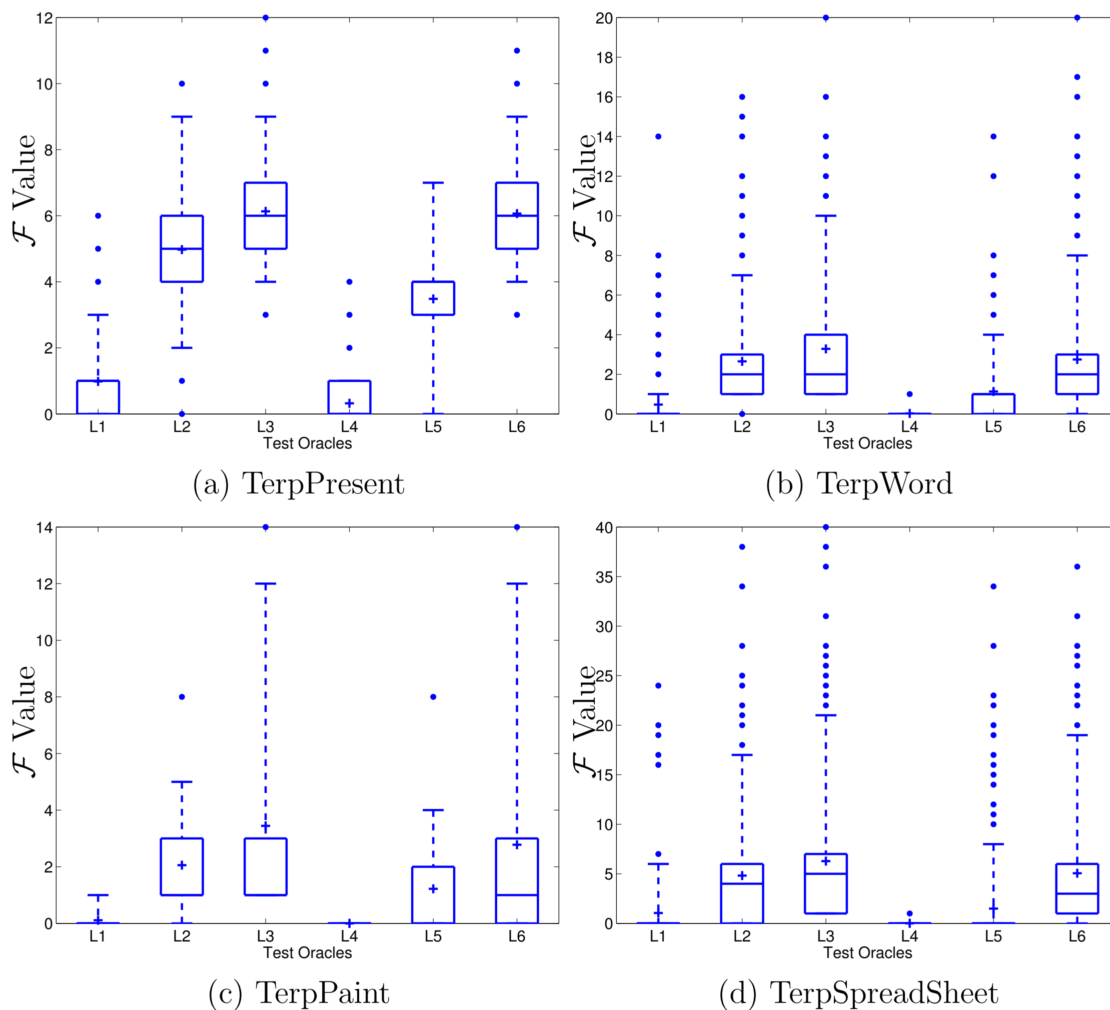
capture the concepts they are supposed to measure. For example, in this experiment our measure of cost combines human effort and execution cost. A more accurate measure would use domain-specific knowledge to assign appropriate "weights" to these cost components.

Other threats related to our cost-benefit model were discussed earlier. The results of our experiment, should be interpreted keeping in mind the above threats to validity.

4.5 Results

4.5.1 Fault-Detection Effectiveness

Recall that $\mathcal{F}(T, L)$ was defined as the number of faults detected by test case T when using oracle L. This value is computed from the data table as $\mathcal{F}(T, L) = \sum_{f \in F} DT(T, f, L)$, where the function DT returns 1 if the the entry for column L in the row corresponding to test T and fault f is `Mismatch`; 0 otherwise. F is the set of all faults in the data table.



(a) TerpPresent (b) TerpWord

(c) TerpPaint (d) TerpSpreadSheet

Figure 8: Distribution of $\mathcal{F}$ values by test oracle

The $\mathcal{F}$ values for each test case are summarized as box-plots in Figure 8. A box-plot is a concise representation of multiple data distributions; each distribution is shown as one box. The top and bottom edges of each box mark the third and first quartiles respectively. The plus sign inside the box marks the mean value and the horizontal line inside the box (it sometimes overlaps with the first/third quartile) marks the median value. The whiskers extend from the quartiles and cover 90% of the distribution. The remaining data-points (10%) are considered outliers and are shown as dots beyond the whiskers.

There are four box-plots in Figure 8, one for each subject application. For example, Figure 8(a) shows the results for TerpPresent. This plot contains six boxes, corresponding to the six test oracles. The x-axis lists the oracles and the y-axis shows the $\mathcal{F}$ values. From visual examination of the graph, we see that L2 (mean $\mathcal{F}$ value = 5) does better than L1 (mean $\mathcal{F} = 1$). However, L3 (mean $\mathcal{F} = 6$) is very close to L2. Comparing L4, L5, and L6, we note that the difference between L4 and L5 is not as stark as the difference between L1 and L2; moreover, L6 does better than L5 (which was not the case for L2 vs. L3). Comparison of L1 to L4 (mean $\mathcal{F} = 0.5$) shows that L1 does better than L4. Similarly L2 does better than L5 (mean $\mathcal{F} = 3.5$). However, L3 and L6 are very close. The results for the other applications are more or less similar; the only visual difference is that L3 does better than L6 for these applications.

In summary, visual examination of the box-plots suggest that the "effectiveness order" of test oracles (as measured by their mean $\mathcal{F}$ values) is {L3, L6, L2, L5, L1, L4}, *i.e.*, L3 is the best and L4 is the worst. This result suggests that the oracle information and execution frequency does have an impact on fault-detection effectiveness. Checking the entire state as opposed to only the active window is effective if the oracle is invoked after the last event in the test case. If, on the other hand, the oracle is invoked after each event, then checking only the active window does well. With the exception of TerpPresent, checking the current widget seems ineffective. The characteristics of GUIs that lead to these results will be discussed in detail in Section 4.6.

As demonstrated above, box-plots are useful to get an overview of data distributions. However, valuable information is lost in creating the abstraction. For example, it is not clear *how many* test cases detected specific numbers of faults. This is important to partially address Q4. Even though L3 and L6 more or less showed similar results in the box-plots, *do more test cases* detect more faults with L3 than L6? If this is the case, a tester who has a small number of test cases may get better results with L3 and L6.

