Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software

Atif M. Memon, *Member*, *IEEE Computer Society*, and Qing Xie, *Student Member*, *IEEE*

Abstract—Software is increasingly being developed/maintained by multiple, often geographically distributed developers working concurrently. Consequently, rapid-feedback-based quality assurance mechanisms such as daily builds and smoke regression tests, which help to detect and eliminate defects early during software development and maintenance, have become important. This paper addresses a major weakness of current smoke regression testing techniques, i.e., their inability to automatically (re)test graphical user interfaces (GUIs). Several contributions are made to the area of GUI smoke testing. First, the requirements for GUI smoke testing are identified and a GUI smoke test is formally defined as a specialized sequence of events. Second, a GUI smoke regression testing process called *Daily Automated Regression Tester* (DART) that automates GUI smoke testing is presented. Third, the interplay between several characteristics of GUI smoke test suites including their size, fault detection ability, and test oracles is empirically studied. The results show that: 1) the entire smoke testing process is feasible in terms of execution time, storage space, and manual effort, 2) smoke tests cannot cover certain parts of the application code, 3) having comprehensive test oracles may make up for not having long smoke test cases, and 4) using certain oracles can make up for not having large smoke test suites.

Index Terms—Smoke testing, GUI testing, test oracles, empirical studies, regression testing.

1 Introduction

Many of today's software applications are developed and maintained by multiple programmers, often geographically distributed, who work on parts of the overall application code. While leading to improved code churn rates, this practice also leads to problems. For example, developers may not realize that they have inadvertently broken parts of the code. Consequently, rapid-feedback-based quality assurance mechanisms are integrated into the development and maintenance cycle. One such mechanism requires nightly/daily building of the software and execution of smoke regression tests. In recently reported work [1], [2], an important weakness of current smoke testing techniques, i.e., their inability to automatically and efficiently smoke test the graphical user interface (GUI) front-end part of the software, was addressed. The concept of GUI smoke tests was introduced and a framework called *Daily Automated Regression Tester* (DART) that retests frequent builds of GUI software was described.

DART has been implemented and successfully deployed. The key to the success of DART is that developers can work on the code during the day; DART automatically launches the GUI *application under test* (AUT) at night, builds it and runs GUI smoke test cases. Coverage and bug report summaries are e-mailed to developers. DART automates everything required for GUI smoke testing including structural GUI analysis (called *GUI ripping* [3]), test case generation [4], test oracle creation [5], code instrumentation, test execution, coverage evaluation, regeneration of test cases and their reexecution, and automated reporting via a Web-based virtual scoreboard called DART-Board. Together with the operating system's task scheduler (e.g., MS Windows task scheduler, Unix cron job), DART can execute frequently with little input from the developer/tester to smoke test the GUI software. DART and its components were described in a preliminary report of this research [1]. The fault-detection effectiveness of GUI smoke tests was also demonstrated.

This paper builds upon the previous research by studying the following important, previously ignored, aspects of automated GUI smoke testing:

- *Size of a smoke test suite*: Smoke test suites of various sizes are created and the impact of size on fault-detection effectiveness is studied.
- *Complexity of test oracles*: Five oracles of varying complexity are developed for GUI smoke tests and their effectiveness is studied.
- *Characteristics of faults* that can and cannot be detected by the GUI smoke test cases are studied.

Most importantly, the interplay between the above factors is empirically studied. For example, the combined effect of using different-sized test suites with various test oracles on faults detected is studied. Finally, guidelines are presented for practitioners who perform smoke testing of GUI-based software to help them generate and execute effective smoke test suites, taking into consideration their fault-detection ability and size so that they can be executed in one night.

The specific contributions of this work include:

- identification of requirements for GUI smoke tests,
- formal definition of a GUI smoke test case,

• *The authors are with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: {atif, qing}@cs.umd.edu.*

Manuscript received 15 Mar. 2005; revised 22 June 2005; accepted 28 June 2005; published online 3 Nov. 2005.
Recommended for acceptance by Harman, Korel, and Linos.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0066-0305.

- study of code that cannot be covered by the smoke tests, and
- a first empirical study evaluating the strengths and weaknesses of GUI smoke tests and tradeoffs between test oracles, test suite size, and faults detected.

Structure of the paper. Background and related work is presented in the next section. In Section 3, a brief overview of the DART process is given. A GUI smoke test case is formally defined in Section 4. In Section 5, test oracles for smoke test cases are discussed. Empirical studies are described in Section 6. Finally, a discussion of limitations, and ongoing and future work is presented in Section 7.

2 Background and Related Work

Nightly/daily builds and smoke tests [6], [7], [8] have become widespread [9], [10]. During nightly builds, a development version of the software is checked out from the source code repository tree, compiled, linked, and "smoke tested" ("smoke tests" are also called "sniff tests" or "build verification suites" [11]). Typically, *unit tests* [9] and sometimes *acceptance tests* [12] are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system [11]. Smoke tests exercise the entire system; they do not have to be an exhaustive test suite but they should be capable of raising a "something is wrong here" alarm. A build that passes the smoke test is considered to be "a good build." As is the case with all testing techniques, it is quite possible that problems are found in a good build during more comprehensive testing later or after the software has been fielded.

Daily building and smoke testing have been used for a number of large-scale commercial and open-source projects. For example, Microsoft used daily builds extensively for the development of its Windows NT operating system [7]. By the time it was released, Windows NT 3.0 consisted of 5.6 million lines of code spread across 40,000 source files. The NT team attributed much of the project's success to their daily build process. The GNU project continues to use daily builds for most of its projects. For example, during the development of the Ghostscript software, daily builds were used widely. The steps for daily builds involved compiling the source, executing smoke tests, and updating the CVS repository. Similarly, *WINE* [13], *Mozilla* [14], *AceDB* [14], and *openwebmail* [15] use nightly/daily builds.

