What Test Oracle Should I Use for Effective GUI Testing?

Atif Memon
Department of Computer Science
and Fraunhofer Center for
Experimental Software Engineering
University of Maryland
College Park, Maryland, USA
atif@cs.umd.edu

Ishan Banerjee, Adithya Nagarajan
Department of Computer Science
University of Maryland
College Park, Maryland, USA
{ishan, sadithya}@cs.umd.edu

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. In this paper we show that the test oracle used during testing also contributes significantly to test effectiveness and cost. A test oracle is a mechanism that determines whether a software executed correctly for a test case. We define a test oracle to contain two essential parts: oracle information *that represents expected output, and an* oracle procedure *that compares the oracle information with the actual output. By varying the level of detail of oracle information and changing the oracle procedure, a test designer can create different types of test oracles. We design 11 types of test oracles and empirically compare them on four software systems. We seed faults in each software to create 100 faulty versions, execute 600 test cases on each version, for all 11 types of oracles. In all, we report results of 660,000 test runs on each software. We show (1) the time and space requirements of the oracles, (2) that faults are detected early in the testing process when using detailed oracle information and complex oracle procedures, although at a higher cost per test case, and (3) that employing expensive oracles results in detecting a large number of faults using relatively smaller number of test cases.*

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

1 Introduction

Software testing is an important software engineering activity widely used to find defects in programs. During the testing process, *test cases* are executed on an *application under test* (AUT) and *test oracles* are used to determine whether the AUT executed as expected [1]. The test oracle may either be automated or manual; in both cases, the actual output is compared to a presumably correct expected output.

Testers widely believe that the overall effectiveness of the testing process depends largely on the number and type of test cases used. Test adequacy criteria are used to compare and evaluate the adequacy of test cases and generate more if needed [18]. Our research has shown that the type of test oracle used also has a significant impact on test effectiveness. There has been no reported work comparing different types of oracles, their impact on fault detection effectiveness and cost in terms of space and time. In this paper, we describe several types of test oracles and empirically show their relative strengths, weaknesses, and costs.

We define a test oracle to contain two parts: *oracle information* that is used as the expected output and an *oracle procedure* that compares the oracle information with the actual output [14]. Different types of oracles may be obtained by changing the oracle information and using different oracle procedures. For example, for testing a spreadsheet, 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, the oracle procedure for a spreadsheet may (a) check for equality between expected and actual cell values, or (b) determine whether a cell value falls within a specified 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.

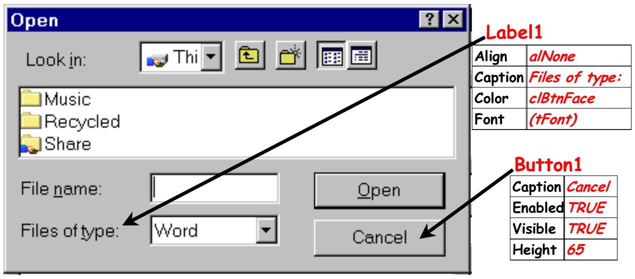
Several researchers have identified the need for different types of oracles, although none have compared them empirically. Notable is the work by Richardson in TAOS [13] who proposes several levels of test oracle support, and Siepmann et al. in their TOBAC system [15] who provide seven ways of implementing the oracle procedure.

In this paper, we define several types of oracle information and procedures and empirically compare them. Our previous work [9] laid the foundation for test oracles of a class of event-based systems; specifically those that have a Graphical User Interface (GUI) frontend. We now leverage the technology to develop and study several types of test oracles. We feel that the GUI domain is ideal 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. We define four types of oracle information in increasing level of detail and cost: *widget*, *active window*, *visible windows*, and *all windows*. The oracle procedure too has several increasing levels of complexity and cost: “check for equality of *widget*, *active window*, *visible window*, *all windows* after each event” and “check *all windows* after the last event” of the test case. (We provide details and examples in Sections 3.1 and 3.2.) Combining the oracle information and oracle procedures gives us 11 different types of oracles. We empirically compare these test oracles on four software systems. We seed faults in each software to create 100 faulty versions, execute 600 test cases on each version, for the 11 types of oracles. In all, we report results of 660,000 test runs for each software.