We now show the number of test cases that detected specific numbers of faults for different test oracles. Figure 9 shows six histograms for TerpPresent, one for each test oracle. The x-axis represents the $\mathcal{F}$ values; the y-axis shows the number of test cases that had the particular $\mathcal{F}$ values. There are several important points to note about these plots. First, they have an $\mathcal{F} = 0$ column (the first dark column; in some cases this column is very tall; in these cases, it has been chopped – the number adjacent to the top of the column represents its height); this column is important since it accounts for test cases that detected faults with at least one test oracle but not with the current oracle. Second, the sum of all the columns is equal to the number of test cases in the "filtered" data table.

To allow easy visual comparison, we have used the same x-axis and y-axis scales for all six plots. For TerpPresent, we see that a larger number of test cases have a larger $\mathcal{F}$ value for L3 and L6. In fact, the zero column for L3 and L6 contains no test cases, *i.e.*, all test

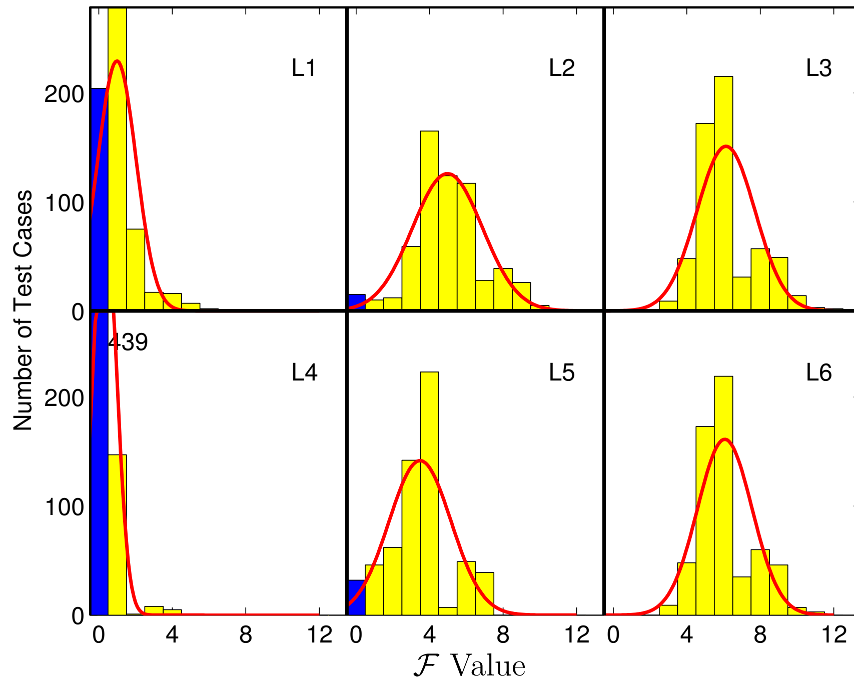


Figure 9: Histogram for TerpPresent

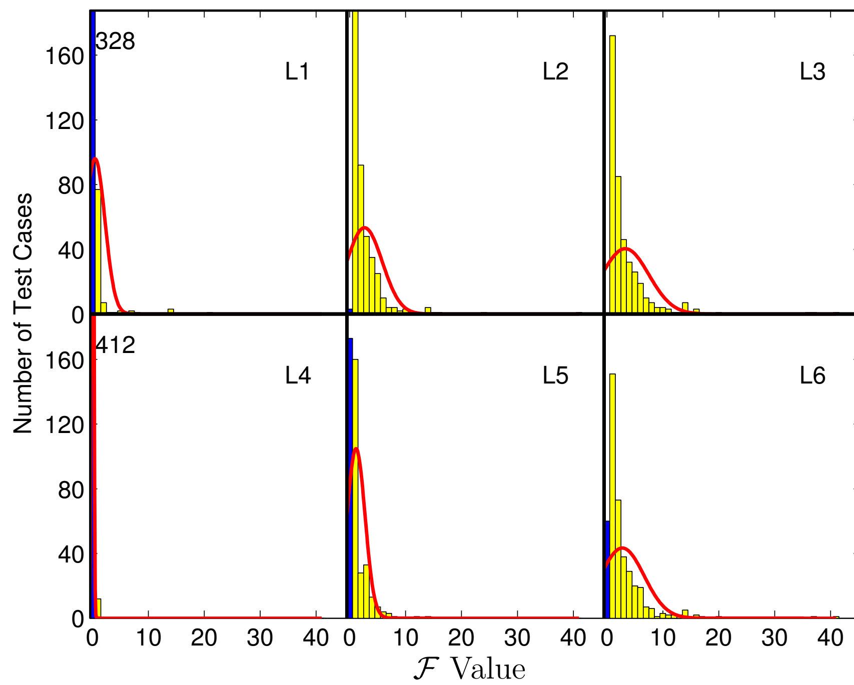


Figure 10: Histogram for TerpWord

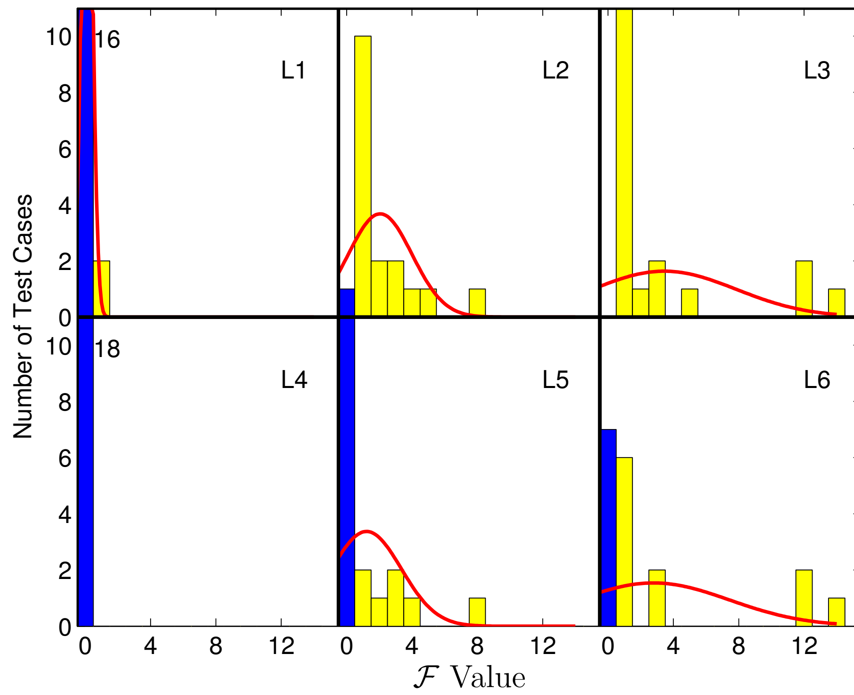


Figure 11: Histogram for TerpPaint

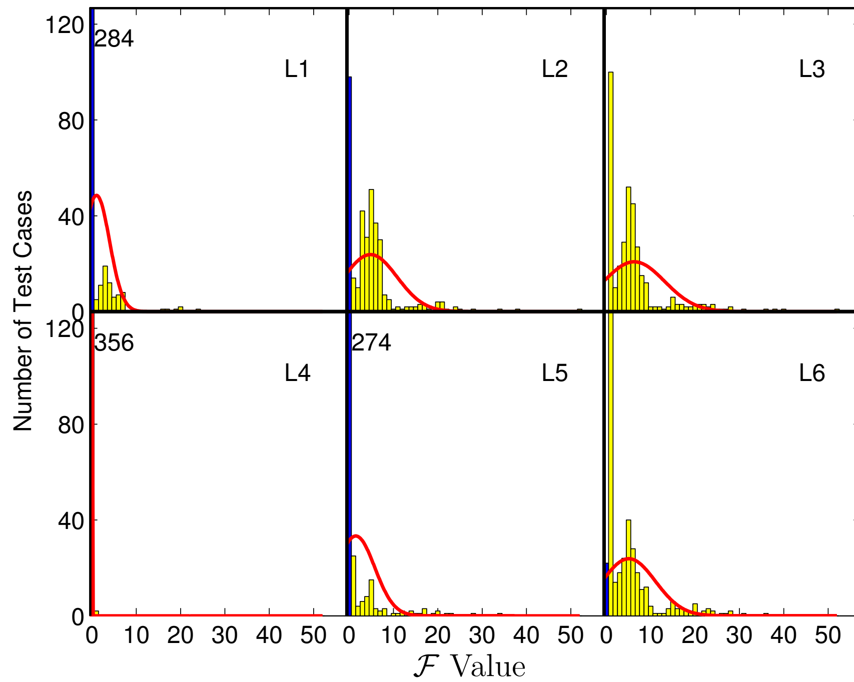


Figure 12: Histogram for TerpSpreadSheet

cases detected at least one fault when using L3 and L6. The zero column is tallest for L4, followed by L1. Hence a large number of test cases did not detect even a single fault when using L1 and L4. In case of TerpWord (Figure 10), approximately 60 test cases did not detect even a single fault for L6. Moreover, the column corresponding to $\mathcal{F} = 1$ for oracle L3 is shorter than that of L2; however, a larger number of test cases have higher $\mathcal{F}$ values. For TerpPaint (Figure 11), the oracle L4 detected no faults, represented by a single zero column of height 18. For TerpSpreadSheet (Figure 12), L3 did significantly better than L2, indicated by a taller $\mathcal{F} = 1$ column; L2 has a very tall $\mathcal{F} = 0$ column.

The probability that a test case will detect a larger number of faults with L3 is high. We also note that oracle L6 does reasonably well. Oracle L4 has the largest number of test cases with zero faults detected. In summary, a tester with a small number of test cases can improve overall fault detection effectiveness by using oracle L3. This result partly answers Q4.

4.5.2 Statistical Analysis

The results discussed thus far have been based on visual examination of the data. While visual examination provides an intuitive overview of the data, valuable information is lost. For example, each test case has six data-points (the six $\mathcal{F}$ values) that are correlated. This correlation is difficult to show and compare visually, especially for large data-sets.