Although there is no prior work that directly addresses the research presented in this paper, except our own work [1], [2], several researchers and practitioners have discussed the following concepts that are relevant to its specific parts:

Daily Build Tool Support. There are several tools that may be used to setup and perform smoke testing of software applications. Popular examples include *CruiseControl* [16], *IncrediBuild* [17], *Daily Build* [18], and *Visual Build* [19]. Most of these tools provide more or less identical functionality. CruiseControl, Daily Build, and Visual Build are frameworks for setting up and running continuous build processes. They include plug-ins for tools, e.g., for e-mail notification and source control. Web interfaces provide views to details of the current and previous builds. IncrediBuild speeds up daily building and smoke testing by performing distributed compilation of source; it distributes the compilation task across several available machines in an organizational network. It has been found to be effective for nightly/daily builds of Miscrosoft Visual C++ (6.0, 7.0, and 7.1) applications. While there are tools that support daily building and smoke testing of conventional software, there is no literature on techniques and tools for daily building and smoke testing of GUI software.

Web Testing. Web user interfaces share several characteristics with GUIs. Event sequences performed by users are captured by a browser and sent to a Web server. There have been a number of efforts aimed at automating Web testing. The most popular is to simulate sequences of HTTP requests by using one of several *HTTP torture machines* [20]. The response to each request is then analyzed and its correctness in the context of the single request determined. The disadvantage of this approach is that the tester lacks a global perspective of a typical users' interactions and the collective effect of a sequence of events as seen in a browser's window. Because of this limitation, this testing is restricted to: 1) *load testing* [20], [21] of the servers to determine the number of requests they can handle simultaneously, 2) performing restricted forms of checking such as the validity of links [22], or 3) the client's compliance to HTML requests [23].

GUI Testing Tools. Three different approaches are used to handle GUI software when performing testing. First, and most popular, is to perform no GUI testing at all [11], which leads to compromised software quality. Second is to use test harnesses that "bypass" the GUI and invoke methods of the underlying business logic as if initiated by a GUI. This approach not only requires major changes to the software architecture (e.g., keep the GUI software "light" and code all "important" decisions in the business logic [24]), it also does not perform testing of the end-user software. Third is to use manual GUI testing tools to do limited testing [25], [26]. Examples of some tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module* [27] and capture/replay tools [28] that provide very little automation, especially for *creating* smoke tests.

Capture/replay tools (also called record/playback tools) operate in two modes: *Record* and *Playback*. In the *Record* mode, tools such as *CAPBAK* and *TestWorks* [29] record mouse coordinates of the user actions as test cases. In the *Playback* mode, the recorded test cases are replayed automatically. The problem with such tools is that, since they store mouse coordinates, test cases break even with the slightest changes to the GUI layout. Tools such as *Winrunner* [30], *Abbot* [31], and *Rational Robot* [32] avoid this problem by capturing GUI widgets rather than mouse coordinates. Although playback is automated, significant effort is involved in creating the test scripts, detecting failures, and editing the tests to make them work on the modified version of software. Developers/testers who employ these tools typically come up with a small number of smoke tests [33]. Moreover, the tools do not support smoke testing of GUI software.

The most comprehensive and complete solution is provided by DART [1], [2] that addresses the needs of smoke testing of software applications that have a GUI. DART automates smoke testing of GUIs by using model-based testing techniques. An overview of DART is presented next.

3 The DART Process

The main design goal of DART is to automate GUI smoke testing. A test designer uses a process called the "DART process" to realize this automation. This process is outlined next; the goal is to provide the reader with a high-level picture of the operation of DART. Note that the names of the modules of DART are in boldface font and described later in Section 6.

1. The developer (or test designer) identifies the AUT. This essentially means that the developer identifies the locations of source files and any library modules needed to compile/build the AUT. DART maintains an internal identifier for these AUT artifacts. This version of the AUT used for DART setup is called the "baseline AUT."
2. DART analyzes the (baseline) AUT's GUI structure (using a module called the **GUI ripper**) by automatically traversing all the windows of the GUI and extracting all the GUI objects (widgets) and their properties. The internal AUT representation is a set of triples, each of the form (*widget*, *property*, *value*) and *event-interaction* graphs (discussed in Section 4).
3. DART computes the total number of possible smoke test cases (event sequences) that *may be executed* on the AUT. An example of the total number of smoke test cases that may be executed on the subject applications used in the empirical studies is shown in Table 2. The test designer then specifies the *number* of test cases that *should be executed*. The numbers specified by the test designer in the empirical study are shown in Table 3. Note that this type of specification does not give the test designer low-level control of exactly what test cases to generate and execute; however, it is very quick and effective. By default, all length 1 and length 2 test cases (where length is the number of events) are executed.
4. DART's automated **test case generator** generates the smoke test cases (Section 4).
5. A **test oracle generator** automatically creates, for each test case, an expected output that is used to verify the correctness of the next version of the AUT (Section 5). The *smoke test suite* for subsequent versions is now ready.
6. The development team uses change requests and bug reports to modify the AUT.
7. The operating system's task scheduler launches DART, which in turn launches the AUT. DART automatically instruments the AUT's source code using a **code instrumenter** (e.g., JCover [34]).
8. Test cases are executed (using a **test case executor**) on the instrumented modified AUT automatically and the output is compared to the stored expected output (from Step 5). An execution report is generated in which the executed test cases are classified as *successful* or *unsuccessful*. A coverage report is also generated, which indicates what statements and branches have been executed. These results are e-mailed to the developers. A Web-based virtual scoreboard called the DART-Board is automatically created. The DART-Board provides a summary of the results with links to test cases and detailed test results.
9. The next morning, the developers examine the reports. They also examine the unsuccessful test cases. Note that a test case may be unsuccessful because 1) it crashed the software, 2) the expected output did not match the actual output, or 3) an event in the test case had been modified (e.g., deleted) preventing the test case from executing.
10. The developers submit the bug reports.

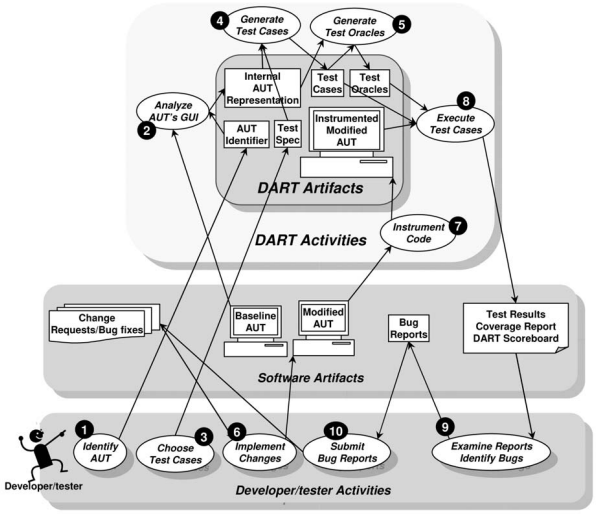

Fig. 1. The DART process.