The results of our experiments show that (1) oracles that use detailed oracle information and complex procedures are expensive both computationally and in terms of space, (2) defects are detected early in the testing process when using an expensive oracle, and (3) using expensive oracles allows catching a large number of defects using relatively smaller number of test cases.

The specific contributions of this work include:

1. a first empirical study comparing test oracles,
2. definition of different levels of GUI oracle information,
3. development of different oracle procedures for GUIs, and
4. guidelines to test designers about designing test oracles, their relative strengths and weaknesses.



Label1

Align	alNone
Caption	Files of type:
Color	clBtnFace
Font	(tFont)

Button1

Caption	Cancel
Enabled	TRUE
Visible	TRUE
Height	65

(a)

State = {(Label1, Align, alNone), (Label1, Caption, “Files of type:”), (Label1, Color, clBtnFace), (Label1, Font, (tfont)), (Form1, WState, wsNormal), (Form1, Width, 1088), (Form1, Scroll, TRUE), (Button1, Caption, Cancel), (Button1, Enabled, TRUE), (Button1, Visible, TRUE), (Button1, Height, 65), …}

(b)

Figure 1. (a) `Open` GUI, (b) its Partial State

In the next section, we define GUI states and test cases. In Section 3 we use these definitions to describe the parts of a GUI test oracle, namely oracle information and oracle procedure. We also describe how our general definition of oracle information and procedure may be used to develop different types of test oracles. In Section 4, we present details of experiments. Finally, we conclude with a discussion of related work in Section 5 and future research opportunities in Section 6.

2 GUI Model

Before we develop the different types of GUI oracles, we define the basic concepts needed to understand their design. We begin by modeling a GUI state in terms of the *widgets* (GUI’s basic building blocks) the GUI contains, their properties, and the values of the properties. We also define GUI *events* (actions performed by the user) and use this definition to develop GUI test cases.

2.1 Widgets, Properties and Values

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

For example, consider the `Open` GUI shown in Figure 1(a). This GUI contains several widgets, two of which are explicitly labeled, namely `Button1` and `Label1`; for each, a small subset of properties is shown. 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 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 `Open` GUI, partially shown in Figure 1(b), contains all the properties of all the widgets in `Open`.

In this research, we extensively use the definition of the state of a GUI to develop the oracle information and procedure. As will be seen later, we associate oracle information with each test case. Hence, we formally define a GUI test case next.

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 change its state. These 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 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 [10]. We restrict our testing to such *legal* event sequences, 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. □

Our concepts 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 $e_1; e_2; \ldots; e_n$. □

Now that we have briefly defined the basic GUI concepts, (the interested reader is referred to [8] for details and examples), we now describe a GUI test oracle.

3 GUI Test Oracle

In earlier work [9], we developed the design of a GUI test oracle shown in Figure 2. We now briefly present the design and extend it to develop different types of oracles.

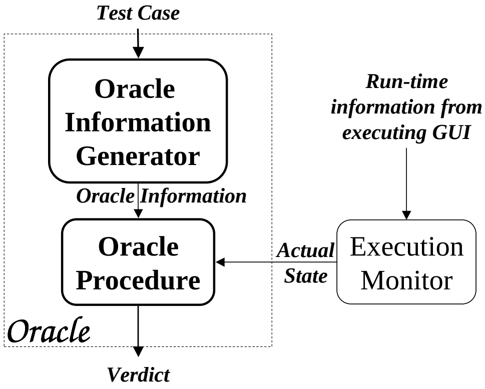


Figure 2. An Overview of the GUI Oracle.

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 [9] or by using a "correct" version of the software [16, 17] (as described in Section 4). Likewise, the *actual state* (also described by a set of widget, property, and value triples) is obtained from an *execution monitor*. The execution monitor may use any of the techniques described in [9], 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.