We now want to determine whether the differences in $\mathcal{F}$ values observed for each test case per test oracle are statistically significant. In particular, we want to study the differences between the oracles within the sets {L1, L2, L3}, {L4, L5, L6}, {L1, L4}, {L2, L5}, and {L3, L6}. Several statistical tests may be used for this study. Choosing the right test is based on the number and the nature of the dependent (in this case the $\mathcal{F}$ values) and the independent variables (*i.e.*, the test oracle). For this experiment, the distribution of the data (Normal vs. non-Normal), the number of groups (2 or 3), size of groups, and whether the groups are matched or not will be considered.

These factors and their impact on choice of statistical tests are briefly discussed next. Many statistical tests are based upon the assumption that the data is normally distributed. These tests are called *parametric* tests; common examples include the t-test and analysis of variance (ANOVA) [37]. Since all the computations in parametric tests are based on the data normality assumption, the significance level (p-value) cannot be considered reliable when the data distribution is not normal. Tests that do not make assumptions about distribution are referred to as *non-parametric* tests; these tests rank the dependent variable from low to high and then analyze the ranks. Examples of non-parametric tests include Wilcoxon, Mann-Whitney, and Friedman [37].

Another factor that plays a role in choice of statistical tests is sample size. If the sample size is small, choosing a parametric test with non-normally distributed data, or choosing a non-parametric test with normally distributed data, will yield an inaccurate p-value. On the other hand, if the sample size is large (*e.g.*, more than 2000), the distribution can be considered normal based upon the central limit theorem [37]. Consequently, in practice, with large sample size, the choice of parametric vs. non-parametric does not matter, although the p-value tends to be large for non-parametric tests; however, the discrepancy is small.

There are several ways to formally determine normality of data. A popular way is to use

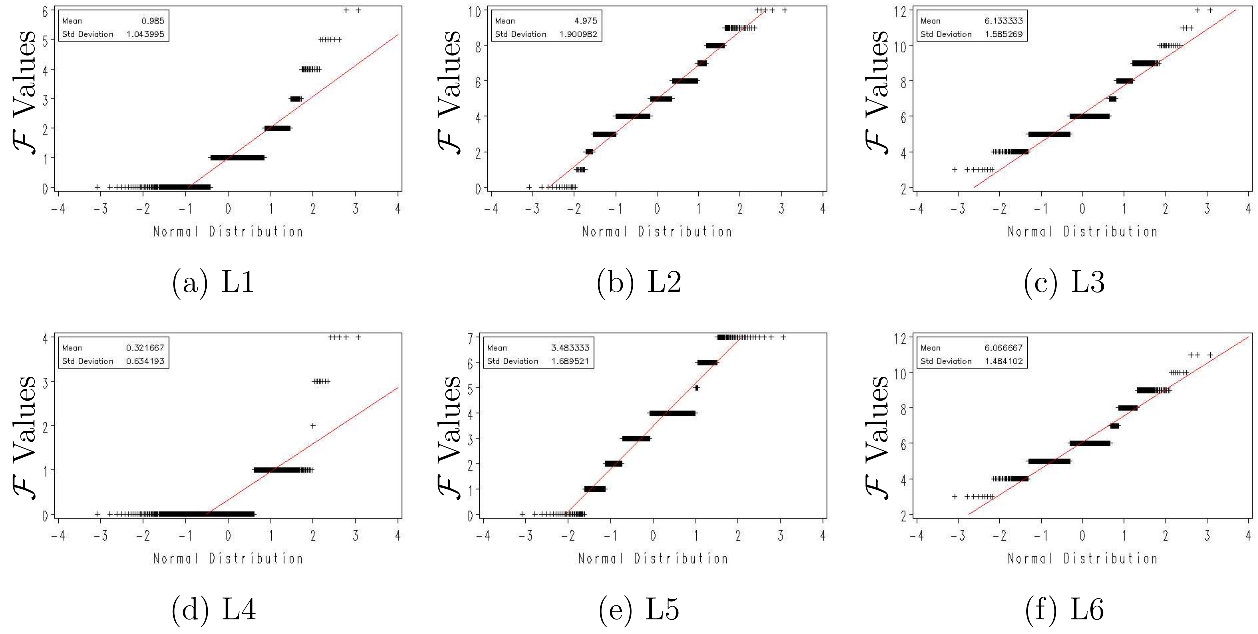
a quantile-quantile (QQ) plot. A quantile is the fraction (or percentage) of points below a given value. For example, 70% of the data fall below the "0.7 quantile" point. A QQ-plot shows the quantiles of a data set (in our case the $\mathcal{F}$ values using different oracles) against the quantiles of a second data set (normal distribution). A mark (x,y) on the plot means that $f_2(x)$ and $f_1(y)$ share the same quantile, where f_1 and f_2 are the functions for the first and second data set. If the two sets come from a population with the same distribution, the plot will be along a $(x = y)$ straight line. The greater the departure from this reference line, the greater the evidence for the conclusion that the two data sets have come from populations with different distributions [42]. Figure 13 through Figure 16 show the quantile-quantile plots for each application. The x-axis is the normal distribution; the y-axis is the $\mathcal{F}$ distribution. Since the standard deviation for the $\mathcal{F}$ values is 0 for TerpPaint with oracle L4, there is no QQ-plot for this instance. None of the QQ-plots show a straight line, implying non-normality.