The smoke testing process restarts as Steps 3 through 10 are repeated throughout the development cycle of the AUT. If needed, the developer may also include code-based smoke test cases in the above cycle.

The above (numbered) steps are also shown in Fig. 1. The figure contains four main parts: DART artifacts, DART activities, software artifacts, and developer/tester activities. As the names suggest, the DART activities are done automatically whereas the developer/tester activities are done manually. DART encapsulates all the DART artifacts. The numbered (the numbers correspond to the above steps) ovals represent activities and the boxes represent artifacts. Directed edges to and from activities represent the consumption and production of artifacts, respectively.

Subsequent sections provide details of smoke test cases and test oracles.

4 GUI Smoke Test Cases

Users interact with a GUI by performing *events* on some widgets, such as clicking on a button, opening a menu, and

dragging an icon. During GUI testing, test cases, consisting of sequences of events, are executed on the GUI.[1] For GUI smoke testing, a tester has to produce test cases that satisfy the following requirements:

- The smoke test cases should be generated and executed quickly, i.e., in one night.
- The test cases should provide adequate coverage of the GUI's functionality. As is the case with smoke test cases of conventional software, the goal is to raise a "something is wrong here" alarm by checking that GUI events and event-interactions execute correctly.
- As the GUI is modified, many of the test cases should remain usable. Earlier work showed that GUI test cases are very sensitive to GUI changes [33]. The goal here is to design test cases that are robust, in that a majority of them remain unaffected by changes to the GUI.
- The smoke test suite should be divisible into parts that can be run (in parallel) on different machines.

These requirements naturally follow from those of smoke tests for conventional software, i.e., the need for speed [12], breadth of coverage, and low overhead and maintenance. This research uses a specialized GUI model to automatically generate smoke test cases that satisfy the above requirements. This model is called the *event-interaction graph*, which is based on a structure called the event-flow graph (EFG) [33]. These models are described next.

4.1 Event-Flow Graphs

Since GUIs may be used as a front-end to many different types of software applications, the space of all possible GUIs is enormous. It would be extremely difficult to create one model for all possible types of GUIs. Hence, to provide focus, this research models a subclass of GUIs. Specifically, the GUIs in this subclass react to events performed only by a single user and the events are deterministic, i.e., their outcomes are completely predictable. Testing GUIs that react to temporal and nondeterministic events and those generated by other applications is beyond the scope of this work.

Intuitively, an EFG models all possible event sequences that may be executed on a GUI. An EFG contains nodes (that represent events) and edges. An edge from node n_x to n_y means that the event represented by n_y may be performed *immediately after* the event represented by node n_x. An example of an EFG for the `Main` and `Replace` windows of the MS NotePad software is shown in Fig. 2. Events (corresponding to each widget) are shown as labeled boxes. The labels show a meaningful unique identifier for each event. Directed edges show the event-flow relationship between events. For increased readability, only some of the edges are shown. Sets of events are defined and listed in a Legend. For example `TopLevel` is a set containing the events `File`, `Edit`, `Format`, `View`, and `Help`. Similarly, ① is a set containing all the events in `TopLevel` and `ReplaceSet`. An edge from `Copy` to ① represents a number of edges, from `Copy` to each event in ①. According to this EFG, the event `Cancel` can be executed immediately after the event `Find Next`; event `Match case` can be executed after itself; however, event `Replace` cannot be executed immediately after event `Cancel`.

1. We have shown in earlier work that simply executing each event in isolation is not enough for effective GUI testing [4].

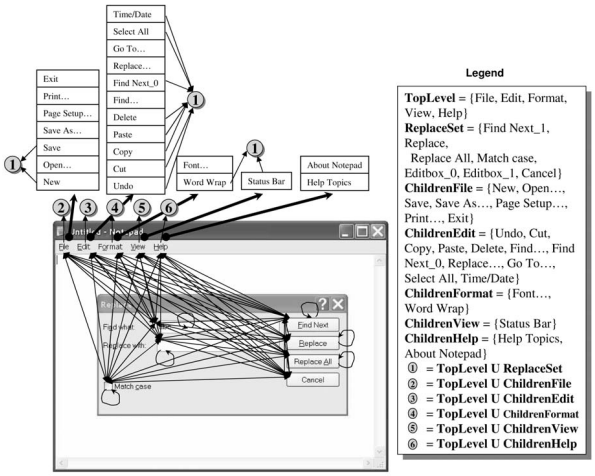

Fig. 2. Example of an event-flow graph.

4.2 Event-Interaction Graphs

Since an EFG models *all possible* (an extremely large number of) event interactions, it cannot be used directly for rapid testing; abstractions are used to model only specialized (and, hence, a smaller number of) event sequences. In a typical GUI, 20-25 percent of the events are used to manipulate the structure of the GUI; examples include events that open/close windows/menus. For example, in Microsoft Word, of the total 4,210 events, 80 events open menus, 346 events open windows, and 196 events close windows; the remaining 3,588 events interact with the underlying code. The code for events that open menus and windows is straightforward, usually generated automatically by visual GUI-building tools. This code is very unlikely to interact with code for other events; hence, very few errors are revealed by executing interactions between these events. The remaining events in the GUI are *nonstructural events* that do not cause structural changes to the GUI; rather, they are used to perform some action; common examples include the `Copy` event used for copying objects to the clipboard.

Events that interact with the underlying software include nonstructural events and those that close windows. These events are called system-interaction events. Intuitively, the GUI smoke test cases are composed only of these events (and any other events necessary to "reach" the system-interaction events). Once these events have been identified in a GUI, the EFGs are transformed to event-interaction graphs, which are used to generate smoke tests. We now define some terms that help in this transformation.