The above partitioning of functionality allows us to define a simple algorithm for test execution. Given a test case, we execute all its events, compute the expected state, obtain the GUI's actual state, compare the two states, and determine 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, e_1; e_2; $\ldots$; e_n is the event sequence; (2) a set of integers **OPF** (LINE 2) that determines how frequently the oracle procedure is invoked (details in Section 3.2); (3) $\mathbf{C}_{OI}$ (LINE 3) is a boolean constraint used to obtain relevant triples for the oracle information. Examples of some constraints are shown in Section 3.1; (4) $\mathbf{C}_{AS}$ (LINE 4) is a similar boolean constraint but used by the oracle procedure to obtain relevant triples for both the actual and expected 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). The constraint $\mathbf{C}_{OI}$ is used to select a subset of the complete state. This constraint is discussed in Section 3.1. Similarly, the actual state $\mathbf{AS}_i$ of the GUI, also constrained by a $\mathbf{C}_{AS}$ is obtained (LINE 8). The oracle procedure is then invoked (LINE 9) that determines whether the software's execution

```
ALGORITHM :: ExecTestCase(
  T: Test case; /* T = < $S_0$, $e_1$; $e_2$; $\ldots$; $e_n$ > */                 1
  OPF $\subseteq \{1, 2, 3, \ldots, n\}$; /* oracle procedure freq. */              2
  $\mathbf{C}_{OI}$: Boolean Constraint; /* on oracle information */               3
  $\mathbf{C}_{AS}$: Boolean Constraint; /* on actual state */){                   4

  FOREACH $\mathbf{e}_i$ in T DO { /* for all events */                            5
    EXECUTE($\mathbf{e}_i$); /* perform the event on the GUI */                    6

    /* obtain the expected state for event $\mathbf{e}_i$ */
    $\mathbf{OI}_i \leftarrow$ GETORACLEINFO($i$, $\mathbf{C}_{OI}$);               7

    /* extract the GUI's actual state */
    $\mathbf{AS}_i \leftarrow$ GETACTUALSTATE($i$, $\mathbf{C}_{AS}$);              8

    /* invoke the oracle procedure */
    IF !(OP($\mathbf{AS}_i$, $\mathbf{OI}_i$, $\mathbf{C}_{AS}$, OPF, $i$)) THEN {    9
        RETURN(FAIL)}} /* test case fails */                                        10
  RETURN(PASS)} /* if no failure, report success */                                 11
```

Figure 3. Test Execution Algorithm

was correct for the event.

Having outlined the high-level algorithm for the test case executor, we now develop a formal model of oracle information and procedure and explain the roles of the constraints $\mathbf{C}_{AS}$ and $\mathbf{C}_{OI}$.

3.1 Test Oracle Information

Intuitively, 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 2 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 we have deliberately defined oracle information in very general terms, thus allowing us to create different types of oracles for our study. 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 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. We use a boolean constraint (called $\mathbf{C}_{OI}$ LINE 6 in Figure 3) to define the following four different types of oracle information that we later use in our study. For every triple that is included in the state description, the constraint must evaluate to TRUE.

widget: 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. Note that 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.

active window: 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.

visible windows: the set of all triples for all widgets that are part of the currently *visible* windows of the GUI. The constraint is written as $(inWindow(\#1, x)\ \&\&\ isVisible(x))$, where $isVisible(x)$ is TRUE if window x is visible and FALSE otherwise. Note that visibility is a property of a window, which can be set, for example, by invoking the `SetVisible()` method in Java. Windows that are partially or fully hidden by other overlapping windows are also considered to be visible as long as this property is set.

all windows: the set of all triples for all widgets of all windows. Note that the constraint for this set is simply TRUE since this is the complete state of the GUI.