Since our sample sizes are small (*e.g.*, 18 for TerpPaint), we need to determine normality of the data before we choose the statistical tests. For illustration, the solid line superimposed on the histograms (Figures 9 through 12) shows the normal distribution approximation; this illustration suggests that the data is not normal. Finally, our data is matched, *i.e.*, each data point (*e.g.*, $\mathcal{F}$ value for oracle L1 with test case T) in one distribution (for oracle L1) has a corresponding matched point in all other distributions (the matched points are the $\mathcal{F}$ values for oracles L2–L6 with test case T). Considering all these factors, we chose the Friedman test for the three matched groups statistical comparison ({L1, L2, L3}, {L4, L5, L6}) and Wilcoxon signed ranks test for two matched groups comparison ({L1, L4}, {L2, L5}, {L3, L6}). We will not run a test to compare {L1, L4} for TerpPaint.

	Sample Size	Friedman Test	Statistic Value	P-Value
TerpPresent	1800	L1/L2/L3	1095.2514	<.0001
		L4/L5/L6	1174.8991	<.0001
TerpWord	1272	L1/L2/L3	710.9736	<.0001
		L4/L5/L6	577.6345	<.0001
TerpPaint	54	L1/L2/L3	31.0000	<.0001
		L4/L5/L6	18.0000	0.0001
TerpSpreadSheet	1074	L1/L2/L3	542.5014	<.0001
		L4/L5/L6	598.8733	<.0001

Table 4: Friedman Test Results.

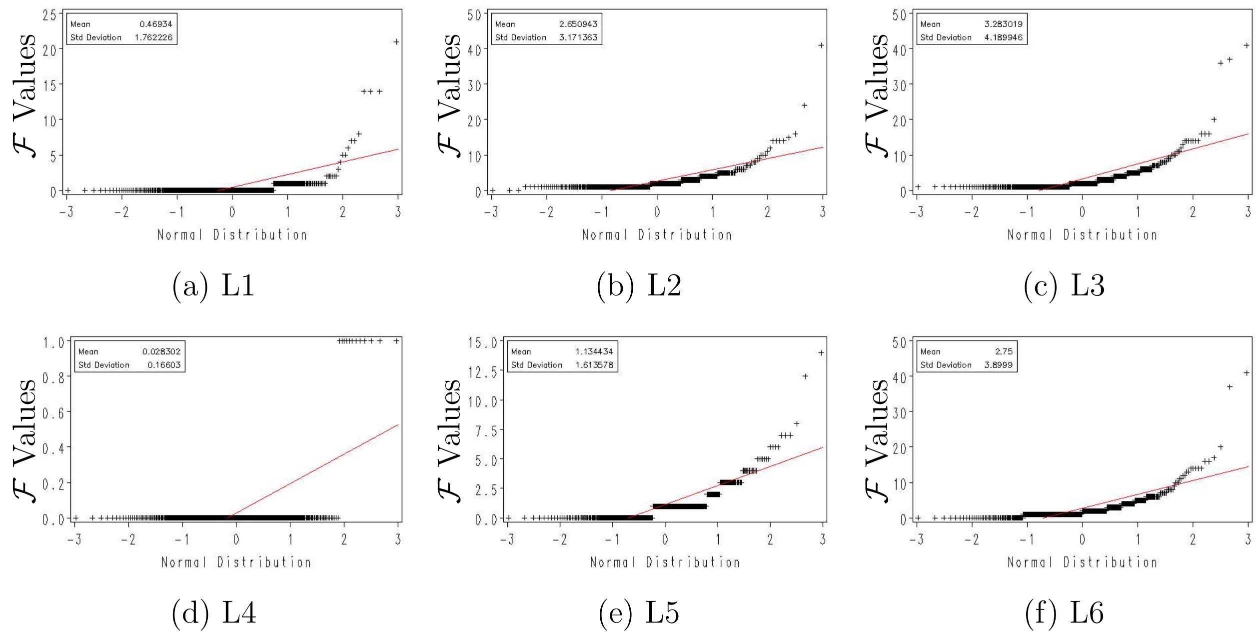
Friedman Test: This test compares the mean $\mathcal{F}$ values for the test oracle sets {L1, L2, L3}, and {L4, L5, L6} based on their rank scores. The null hypothesis here is that the mean values do not differ. Table 4 summarizes the results of this test. The statistic value shown here is the standard Cochran-Mantel-Haenszel (CMH) statistic used by most popular statistical software packages. The p-values are obtained by a table lookup using the sample size and CMH value. As the result shows, all p-values are less than 0.05. Hence, the null hypothesis is rejected. The alternative hypothesis, *i.e.*, the mean $\mathcal{F}$ values do differ in a statistically significant way, is accepted. An additional Wilcoxon matched pairs test on the oracle pairs {L1, L2}, {L2, L3}, {L1, L3}, {L4, L5}, {L5, L6}, and {L4, L6} showed that the differences between these oracle pairs are also statistically significant.