Intuitively, an event-flow path represents a sequence of events that can be executed on the GUI. Formally, an event-flow-path is defined as follows:

Definition. *There is an* event-flow-path *from node n_x to node n_y iff there exists a (possibly empty) sequence of nodes*

$n_j; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ *in the event-flow graph* E *such that* $\{(n_x, n_j), (n_{j+k}, n_y)\} \subseteq edges(E)$ *and*

$$\{(n_{j+i}, n_{j+i+1}) for\ 0 \leq i \leq (k-1)\} \subseteq edges(E).$$

In the above definition, the function $edges$ takes an event-flow graph as an input and returns a set of ordered-pairs, each representing an edge in the event-flow graph. The notation $< n_1; n_2; \ldots; n_k >$ is used for an event-flow path. Several examples of event-flow paths from the EFG of Fig. 2 are:

$$< File; Edit; Undo >, < File; MatchCase; Cancel >,$$
$$< MatchCase; Editbox\_1; Replace >, \text{ and}$$
$$< MatchCase; FindNext; Replace > .$$

Smoke test cases consist of those event-flow-paths that start and end with system-interaction events, without any intermediate system-interaction events.

Definition. *An event-flow-path* $< n_1; n_2; \ldots; n_k >$ *is* interaction-free *iff none of* $n_2, \ldots, n_{k-1}$ *represent system-interaction events.*

Of the examples of event-flow paths presented above, $< File; Edit; Undo >$ are interaction-free (since $Edit$ is not a system-interaction event) whereas

$$< MatchCase; Editbox\_1; Replace >$$

is not (since $Editbox\_1$ is a system-interaction event).

Intuitively, two system-interaction events may interact if a GUI user may execute them in an event sequence without executing any other intermediate system-interaction event.

Definition. *A system-interaction event* e_x interacts-with *system-interaction event* e_y *iff there is at least one interaction-free event-flow-path from the node* n_x *(that represents* e_x*) to the node* n_y *(that represents* e_y*).*

For the EFG of Fig. 2, the above relation holds for the following pairs of system-interaction events:

$$\{(New, Date/Time), (FindNext\_1, WordWrap),$$
$$(Editbox\_0, Editbox\_1), \text{ and } (Delete, Cancel)\}.$$

The interaction-free event-flow-paths for these pairs are

$$< New; Edit; Date/Time >,$$
$$< FindNext\_1; Format; WordWrap >,$$
$$< Editbox\_0; Edibox\_1 >, \text{ and } < Delete; Cancel >,$$

respectively. Note that an event may interact-with itself. For example, the event `MatchCase` interacts with itself. Also note that "e_x interacts-with e_y" does not necessarily imply that "e_y interacts-with e_x." In the EFG example, even though $Replace$ interacts-with $Cancel$, the event $Cancel$ does not interact-with $Replace$.

The interacts-with relationship is used to create the event-interaction graph. This graph contains nodes, one for each system-interaction event in the GUI. An edge from node n_x (that represents e_x) to node n_y (that represents e_y) means that e_x interacts-with e_y. The algorithm to convert an EFG to an event-interaction graph is shown in Fig. 3. The

```
1  N /* Nodes set of EIG */
2  E /* Edges set of EIG */
3  PROCEDURE :: GenerateEIG(
4    Event Flow Graph (N, E)) {
5   N = N
6   E = E
7
8   FORALL n ∈ N DO
9    start(n) = { n_i | (n, n_i) ∈ E, and n ≠ n_i }
10   end(n) = { n_i | (n_i, n) ∈ E, and n ≠ n_i }
11
12  FORALL n ∈ N DO
13   IF EventType(n) ≠ system-interaction
14    FORALL n_x ∈ end(n) DO
15     FORALL n_y ∈ start(n) DO
16      E = E ∪ (n_x, n_y)
17      IF n_x ≠ n_y
18       start(n_x) = start(n_x) ∪ {n_y}
19       end(n_y) = end(n_y) ∪ {n_x}
20    FORALL n_x ∈ end(n) DO
21     remove n from start(n_x)
22    FORALL n_y ∈ start(n) DO
23     remove n from end(n_y)
24    remove n from N
25    remove all edges (n, n_i) from E
26    remove all edges (n_i, n) from E
27 }
```

Fig. 3. Generate event-inteaction graph from event-flow graph.

procedure `GenerateEIG` takes as input an EFG, represented as a set of nodes N and a set of edges E. It removes all nonsystem-interaction event nodes and their associated edges from the given EFG. At the termination of the procedure, the event-interaction graph is obtained, represented as a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$. $\mathcal{N}$ and $\mathcal{E}$ are initialized to N and E (lines 2-3). When traversing all edges of the EFG, a list of nodes $\text{start}(n)$ on the edges that start from the node n (except itself) is obtained for all nodes. Similarly, a list of nodes $\text{end}(n)$ that end with the node n (except itself) for all nodes (lines 4-6) is computed. For each node n of the EFG (line 7), all new edges (n_x, n_y) are added to $\mathcal{E}$ if there is an interaction-free path $< n_x; n; n_y >$ in the EFG (lines 8-11); $\text{start}(n_x)$ and $\text{end}(n_y)$ are updated to add n_y and n_x in the lists, respectively, if n_x and n_y are not the same node (lines 12-14). Accordingly, n is removed from the start and end lists (lines 15-18). Finally, n is removed from $\mathcal{N}$ (line 19); all edges associated with n are removed from $\mathcal{E}$ (lines 20-21). The event-interaction graph for the EFG of Fig. 2 is shown in Fig. 4. Note that the space of event-sequences has reduced considerably since only the system-interaction event interactions are modeled in this graph.