For brevity, we will use the terms LOI1 to LOI4 for the above four *levels of oracle information*. In Figure 3, we used the subroutine GETORACLEINFO(i, $\mathbf{C}_{OI}$) to compute the oracle information. There are several different ways to compute GETORACLEINFO. We now outline three of them:

1. Using **capture/replay tools** is the most commonly used method to obtain the oracle information [7]. Capture/replay tools are semi-automated tools used to record and store a tester's manual interaction with the GUI with the goal of replaying it with different data and observing the software's output. A detailed discussion of these tools is beyond the scope of this paper. The key idea of using these tools is that 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. We have used **formal specifications** in earlier work [9] to automatically derive oracle information. These specifications are in the form of pre/postconditions for each GUI event.
3. In our experiments presented in this paper, we have used a third approach that we call **execution extrac-**

tion. During this process, a test case is executed on an existing, presumably correct version of the software and its state is extracted and stored as oracle information. We have employed platform-specific technology such as Java API[1], Windows API[2], and MSAA[3] to obtain this information.

3.2 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 match, `FALSE` otherwise. Formally we define an oracle procedure as:

Definition: A **test oracle procedure** is a function $\mathcal{F}$(**OI**, **AS**, $\mathbf{C}_{OI}$, $\mathbf{C}_{AS}$, Φ) $\longrightarrow$ {`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. $\mathcal{F}$ returns `TRUE` if **OI** and **AS** "match" as defined by Φ; `FALSE` otherwise. □

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 4. Note that our specific implementation `OP` of $\mathcal{F}$ takes an extra parameter i that accounts for this frequency. Also note that Φ is hard-coded to "set equality" (Line 7 of Figure 4). `OP` (as invoked from LINE 9 of Figure 3) takes five parameters described earlier. The process of comparing 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 using the constraint $\mathbf{C}_{AS}$ to allow for *set equality* comparison. The oracle procedure returns `TRUE` if the actual state and oracle information sets are equal.

We now use the definition of `OP` to develop five different types of oracle procedures. Note that it is important to specify the constraint $\mathbf{C}_{AS}$ and the set **OPF** to completely specify the oracle procedure.

LOP1: After each event of the test case, we 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** = $\{1, 2, 3, \ldots, n\}$. 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 4).

LOP2: After each event of the test case, we 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** = $\{1, 2, 3, \ldots, n\}$.