(a) L1 (b) L2 (c) L3

(d) L4 (e) L5 (f) L6

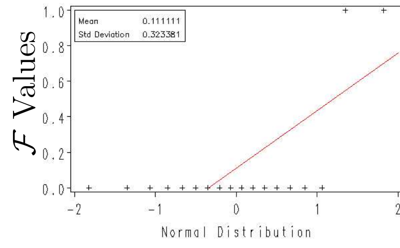
Figure 13: Quantile-Quantile plot for TerpPresent



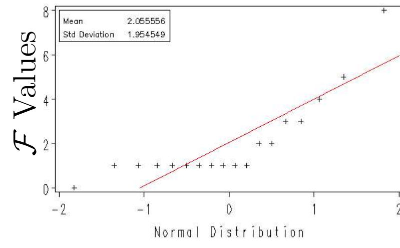
(a) L1 (b) L2 (c) L3

(d) L4 (e) L5 (f) L6

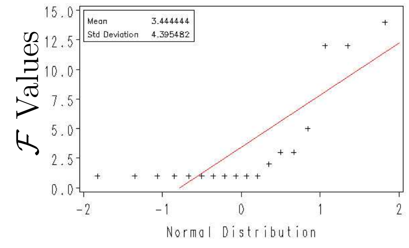
Figure 14: Quantile-Quantile plot for TerpWord



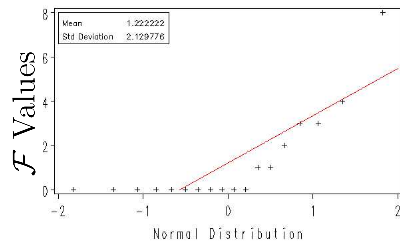
(a) L1



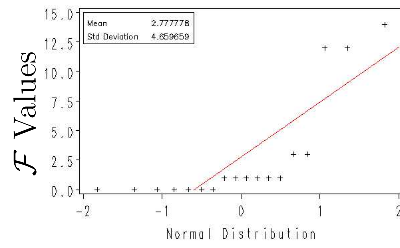
(b) L2



(c) L3

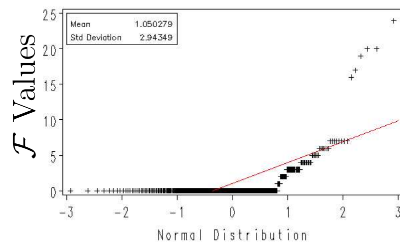


(e) L5

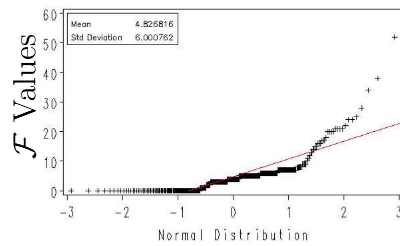


(f) L6

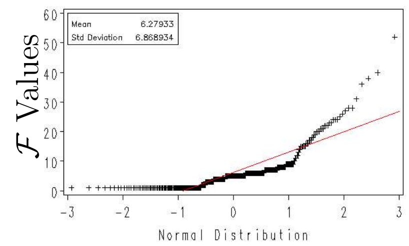
Figure 15: Quantile-Quantile plot for TerpPaint



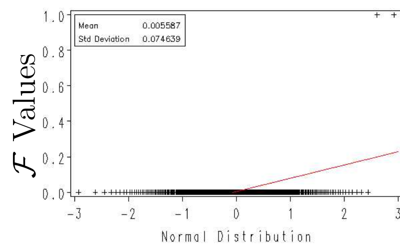
(a) L1



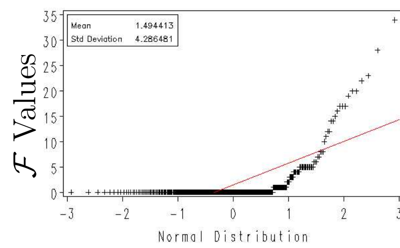
(b) L2



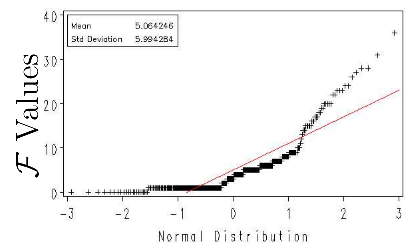
(c) L3



(d) L4



(e) L5



(f) L6

Figure 16: Quantile-Quantile plot for TerpSpreadSheet

	Sample Size	Wilcoxon Test	Statistic Value	P-Value
TerpPresent	**600**	L1/L4	20808	<.0001
		L2/L5	33764	<.0001
		L3/L6	115.5	<.0001
TerpWord	**424**	L1/L4	1785	<.0001
		L2/L5	17490	<.0001
		L3/L6	6440	<.0001
TerpPaint	**18**	L1/L4	*	*
		L2/L5	27.5	0.0020
		L3/L6	14	0.0156
TerpSpreadSheet	**358**	L1/L4	1387.5	<.0001
		L2/L5	10764	<.0001
		L3/L6	1870.5	<.0001

Table 5: Wilcoxon Test Results.

	L1	L2	L3	L4	L5	L6
TerpPresent	11.5	567.69	947.6	1	48.51	82.56
TerpWord	12.93	404.79	671.56	1	30.29	59.56
TerpPaint	14.17	1106.56	1667.17	1	63.33	125.83
TerpSpreadSheet	13.61	600.29	1268.06	1	43.24	104.7

Table 6: Average Number of Widget Comparisons Per Test Case.

Wilcoxon Signed Ranks Test: The null hypothesis here is that there is no statistically significant difference between the means among the oracles in the sets {L1, L4}, {L2, L5}, and {L3, L6}. The results of the tests are summarized in Table 5. All p-values are less than 0.05, resulting in the rejection of the null hypothesis and acceptance of the alternative hypothesis.

The above two analyses helped us to answer the first parts of Q1 and Q2. Based on the results of the Friedman test, and the earlier visual comparison, we conclude that the oracle information has a significant impact on fault detection effectiveness of a test case; checking more widgets is beneficial. Based on the results of the Wilcoxon signed ranks test, and the earlier visual observations, we conclude that the frequency of invoking the test oracle does have a significant impact on the fault detection effectiveness of a test case; invoking the test oracle frequently is beneficial.

4.5.3 Faults Detected Per Comparison

We now address the issue of the cost of the oracles. First we look at the number of comparisons that each oracle performs per test case. The average number of comparisons per test case for oracle L is represented as $\Delta(L)$, and is shown in Table 6. As expected, $\Delta(L4) = 1$. The value of $\Delta(L1)$ is larger than $\Delta(L4)$ due to one comparison per event in the test case. The values of $\Delta(L2)$ and $\Delta(L3)$ depend on the number of widgets in the active window and in all the open windows respectively. Similarly, $\Delta(L5)$ and $\Delta(L6)$ depend on the number of widgets in the active window and in all the open windows when the test case ends, respectively.

Recall that we have defined ξ as the faults-detected-per-comparison for each test case.

Higher values of ξ are considered better. We now compute ξ and present the results as box-plots.