An event-interaction graph may be traversed in a number of ways to generate sequences of system-interaction events, which form the GUI smoke test cases. For example, all length 1 event sequences may be generated by simply enumerating all the nodes in the graph. All length 2 event sequences may be generated by enumerating each node with its adjacent node. The number of lengths 1 and 2 system-interaction event sequences represented by the event-interaction graph may be larger than the number of event sequences of lengths 1 and 2, respectively, represented

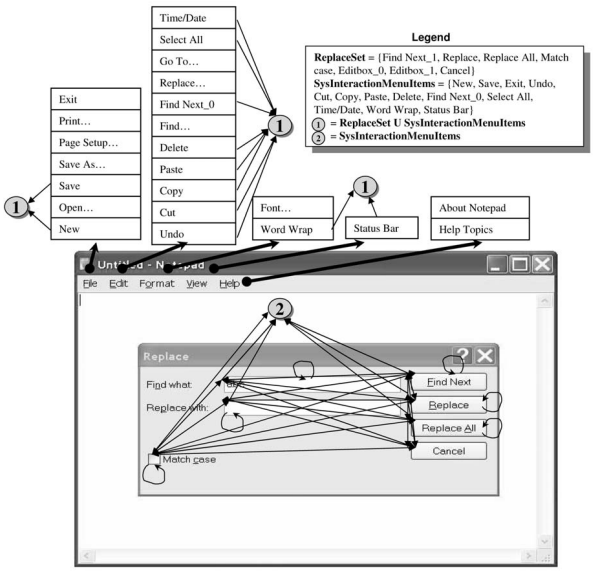

Fig. 4. The event-interaction graph for the event-flow graph of Fig. 2.

by the corresponding event-flow graph. This is not surprising since the system-interaction event sequences represented by the event-interaction graph are a compact form of longer, and, hence, a larger number of, event sequences represented by the event-flow graph.

The remaining question is how to execute the generated system-interaction event sequences. At execution time, other events needed to "reach" the system-interaction events are generated on-the-fly. A simple graph traversal algorithm is used on the EFGs to obtain these events. For example, the system-interaction sequence $< New; Undo >$ will expand to $< File; New; Edit; Undo >$ during test-case execution.

The smoke test cases exhibit the following properties:

1. The test cases are short; they can be generated and executed very quickly.
2. The smoke test case only consists of system-interaction events; changes to the GUI layout, such as moving events from one window to another and changing the menu structure, leave most of the test cases unaffected. Other events are generated on-the-fly *during* test execution.
3. The expected state is stored only for system-interaction events; it will become obsolete only if a system-interaction event is modified.
4. All system-interaction events are executed; most of the GUI's functionality is covered.
5. Each test case is independent and the suite can be distributed.

The most important property of smoke tests is that they can be generated and executed automatically using DART.

5 GUI Test Oracle

Since the goal of smoke testing is to ensure that the software has not "broken" during modifications, automated GUI test oracles check that the "software does what it was doing before modifications" were made. An automated test oracle contains three parts. First, the *oracle information generator* automatically derives the *oracle information* (expected state) using either a formal specification of the GUI as described in earlier work [5] or by using a "baseline" version of the software [35], [36] (as described later in this section). Second, the *actual state* is obtained from an *execution monitor*, which may use any of the techniques described in [5], such as screen scraping, querying, and reverse engineering [3]. Finally, an *oracle procedure* automatically compares the two states and reports GUI errors.

For any smoke test case $e_1; e_2; \ldots e_n$, where each e_i is a system-interaction event, the oracle information is represented as a sequence of states $S_1; S_2; \ldots S_n$ that capture the complete (or partial) state of the GUI after each event. The GUI state is represented as a set of *widgets* $W = \{w_1, w_2, \ldots, w_l\}$ (e.g., buttons, panels, text fields) that constitute the GUI, a set of *properties* $P = \{p_1, p_2, \ldots, p_m\}$ (e.g., background color, size, font) of these widgets, and a set of *values* $V = \{v_1, v_2, \ldots, v_n\}$ (e.g., red, bold, 16pt) associated with the properties. More specifically, the *state* of a GUI at a particular time is the set S of triples $\{(w_i, p_j, v_k)\}$, where $w_i \in W$, $p_j \in P$, and $v_k \in V$. The tools in DART are able to capture the state of a GUI at any instant during its execution. Platform-specific technology such as Java API is used to capture this information. Details of the technique are beyond the scope of this work; the interested reader is referred to previously published work [3]. For the purpose of this work, it is sufficient to understand that a snapshot of the GUI's state can be captured fully automatically.

During smoke testing, the modified GUI version is compared with the original version and changes are reported. The steps involved are: 1) execute the test case on the original GUI and collect state information, 2) execute the same test cases on the modified version, and 3) compare the GUI's state with the stored information and report any changes. There are several ways to use the above technique in a test oracle. For example, the complete GUI state (i.e., the state of all windows and widgets) may be collected, stored, and compared after each event in the test case, or the complete state may be collected, stored, and compared only after the last event.

Depending on the resources available, DART can collect and compare oracle information at different levels of detail:[2]

- *Widget*: all the properties of the object on which the current event is being performed. This test oracle, although simple, may be used to check the correctness of widgets whose state changes when events are performed on them, e.g., text-boxes, radio-buttons, check-boxes, etc.
- *Active window*: all the properties of all the objects of the *active* window in the GUI.
- *Complete visible*: all the properties of all the objects of all the *visible* windows in the GUI. Note that the term "visible windows" represents all windows for which the `isVisible` property is TRUE. Windows that are "hidden" behind other overlapping windows are

2. The need for these levels is explained in detail in earlier reported work [5].

also considered visible, if their `isVisible` property is TRUE.
- *Complete*: all the properties of all the objects of all the windows in the GUI.

In all the above definitions, the properties and GUI objects include only those that can, in fact, be obtained using DART's reverse engineering tools (described in Section 6).

The comparison of the expected and actual states can be done as frequently as once after each event of the test case or less frequently, e.g., after the last event. It is expected that reducing the frequency of comparison will reduce the test case execution time as well as space required to store the oracle information (since only the last state of the GUI needs to be stored). Note that, in the empirical studies presented in this paper, where all test cases are started in the same GUI state, efficient storage management may be used to conserve space. For example, since the first event in the length 2 event sequence has its own length 1 test case, it is possible to use a pointer to where the state is stored, rather than storing duplicate copies. However, in general, if the test designer starts each test case in a new state, this space-saving strategy cannot be used since intermediate states will be different.

Using the combination of oracle information and frequency of comparison, five oracles of "different levels of comprehensiveness" are obtained for GUI smoke testing. Oracle levels L1, L2, L3, and L4 represent comparing the *widget*, *active window*, *complete visible*, and *complete* oracle information *after each event* of the test case with the actual GUI, respectively. Oracle level L5 represents comparing the complete oracle information *after the last event* of the test case with the actual GUI.