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

  IF ($i \in$ OPF) THEN /* compare? */                                 6
    RETURN(FILTER(OI_i, C_AS) == AS_i)                               7
    ELSE RETURN(TRUE)}                                               8
```

Figure 4. Oracle Procedure Algorithm

LOP3: After each event of the test case, we compare the set of all triples for all widgets that are part of the currently *visible* windows of the GUI. The constraint $\mathbf{C}_{AS}$ is written as ($inWindow(\#1, x)$ && $isVisible(x)$) and **OPF** = $\{1, 2, 3, \ldots, n\}$.

LOP4: After each event of the test case, we compare the set of all triples for all widgets of all windows. The constraint $\mathbf{C}_{AS}$ is written as `TRUE` and **OPF** = $\{1, 2, 3, \ldots, n\}$.

LOP5: After the *last* event of the test case, we compare the set of all triples for all widgets of all windows. The constraint $\mathbf{C}_{AS}$ is written as `TRUE` and **OPF** = $\{n\}$.

Even though we define and use only five types of oracle procedures, our definition of `OP` is very general and may be used to develop a large variety of test oracle procedures.

3.3 Oracle Types

Using the four oracle information types (LOI1-LOI4) and five oracle procedures (LOP1-LOP5) described in the previous section, we define different types of test oracles. The key idea of defining these multiple types of oracles is that even though detailed oracle information (say LOI4, i.e., "all windows") may be available (perhaps computed earlier), the tester may choose to save time and compare it against a subset of the actual state, e.g., only "current widget". Hence, LOP1 (comparing against the current widget only) can be used in the presence of all levels of oracle information (LOI1-LOI4). Similarly, LOP2 can be used for LOI2-LOI4. Note that it does not make sense to talk about comparing LOI1 with more actual state information than the active widget; the additional information in the actual state will be simply ignored. The combinations of oracle information and procedure gives us 11 types of test oracles, marked with an × in Table 1.

[1] java.sun.com

[2] msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp

[3] msdn.microsoft.com/library/default.asp?url=/library/en-us/msaa/msaaccrf_87ja.asp

	LOP1	LOP2	LOP3	LOP4	LOP5
LOI1	×				
LOI2	×	×			
LOI3	×	×	×		
LOI4	×	×	×	×	×

Table 1. Types of Test Oracles

4 Experiments

Having presented the design of test oracles and our ability to create many types of oracles, we now present details of an experiment using actual software and test cases to compare the different oracle types. 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.

1. What is the cost (in terms of time and space) incurred in using different oracles?
2. Do different types of test oracles detect different number of faults for a given number of test cases?
3. Are faults detected early in the testing process when using detailed oracle information and complex oracle procedures?

To answer the questions we take different (assumed correct) software, artificially seed faults in them (borrowing this technique from mutation testing), generate test cases and multiple types of test oracles for each test case, and detect the number of faults found by the test cases for each oracle type, while measuring the following variables:

Number of Faults Detected: We record the total number of faults detected by each oracle type, from the pool of faulty programs.

Oracle Comparison Time (OCT): This is the time required to execute the oracle procedure.

Space: Each oracle type has different space requirements, primarily because of the level of detail of the oracle information. We measure the space required to store different levels of oracle information.

4.1 Subject Applications

The subject applications for our experiments are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[4] and consists of six applications out of which we use four – TerpWord, TerpPresent, TerpPaint and TerpSpreadSheet. They have been implemented using Java. Table 2 summarizes the characteristics of these applications. Note that these applications are fairly large with complex GUIs.

	LOC	Classes	Windows
TerpWord	1,747	9	8
TerpPresent	4,769	4	5
TerpPaint	9,287	42	8