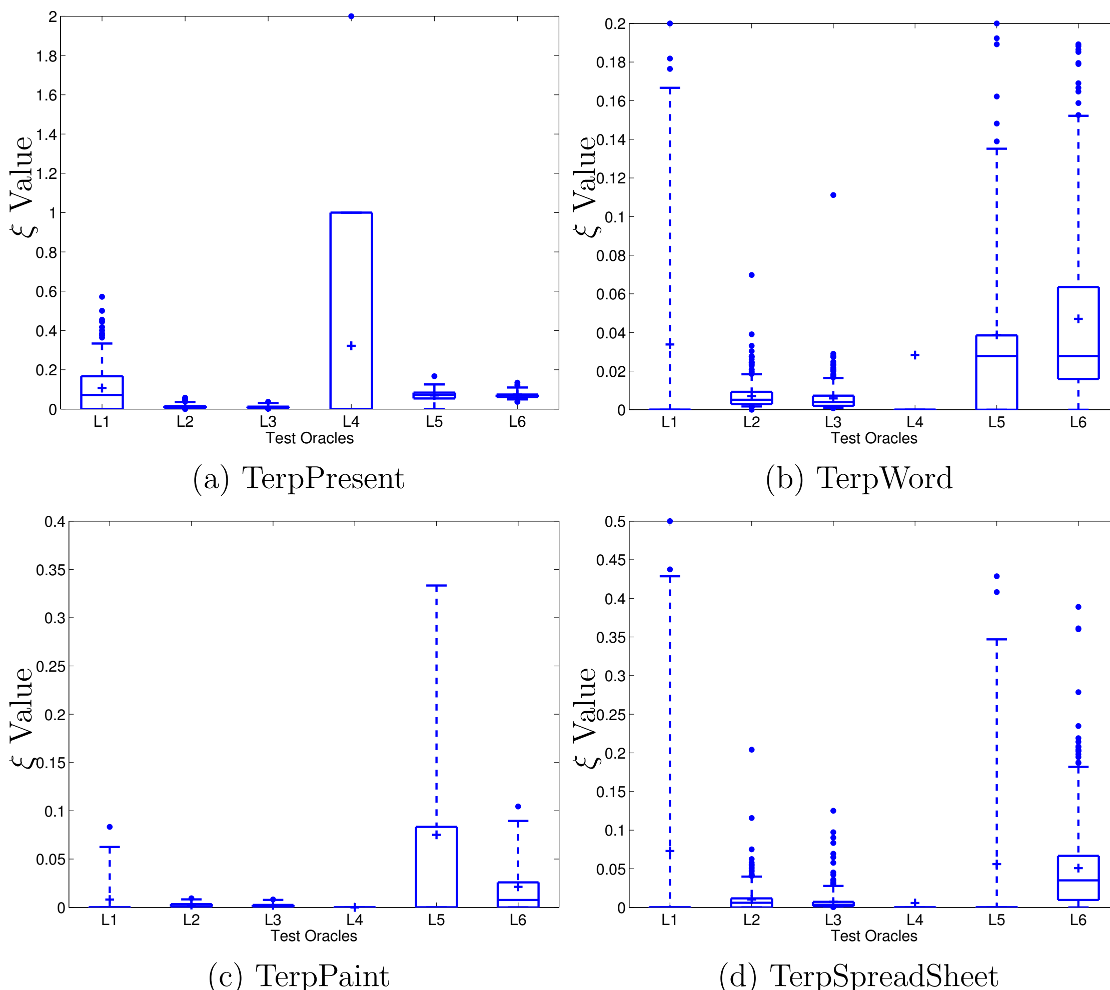


Figure 17: ξ values for all test cases.

The results for TerpPresent are summarized in Figure 17(a). Since L3 requires the maximum number of comparisons (the entire state of the GUI after each event in the test case), it is penalized the most by the ξ measure; L2 is close behind. Since the number of comparisons is smaller for L5 and L6, their ξ values are better. In the case of TerpPresent, checking the widget alone helped to detect a non-trivial number of faults; combined with a very small number of comparisons required, the ξ value of L4 was better than all other oracles, followed by L1.

The results for TerpWord (Figure 17(b)) are different primarily because L1 and L4 did not detect many faults; simply checking the widget was inadequate. L2 and L3 again suffered due to the large number of comparisons they require. L5 did much better due to its reduced frequency of comparison. Although L6 compares the entire state whereas L5 compares only the active window, L6 did much better due to its larger $\mathcal{F}$ value. This difference did not help

L6 for TerpPaint (Figure 17(c)) since the entire state is much larger for this application. Since L5 did not detect many faults for TerpSpreadSheet, its ξ value is very low (Figure 17(d)).

We are now ready to answer Q1, Q2, and Q3. In case of Q1, we saw that the oracle information does have an impact on the fault-detection effectiveness of a test case. In case of Q2, the invocation frequency of a test oracle has a very significant impact on the fault-detection effectiveness of a test case. Considering the ξ measure, the additional effectiveness is not worth the cost for L2 and L3 due to the extremely large number of comparisons required for L2 and L3; using L5 and L6 is more practical. However, for L1 vs. L4, the additional cost is very low and helps fault detection.

In case of Q3, the combination of oracle information and procedure that provides the best cost-benefit ratio depends largely on the GUI. We will discuss details of GUI characteristics in Section 4.6.

4.5.4 Relationship Between Test Oracles and Fault-Detection Position

We observed that whenever the test oracles L3, L2, and L1 detected a fault at event position a, b, and c respectively, then in many cases (*e.g.*, 33% for TerpWord, 62% for TerpPresent) one of the relationships $a < b$ or $b < c$ held (we had expected $a = b = c$). In other words, when oracles contained more information, they tended to detect faults earlier in the event sequence.

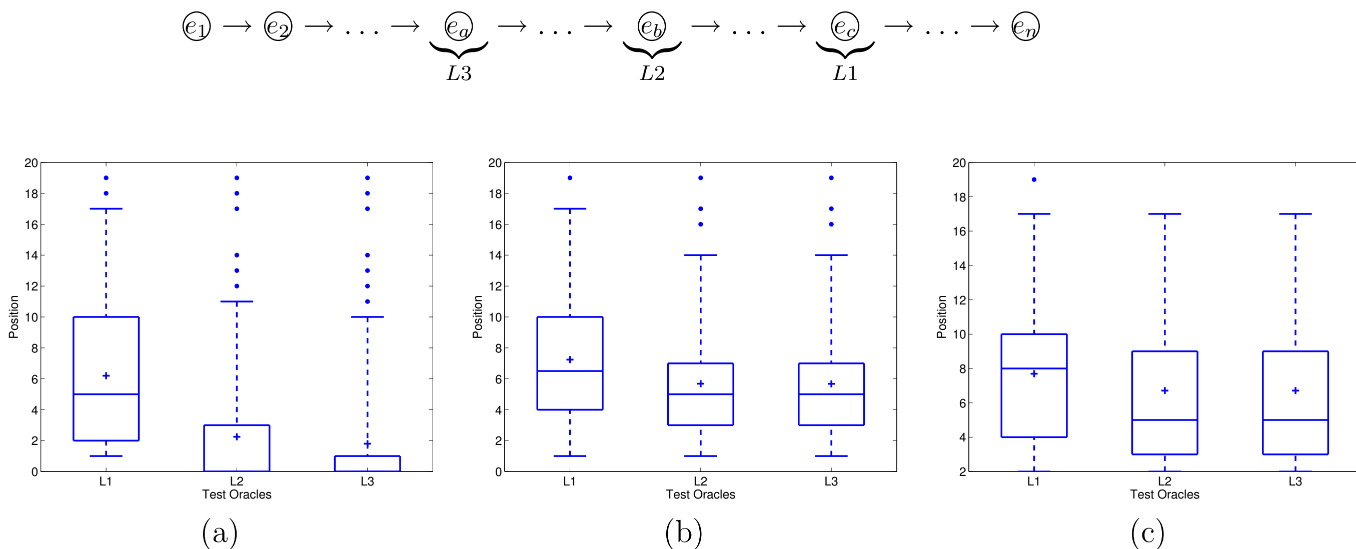


Figure 18: Position where the fault is detected vs. oracle for (a) TerpPresent, (b) TerpWord, and (c) TerpSpreadSheet.