6 Empirical Studies

DART has been implemented in Java. It contains several modules:

1. The *GUI ripper* is the automated module that creates event-interaction graphs. "GUI Ripping" is a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets (GUI objects), properties, and values. An important characteristic of the GUI ripper that has an impact on DART is its ability to extract different types of widgets. By default, the GUI ripper extracts only "known" widget types, typically those provided by the implementation platform's library. Custom widgets (those developed by the programmer, e.g., cells in a spreadsheet) need additional specification—the developer needs to specify the types of events that may be executed on the custom widgets and methods to extract their state. As will be seen later, the capabilities of the GUI ripper have an impact on the fault-detection effectiveness of DART.
2. The *test-case generator* uses the event-interaction graphs to create the smoke tests.
3. The *test-oracle* generator automatically executes the generated test cases on the latest GUI version and stores the captured state.

TABLE 1
TerpOffice Applications

Subject Application	Windows	Widgets	LOC	Classes	Methods	Branches
TerpWord	11	112	4893	104	236	452
TerpSpreadSheet	9	145	12791	125	579	1521
TerpPaint	10	200	18376	219	644	1277
TerpCalc	1	82	9916	141	446	1306
TerpPresent	12	294	44591	230	1644	3099
TOTAL	43	833	90567	819	3549	7655

4. The *test executor* executes an entire test suite automatically on the AUT. It performs all the events in each test case and invokes the test oracle to compare the actual output with the expected output. Events are triggered on the AUT using the native OS API. For example, the windows API *SendMessage* is used for windows applications and Java API *doClick* for Java applications.

Details of two empirical studies using actual software subjects to demonstrate important characteristics of GUI smoke test suites are now presented. First, the faults found by the smoke test cases and test oracles are studied. Then, smoke test suites of different sizes are developed and studied. In particular, the following questions are addressed:

1 What is the fault detection ability of the GUI tests?
2 What is the impact of using different test oracles?
3 What is the impact of test suite size on test results?
4 What is the relationship between faults, test suite size, and test oracles?
5 Are GUI smoke tests especially suited to detect certain classes of faults? What are the characteristics of faults that cannot be detected by the smoke test cases?
6 Are there parts of the code that cannot be executed by the smoke test cases? What are the characteristics of this code?

6.1 Subject Applications

The subject applications for the studies are part of an open-source office suite developed in the Department of Computer Science at the University of Maryland by undergraduate students in the senior Software Engineering course. It is called TerpOffice[3] Version 2.0 and includes TerpWord (a small word-processor), TerpSpreadSheet (a spreadsheet application), TerpPaint (an image editing/manipulation program), TerpCalc (a scientific calculator with graphing capability), and TerpPresent (a presentation tool). They have been implemented using Java. Table 1 summarizes the characteristics of these applications. Note that these applications are fairly large with complex GUIs. With the exception of TerpCalc, all the applications are roughly the size of MS WordPad. The number of widgets listed in the table are the ones on which system-interaction events can be executed (i.e., text-labels are not included). The LOC are the number of statements in the programs. The `Help` menu is also not included since the help application is launched in a separate Web browser. Most of the code

3. http://www.cs.umd.edu/users/atif/TerpOffice.

TABLE 2
Total Number of System-Interaction Event Sequences

	Length		
Subject Application	1	2	3
TerpWord	112	1253	16012
TerpSpreadSheet	145	3246	92404
TerpPaint	200	8699	632040
TerpCalc	82	6561	524800
TerpPresent	294	19918	2293752
TOTAL	833	39677	3559008

TABLE 3
Number of Smoke Tests Generated

	Length			
Subject Application	1	2	3	Total
TerpWord	112	1253	3880	5245
TerpSpreadSheet	145	3246	2318	5709
TerpPaint	200	8699	0	8899
TerpCalc	82	6561	0	6643
TerpPresent	294	10998	0	11292
TOTAL	833	30757	6198	37788

written for the implementation of each application is for the GUI. None of the applications have complex underlying "business logic." This property of the subject application is especially important for seeding GUI faults (discussed later) since almost the entire code is for the GUI; there is no need to distinguish between GUI-code and business-logic-code during fault seeding.

For each application, DART generated the event-interaction graphs. The sizes of the event-interaction graphs in terms of nodes and edges are shown in columns labeled as "length 1" and "length 2," respectively, in Table 2.

6.2 Study 1: Fault Detection Effectiveness of Smoke Tests

The goals of the first study are to determine the number and types of faults that can be detected by the smoke test cases, characteristics of code that is missed, and the time/space required to execute the smoke test cases. The study involved performing the following steps:

1. For each subject application, generate smoke test cases and associated oracle information.
2. Use fault seeding techniques to artificially seed predetermined classes of faults in the subject applications.
3. Execute all test cases on the subject applications. During execution, compare the actual GUI state to the oracle information.

The following information is recorded in this study: number and type of faults detected, code coverage of the test cases, and the time required to execute the test cases and space required to store them.

6.2.1 Test Cases

DART's test-case generator used the event-interaction graphs to automatically generate 5,000-11,000 smoke test cases for each application. The exact number of test cases for each application, specified by the test designer, is shown in Table 3. The table shows, for each application, the total number of test cases that were *actually generated*. Not all length 3 test cases were generated for some applications since DART would not be able to run them on one machine in one night, thus defeating the purpose of smoke testing. The total number of test cases for all five applications was 833, 30,757, and 6,198 of lengths 1, 2, and 3, respectively. Note that (compared to Table 2) roughly half of all possible length 2 test cases were generated for TerpPresent; this application contains a large number (approximately 50) of system-interaction events that are used to change the text font. All but one of these events were eliminated from the event-interaction graph. In practice, a tester may edit the event-interaction graph to reduce the number of smoke test cases, thereby reducing testing cost.

6.2.2 Oracle Information

The automated test-oracle generator obtained the oracle information. As discussed earlier, this module of DART automatically executes test cases on the software and captures its state (widgets, properties, and values) automatically. By running this tool on our five subject applications for all the smoke test cases, all levels of oracle information were obtained.