TerpSpreadSheet	9,964	25	6
TOTAL	25,767	80	27

Table 2. TerpOffice Applications

4.2 Fault Seeding

Fault seeding is used to introduce known faults into the program under test. Artificially seeded faults should be similar to faults that are naturally introduced into a program due to mistakes made by developers. Offutt et al. have suggested approaches for determining whether a fault is realistic [11]. Harrold et al. [5] have developed fault seeding techniques using program dependence graphs.

We seeded faults in the TerpOffice applications to create 100 faulty versions for each application. Here we discuss the issues faced while seeding faults in GUI applications.

We define a *GUI fault* as one that manifests itself on the visible GUI at some point of time during the software's execution. We adopted an observation-based approach to seed GUI faults, i.e., we observed "real" GUI faults in real applications. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[5] was used by the developers to report and track faults in TerpOffice version 1.0 while they were working to extend its functionality and developing version 2.0. The reported faults are an excellent representative of faults that are introduced by developers during implementation. Table 3(a) shows an example of a fault reported in our Bugzilla database and Table 3(b) shows the (later) corrected segment of the same code. Table 3(c) and 3(d) show examples of faults seeded into this code.

We created 100 faulty versions for each software. Note that exactly 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 test case failure, i.e., a mismatch between actual state and oracle information.

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

[5] bugzilla.org

Reported Fault in Bug Database	Corrected Code

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);

(a)	(b)

Fault #1	Fault #2

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);

(c)	(d)

Table 3. Seeding GUI Faults

4.3 Test cases

We used an automated tool (GUITAR[6]) to automatically generate 600 test cases for each application. Note that GUITAR employs previously developed structures (*event-flow graphs* and *integration trees* [10]) to generate test cases. A detailed discussion of the details of the algorithms used by GUITAR is beyond the scope of this paper. The interested reader is referred to [8] for additional details and analysis.

4.4 Oracle Information

We employed **execution extraction** (Section 3.1) to generate the oracle information. We used an automated tool (also a part of GUITAR) that implements this technique. The key idea to the technique employed by this tool is that it automatically executes a given test case on a software and captures its state (widgets, properties, and values) automatically. 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 four levels of oracle information. We measured the time and memory required for this process.

4.5 Oracle Procedure and Test Executor

We implemented all five levels of oracle procedure. We used "set equality" to compare the actual state with the oracle information.

We executed all 600 test cases on all 100 versions of the subject applications. When each application was being executed, we extracted its run-time state and compared it with the stored oracle information. A mismatch was reported as a fault. Note that we ignored widget positions during this process since the windowing system launches the software in a different screen location each time it is invoked.

Each test case required approximately 10 seconds to execute. This varied depended on the application and the number of GUI events in the test case. The total execution time was approximately 600,000 seconds 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 11 oracle types.

4.6 Threats to Validity

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

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher's knowledge. We have used an observation-based approach for seeding faults in the GUI applications. This may have affected the detection of faults by the test cases. Faults not exercised by any test case will go undetected. We made an effort to make the faults as close as possible to naturally occurring faults. Some of these faults might not manifest themselves through the GUI.