This was an interesting result since it provided a link between test oracles and the length of a test case. Longer test cases are more expensive to generate and execute. Hence, if a test designer has a suite containing short test cases, oracle L3 has better chances of detecting more faults. The box-plots shown in Figure 18 illustrate the results. No results are shown for TerpPaint since only two test cases detected a fault using L1. The box-plots show that the position at which the fault is detected using L1 is later than that using L2 or L3. However,

for TerpWord and TerpSpreadSheet, the position at which the fault is detected using L2 is almost the same as that using L3.

Hence generating/using a complex (more expensive) oracle is justified if a tester has short test cases. This result partly answers Q4.

4.6 Discussion

Our results are consistent with the characteristics of today's GUIs. First, GUIs contain several types of widgets. Some of these widgets have a state (*e.g.*, check-boxes, radio-buttons) whereas others are stateless (*e.g.*, buttons, pull-down menus). Events (such as clicking on a check-box) performed on state-based widgets are used to change (usually toggle) their state. A test oracle that checks the correctness of the state of the *current* widget (*i.e.*, on which an event was just executed) is able to detect specific types of faults – ones that may adversely affect the current widget's state only; other faults are missed. TerpPresent has many such faults. L1 is an example of this type of oracle. Second, many events affect the state of multiple widgets of the active window, not just the current widget. L2 is able to detect all faults that are manifested anywhere on the active window. Finally, several events affect the state of the entire GUI. For example, `OK` in "preferences setting" has a global impact on the overall GUI. L3 is able to detect faults in such events.

The frequency of oracle invocation has a significant impact on fault detection effectiveness since the constantly changing structure (*e.g.*, currently open windows, active window) of the executing GUI provides a small "window of opportunity" for fault detection. A test oracle (such as L1 or L4) that examines only the current widget, if not invoked immediately after a faulty widget state is encountered, will fail to detect the problem. Hence L4, which waits until the last event, to examine the then-current widget detects fewer faults than L1. L4 is successful only if the widget associated with the test case's last event is problematic, as was the case with TerpPresent. Similarly, L5 detects fewer faults than L2 because a faulty active window is either closed or is no longer the active window by the time the last event in the test case executes; L5 misses these faults. On the other hand, L2 is able to detect such faults immediately as they are manifested on the active window. The small difference between L3 and L6 is due to the windows/widgets that are available at any time for examination. Errors that persist anywhere (*i.e.*, in any window or widget) across the entire test case execution are easily detected by L6 since it examines the entire state of the GUI after the last event. L6 misses only those errors that occurred in windows that were later closed or "disappeared" due to other reasons. The small number of such disappearing errors in TerpWord, TerpPaint, and TerpSpreadSheet show the reduced impact of comparing the entire state after each event.

The cost of test oracles is directly related to GUI layout issues that stem from usability concerns. Factors that impact the cost of our test oracles include the number of windows in the GUI that are open at any time (since L3 and L6 compare a larger number of widgets) and the number of widgets per window (since L2 and L5 compare all the widgets in the active window).

There are several lessons-learned for GUI developers and test designers. First, testers who use capture/replay tools typically create assertions for very few widgets after each event (*e.g.*, the one on which the current event is being executed). Seeing that L1 and L4 were the least effective at detecting faults, testers need to capture more information with their test

cases, perhaps by using a reverse engineering tool such as ours; use of such automated tools will also reduce the overall effort required to create these oracles. Second, since it is difficult and expensive to create many long GUI test cases, testers who conserve their resources and create few short test cases should use test oracles such as L3 and L6 that check a more complete state of the GUI to improve fault-detection effectiveness. Third, testers should realize that the dynamic nature of GUIs provides a small window of opportunity to detect faults. They should place their assertions at strategic places in the test case (*e.g.*, before a window/menu is closed) to maximize fault-detection effectiveness. Finally, GUI designers must realize that their decisions will not only have an impact on usability but also on its "testability."

5 Conclusions

In this paper, we showed that *test oracles* play an important role in determining the effectiveness and cost of the testing process. We defined two important parts of a test oracle: *oracle information* that represents expected output, and an *oracle procedure* that compares the oracle information with the actual output. We described a technique to specify different types of test oracles by varying the level of detail of oracle information and changing the oracle procedure; we employed this technique to create six instances of test oracles for an experiment. The results show that test oracles do affect the fault-detection ability of test cases in different and interesting ways: (1) test cases significantly lose their fault-detection ability when using "weak" test oracles, (2) in many cases, invoking a "thorough" oracle at the end of test case execution yields the best cost-benefit ratio, (3) certain test cases detect faults only if the oracle is invoked during a small "window of opportunity" during test execution, and (4) using thorough and frequently-executing test oracles can make up for not having long test cases.

The results in this paper open several new research opportunities, some of which we outline next:

• We feel that the GUI domain was an ideal starting point for this type of study since the way we define a GUI oracle, in terms of objects (widgets) and their properties that change over time, allows us to "fine-tune" the oracle information and procedure. Our results may be applicable to all event-based software that can be modeled in terms of objects, properties, and their values (*e.g.*, object-oriented software). Test oracles for such software would determine whether the objects executed correctly. In the future, we will extend our pool of subject applications to include non-Java and non-GUI programs.

• We used a technique to generate one class of test cases. This technique is based on a traversal of a graph representation of the GUI, namely event-flow graphs [36]. In the future, we will generate other classes of test cases using techniques such as AI planning [33] and capture/replay tools, and observe the effect of using different test oracles with these test cases.

• Since we have identified differences in fault-detection ability of different test oracles, which is clearly linked to the number of GUI objects that the oracles "cover," we will develop adequacy criteria for test oracles in a way similar to those already available for test cases [57].

• We feel that the relationship that was revealed between test case length and fault-

detection effectiveness is significant and requires further study. In the future, we intend to conduct a detailed experiment involving a large number of test cases of varying length; we will also model the relationship between the length of these test cases and the faults that they reveal.

• The simplifying assumption to minimally model the back-end using fixed data allowed us to focus on issues relevant to the GUI only. In the future, we will extend our back-end models, seed faults in the back-end code, and study the effectiveness of GUI test oracles on back-end faults.