Table 4 shows the space required to store the oracle information for each application. As noted earlier, a space-saving strategy may be used to reduce the storage required in this particular study. However, this strategy does not apply in all situations, e.g., if the start state for each test case is different. Hence, the numbers reported here are without any space-saving. As expected, the size increases from L1 to L4. Most applications (except TerpCalc) have very few invisible windows, which is why the numbers for L3 and L4 are very similar. TerpCalc contains many more invisible windows, which is why the space required for L4 is much larger than that for L3. L4 is 3-4 times larger than L5 since for each event, the oracle information is stored several times, including once before the first event.

6.2.3 Fault Seeding

Fault seeding is a well-known technique used to evaluate fault detection techniques. During fault seeding, classes of known faults are identified, and several instances of each fault class are artificially introduced into the subject program code at relevant points. Care is taken so that 1) the artificially seeded faults are similar to faults that naturally occur in real programs due to mistakes made by developers, 2) faults are seeded in code that is covered by an adequate number of test cases, e.g., they may be seeded in code that is executed by more than 20 percent and less than 80 percent of the test cases, and 3) faults are seeded "fairly," i.e., an adequate number of instances of each fault type is seeded.

TABLE 4
Space Required for Each Oracle

Subject Application	L1(MB)	L2 (MB)	L3 (MB)	L4 (MB)	L5 (MB)
TerpWord	0.21	7.63	13.71	13.85	2.78
TerpSpreadSheet	0.24	12.63	25.04	29.49	7.09
TerpPaint	0.29	29.89	34.61	34.91	10.12
TerpCalc	0.34	33.19	33.25	44.22	13.38
TerpPresent	0.45	81.39	100.03	108.13	18.92
TOTAL	1.53	164.73	206.64	230.6	52.29

Identifying Realistic Fault Classes. To identify realistic fault types, a history-based approach was adopted, i.e., *"real" GUI faults in TerpOffice* were observed. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[4] 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.

The following fault classes were chosen for this study:

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

Seeding via Code Coverage. When test cases were being executed on the subject applications for oracle-information collection, the code coverage of the test cases was also recorded. Blocks of the code that were covered by more than 20 percent of the test cases and less than 80 percent were identified. These blocks were candidates for fault seeding. Roughly 30-40 percent of the total code was available for fault seeding.

Seeding Faults Fairly. The candidate parts of the code were examined manually and the number of opportunities that they provided for seeding each type of fault was obtained. Since a total of 200 faults were to be seeded in each application, for each fault class i, $((f_i/\mathcal{F}) \times 200)$ instances were seeded, where f_i is the number of available opportunities to seed fault class i, and $\mathcal{F}$ is the sum of all opportunities for all fault classes. This explains why more instances of one fault class than others were seeded. For example, a total of 485 instances of fault Type 1, i.e., *modify relational operator*, were seeded because of the large number of relational operators in the subject applications.

Several graduate students were employed to seed the 200 faults in each subject application, thereby creating 200 faulty versions for each application. Exactly one fault was seeded in each version. This model is useful to avoid fault-interaction, which can be a challenging problem in these types of experiments. To determine the number of test case failures, the number of mismatches between the executing GUI's state and the oracle information were counted. The "number of faults" was determined by tracing the failures to the seeded faults.

Fig. 6 shows the distribution of the 1,000 faults that were seeded in the subject applications. The x-axis shows the type/class of fault, as defined earlier, and the total height of the columns shows the number of faults seeded.

4. http://bugs.cs.umd.edu.

TABLE 5
Total Execution Time

Subject Application	Total Execution Time (sec)
TerpWord	416047
TerpSheet	309411
TerpPaint	129756
TerpCalc	120200
TerpPresent	282407
TOTAL	1257821

6.2.4 Test Executor

All the smoke tests were executed on all 200 versions of the subject applications. When each application was being tested, its runtime state was extracted and compared with the stored oracle information. The specific parts of the state that were compared and the frequency of comparision was determined by the test oracle. A mismatch was reported as a failure. Some widget properties, such as positions, width, and height, were ignored during this process since the windowing system launches the software in a different screen location each time it is invoked. A few widgets, such as the "log window" were also ignored for some applications; during replay, this log window displays the current time and system memory usage. Since these values are dynamic, mismatches were anticipated.

Each test case required approximately 5 seconds to execute if it did not crash the software. The time varied by application and the number of GUI events in the test case. The total execution time for each application is shown in Table 5. The execution included launching the application under test, replaying GUI events from a test case on it and analyzing the resulting GUI states. The analysis consisted of recording the actual GUI states of the faulty version and determining the result of the test case execution. The test cases executed on four machines (Pentium 4, 2.2GHz, each with 256MB RAM). Although much of the execution was automated, some machines (and test scripts) had to be restarted because of problems with the JVM.

Recall from Section 5 that five types of test oracles, i.e., L1 through L5 were used in this study. DART was unable to accurately measure the time for L1 because the Java Swing API did not allow direct access to the current widget and its properties. The artificial implementation of L1 therefore required accessing the active window, traversing the internal Swing representation of the active window, locating the current widget, and then examining its properties. The time recorded was more than that required for L2, which is misleading. Hence, the time required for L1 is not reported here (in any case, it was only slightly larger than the time needed for L2). The results are summarized in Table 6. The results clearly show that L2 and L5 were much cheaper than L3 and L4.

6.2.5 Threats to Validity

Threats to external validity are conditions that limit the ability to generalize the results of experiments to industrial practice. Several such threats are identified in this study. First, five GUI-based Java applications have been used as subject programs. Although they have different types of GUIs, this does not reflect a wide spectrum of possible GUIs

TABLE 6
Time Required for Each Oracle

Subject Application	L2 (sec)	L3 (sec)	L4 (sec)	L5 (sec)
TerpWord	22.933	51.526	78.293	32.01
TerpSpreadSheet	18.009	33.206	59.588	21.263
TerpPaint	274	622.739	1034.536	398.611
TerpCalc	89.317	103.612	126.543	44.104
TerpPresent	47.494	79.458	158.2	15.952
TOTAL	451.753	890.541	1457.16	511.94

that are available today. Moreover, the applications are extemely GUI-intensive, i.e., most of the code is written for the GUI. The results will be different for applications that have complex underlying business logic and a fairly simple GUI. Second, all the subject programs were developed in Java by students, which might be more bug-prone than software implemented by professionals. Finally, although the 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. Every effort was made to seed faults that were as close as possible to naturally occurring faults. A history-based approach was used for seeding faults in the GUI applications. This may have affected the detection of faults by the test cases. Faults that are not manifested on the GUI will go undetected.