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

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

4.7 Results

We now present some of the results of our experiments. Due to lack of space, in many of these results, we are unable to present data for all 11 oracle types. Instead, whenever possible, we combine and report results of five important data points: L1 (LOI1, LOP1), L2 (LOI2, LOP2), L3 (LOI3, LOP3), L4 (LOI4, LOP4), and L5 (LOI4, LOP5) (see Table 1).

[6] guitar.cs.umd.edu

Fault-detection ability: *Result: complex oracles are better at detecting faults than the simplest ones.* Figure 5 shows the percentage of faults detected by the test cases for different levels of oracles. The height of the columns shows this percentage. The graph shows that there is an improvement in fault detection from L1 to L3 for almost all applications. The cheapest oracle (L1) detects a very small percentage of faults. However, there is no significant improvement from L3 to L4 and L5. The most expensive oracle does not provide significant improvement in detection over an intermediate oracle.

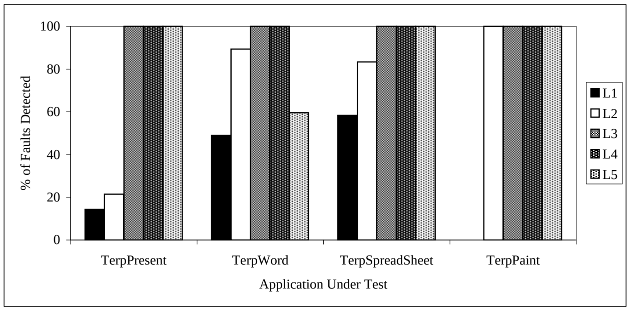


Figure 5. Number of Faults Detected

Test case length: *Result: a greater percentage of shorter test cases with complex oracle procedures are able to detect faults than longer test cases with less complex oracle procedures.* In Figure 6, the x-axis represents the test case length and y-axis shows the number of test cases that successfully detected at least one fault (averaged for all four subject programs). Note that there is a significant increase in the fault detection ability of test cases when equipped with more powerful test oracle procedures. Also note that we did not present results for LOP5 since the comparison is done only after the last event of the test case, in which case the length of the test case is irrelevant.

Number of test cases: *Result: a greater percentage of test cases detect faults when using an expensive oracle.* We show this result using 3-D graphs (Figures 7 and 8). The x-axis shows the number of detected faults, the y-axis shows the number of test cases (averaged over the four subject programs) that detected the faults successfully, and the z-axis shows the different oracles. From the graph, it is seen that only 1100 test cases are able to detect even a single fault for L1, whereas almost 2000 test cases are able to detect faults for L2, L3, and L4. Note, however, that there is no significant difference between L3 and L4. Also, note that for L5 (see Figure 8 for better view), the number of test cases that detect at least one fault is quite large.

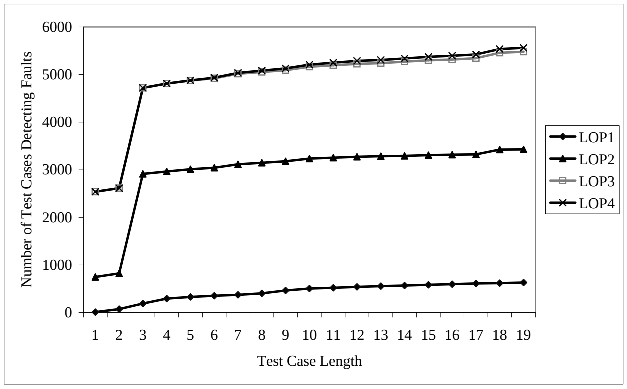


Figure 6. Number of Test Cases of a Specific Length and their Fault Detecting Ability

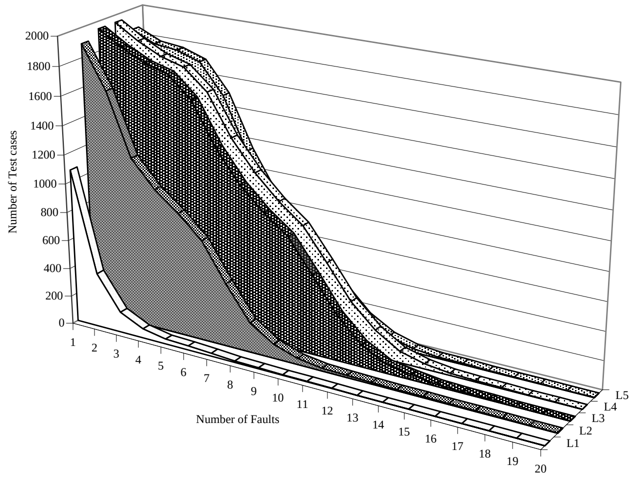


Figure 7. Number of Test Cases Detecting Faults (view 1)

Time: *Result: A complex oracle procedure is expensive in terms of computation time.* The actual time for executing the oracle procedure (OCT) is shown in Figure 9. A more expensive oracle procedure takes longer time to execute. This was true for all the four applications. This is because a higher LOP validates a more detailed GUI state than a cheaper one. An exception is LOP5, which executes faster. This is because LOP5 validates only the final state of the GUI.

Space requirements: *Result: L3 and L4 show a significant increase in storage requirements.* In Figure 10 we compare the storage requirements of different oracles. There is a large increase of storage requirement from L2 to L3. This is because L3 and L4 capture a more complete GUI state information that L1 and L2. These relatively expensive oracle can be used to detect faults by executing shorter test cases.

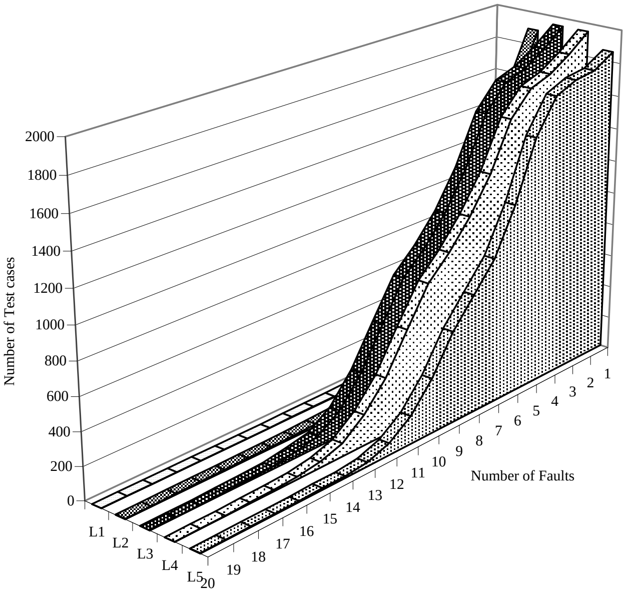


Figure 8. Number of Test Cases Detecting Faults (view 2)

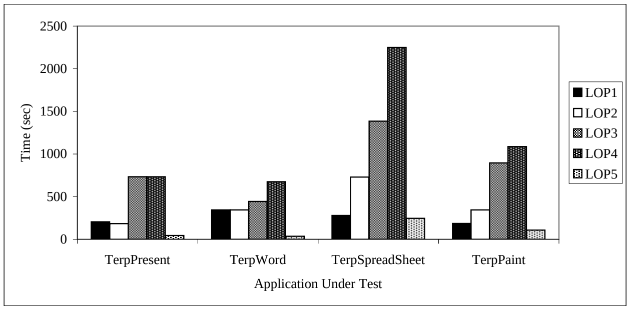


Figure 9. Total time for executing oracle procedure

The above results show that oracle type L5, i.e., checking for the complete GUI's state after the last event of the test case has been executed, is both cheap in terms of space and time, and yet is able to yield results that are comparable to the most expensive test oracles. With the exception of L5, all complex oracles were more expensive both in terms of time and space, but they were also more successful in detecting faults.