• We currently modeled one type of assertion, *i.e.*, check for equality between expected and actual widget properties. In the future, we will extend our test oracles with additional types of assertions, *e.g.*, those that specify timing issues, range checking, etc.

Acknowledgments

The authors thank the anonymous reviewers of the original conference paper [31] whose comments and suggestions also helped to improve this paper's presentation. The anonymous reviewers of this journal version played an important role in reshaping the experimental results and statistical analyses. The authors also thank Adithya Nagarajan and Ishan Banerjee who helped to lay the foundation for this research. This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

References

[1] Antoy, S., and Hamlet, D. Automatically checking an implementation against its formal specification. *IEEE Trans. Softw. Eng. 26*, 1 (2000), 55–69.

[2] Baresi, L., and Young, M. 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] Barringer, H., Goldberg, A., Havelund, K., and Sen, K. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)* (Jan. 2004), Springer-Verlag, pp. 44–57.

[4] Bernot, G., Gaudel, M. C., and Marre, B. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J. 6*, 6 (1991), 387–405.

[5] Blackburn, M., Busser, R., and Nauman, A. Interface-driven model-based test generation of java test drivers. In *Proceedings of the 15th International Software/Internet Quality Week Conference* (2002).

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

Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03) (June 2003), pp. 73–89.

[7] Cheon, Y., and Leavens, G. T. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK, 2002), Springer-Verlag, pp. 231–255.

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

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

[10] Doong, R.-K., and Frankl, P. G. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol. 3*, 2 (1994), 101–130.

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

[12] Elbaum, S., Karre, S., and Rothermel, G. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 49–59.

[13] Finsterwalder, M. Automating acceptance tests for gui applications in an extreme programming environment. In *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering* (May 2001), pp. 114 – 117.

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

[15] Gannon, J., McMullin, P., and Hamlet, R. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst. 3*, 3 (1981), 211–223.

[16] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol. 10*, 2 (2001), 184–208.

[17] Green, B. S. Software test automation. *SIGSOFT Softw. Eng. Notes 25*, 3 (2000), 66–66.

[18] Havelund, K., and Rou, G. An overview of the runtime verification tool java pathexplorer. *Form. Methods Syst. Des. 24*, 2 (2004), 189–215.

[19] Hicinbothom, J. H., and Zachary, W. W. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting* (1993), vol. 2 of *SPECIAL SESSIONS: Demonstrations*, p. 1042.

[20] Jagadeesan, L. J., Porter, A., Puchol, C., Ramming, J. C., and Votta, L. G. Specification-based testing of reactive software: tools and experiments: experience report. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (New York, NY, USA, 1997), ACM Press, pp. 525–535.

[21] JUnit, Java Unit Testing Framework, 2003. Available at `http://junit.sourceforge.net`.

[22] Kepple, L. R. The black art of GUI testing. *Dr. Dobb's Journal of Software Tools 19*, 2 (Feb. 1994), 40.

[23] Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. Mac: A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, Dept. of Computer and Information Science, 1998.

[24] Luckham, D., and Henke, F. W. An overview of anna - a specification language for ada. *IEEE Softw. 2*, 2 (1985), 9–22.

[25] Marick, B. When should a test be automated? In *Proceedings of the 11th International Software/Internet Quality Week* (May 1998).

[26] Marick, B. Bypassing the GUI. *Software Testing and Quality Engineering Magazine* (Sept. 2002), 41–47.

[27] Memon, A., Banerjee, I., and Nagarajan, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering* (2003), IEEE Computer Society, pp. 260–269.

[28] Memon, A., Nagarajan, A., and Xie, Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice 17*, 1 (2005), 27–64.

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

[30] Memon, A. M. Advances in GUI testing. In *Advances in Computers, ed. by Marvin V. Zelkowitz*, vol. 57. Academic Press, 2003.

[31] Memon, A. M., Banerjee, I., and Nagarajan, A. What test oracle should I use for effective GUI testing? In *Proceedings of the IEEE International Conference on Automated Software Engineering* (Oct.12–19 2003), IEEE Computer Society, pp. 164–173.

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

[33] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering 27*, 2 (Feb. 2001), 144–155.

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

[35] Memon, A. M., Soffa, M. L., and Pollack, M. E. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)* (Sept. 2001), pp. 256–267.

[36] Memon, A. M., and Xie, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering 31*, 10 (Oct. 2005), 884–896.

[37] Motulsky, H. *Intuitive Biostatistics*. Oxford University Press, 1995.

[38] Myers, B. A. User interface software tools. *ACM Transactions on Computer-Human Interaction 2*, 1 (1995), 64–103.

[39] Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)* (New York, Mar.2–5 1998), ACM Press, pp. 82–92.

[40] Peters, D., and Parnas, D. L. Generating a test oracle from program documentation: work in progress. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1994), ACM Press, pp. 58–65.

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

[42] Quantile-Quantile Plot. http://www.itl.nist.gov/div898/handbook/eda/section3/qqplot.htm.

[43] Richardson, D. J. Taos: Testing with analysis and oracle support. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1994), ACM Press, pp. 138–153.

[44] Richardson, D. J., Leif-Aha, S., and O'Malley, T. O. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering* (May 1992), pp. 105–118.

[45] Robinson, H. Using pre-oracled data in model-based testing, July 1999. Available at http://www.geocities.com/harry_robinson_testing/pre-oracled.htm.

[46] Rosenblum, D. S. Specifying concurrent systems with TSL. *IEEE Softw. 8*, 3 (1991), 52–61.

[47] Rosenblum, D. S. Towards a method of programming with assertions. In *ICSE '92: Proceedings of the 14th international conference on Software engineering* (New York, NY, USA, 1992), ACM Press, pp. 92–104.

[48] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P., and Qiu, X. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol. 13*, 3 (2004), 277–331.

[49] Siepmann, E., and Newton, A. R. TOBAC: Test Case Browser for Object-Oriented Software. In *Proc. International Symposium on Software Testing and Analysis* (New York, Aug. 1994), ACM Press, pp. 154–168.

[50] Software Research, Inc., capture-Replay Tool, 2003. Available at `http://soft.com`.

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

[52] Vieira, M. E., Dias, M. S., and Richardson, D. J. Object-oriented specification-based testing using UML statechart diagrams. In *Proceedings of the Workshop on Automated Program Analysis, Testing and Verification at* (June 2000).

[53] Vogel, P. A. An integrated general purpose automated test environment. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1993), ACM Press, pp. 61–69.

[54] Weyuker, E. J. On testing non-testable programs. *Comput. J. 25*, 4 (1982), 465–470.

[55] White, L., AlMezen, H., and Alzeidi, N. User-based testing of gui sequences and their interactions. In *Proceedings of the 12th International Symposium Software Reliability Engineering* (2001), pp. 54 – 63.

[56] Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. *Experimentation in software engineering: an introduction.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[57] Zhu, H., Hall, P., and May, J. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec. 1997), 366–427.