5 Related Work

Very few techniques have been developed to automatically generate the expected output for conventional software. Hence, software systems rarely have an automated

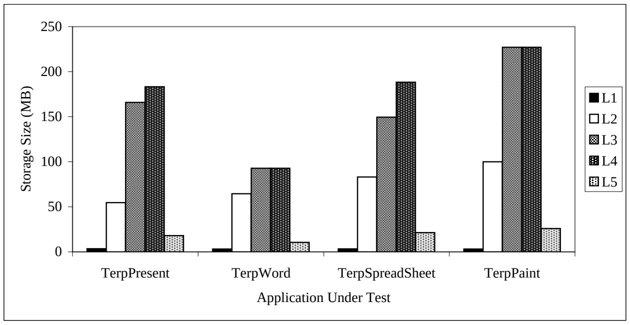


Figure 10. Oracle storage requirements

test oracle [12, 14, 13, 3]. In most cases, the expected behavior of the software is assumed to be provided by the test designer. The expected behavior is specified by the test designer in the form of a table of pairs *(actual output, expected output)* [12], as temporal constraints that specify conditions that must not be violated during software execution [13, 2, 3, 14], or as logical expressions to be satisfied by the software [4]. This expected behavior is then used by the verifier by either performing a table lookup [12], FSM creation [6, 3], or boolean formula evaluation [4] to determine the correctness of the actual output.

Richardson in TAOS (Testing with Analysis and Oracle Support) [13] 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 [15] 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. A popular alternative to manually specifying the expected output is by performing reference testing [16, 17]. Actual outputs are recorded the first time the software is executed. The recorded outputs are later used as expected output for regression testing.

Automated GUI test oracles were developed in the PATHS (Planning Assisted Tester for grapHical user interface Systems) system [9, 8]. PATHS uses AI planning techniques to automate testing for GUIs. The oracle described in PATHS uses a formal model of a GUI to automatically derive the oracle information for a given test case.

6 Conclusions

In this paper, we 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. By varying the level of detail of oracle information and changing the oracle procedure, we developed 11 types of test oracles. We empirically showed that faults are detected early in the testing process when using detailed oracle information and complex oracle procedures, although at a higher cost per test case. Moreover, employing expensive oracles catches a large number of faults using relatively smaller number of test cases.

Our results provide valuable guidelines to testers. If testers have short and a small number of test cases, they can improve their testing process by using complex test oracles. On the other hand, if they have generated test cases using an automated tool (e.g., GUITAR), then they can use cheaper and simpler test oracles to conserve resources.

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 check for the correctness of (partial) states of the objects.

In the future, we will extend our pool of subject applications to include non-Java and non-GUI programs. We will also generate multiple types of test cases and observe the effect of different test oracles on these test cases. Finally, since we have identified differences in fault-detection ability of different test oracles, we will develop adequacy criteria for test oracles in a way similar to those already available for test cases [18].

References

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

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

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

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

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

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

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

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

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

[10] A. M. Memon, M. L. Soffa, and M. E. Pollack. 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)*, pages 256–267, Sept. 2001.

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

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

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

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

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

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

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

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