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 the measures of cost is time. Since GUI programs interact with the windowing system's manager, the execution time of an event 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. Since each event is executed several times (at least 80 times in different test cases), this threat has been adequately handled.

The results of the study, presented next, should be interpreted keeping in mind the above threats to validity. The same threats also apply to the second study.

6.2.6 Results and Discussion

Number of Faults Detected. The column graph in Fig. 5 shows the number of faults detected by the smoke test suite. The x-axis shows the subject applications and the y-axis shows the number of faults detected for all the test oracles combined. The figure shows that, with the exception of TerpSpreadSheet, the smoke tests detected a large number of faults.

As noted earlier, spreadsheet cells are a custom-designed widget class, not a part of Java Swing. The GUI Ripper had not been extended for the spreadsheet. Consequently, the test oracle did not examine the attributes (e.g., contents, style, and font) of the individual cells of the spreadsheet, thereby leading to missed faults. In the future, specialized test oracles for TerpSpreadSheet will be developed to detect these missed faults. These test oracles will be domain dependent, i.e., they will be designed specifically for the cells of TerpSpreadSheet.

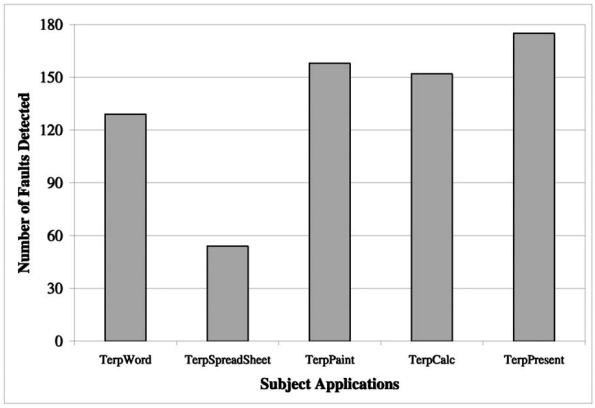

Fig. 5. Number of faults detected.

Result 1. This study showed that the smoke testing process is practical, both in terms of execution time, space required, and manual effort. The manual effort involved selecting the number of smoke test cases to generate and execute (using Table 3).

Classes of Faults Detected. One of the original goals of this study was to examine the classes of faults that the smoke test cases detect. Thus, the number of faults found, classified by fault type, was computed. Fig. 6 summarizes the results. The lightly shaded part of each column shows the number of faults detected; the full column height shows the total number of faults seeded. This result showed that the smoke tests were able to detect all types of faults that were seeded except for fault-type 8 (modify bit operator). To determine why fault-type 8 was missed, the code was manually examined. The problem was again with the test oracle—the seeded faults modified the text font style, which had not been included in the state information. In the future, customized test oracles that examine these text attributes will be developed.

Note that this was an encouraging result since all types of faults were not expected to be manifested on the GUI. As the column graph shows, there were instances of certain types of faults that could not be detected. The missed faults were in code that was not covered by the test cases.

Parts of Code Missed by the Smoke Tests. The code coverage report was manually examined to find why some

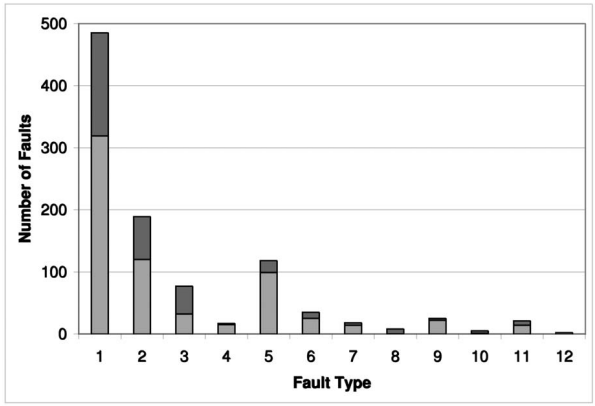

Fig. 6. Classes of faults seeded and detected.

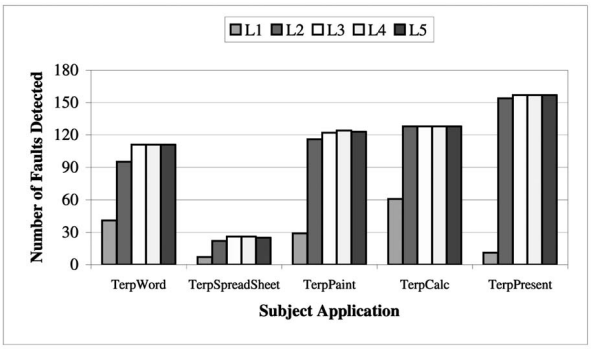

Fig. 7. Number of faults detected for each oracle.

parts were not covered. First, DART was unable to execute code related to those widgets that were not ripped by the GUI Ripper (e.g., the close button in all windows). Second, the smoke test cases did not cause any exceptions; exception handlers accounted for a significant percentage of missed code. For example, there are 44, 69, 73, 74, and 106 exception handlers in TerpWord, TerpSpreadSheet, TerpCalc, TerpPaint, and TerpPresent, respectively, which take around 5-10 percent of the code. Third, since the test cases are replayed using an API that directly communicates with the application, mouse and keyboard events are not generated during replay. Event handlers (e.g., right-click event handler) for such events are not executed. Since TerpPresent is a presentation tool with many events for releasing and moving the mouse, the test cases did not execute a significant part of the code of TerpPresent. Fourth, since the test cases are executed in a controlled environment, i.e., the environment variables (e.g., list of recently accessed files) are reset before executing each test case. Code related to these variables is never executed. Finally, there are events in the GUI that are enabled only after some other event sequence has been executed. If the required event sequence is longer than three events, the smoke test cases cannot execute the code associated with the disabled events.

Result 2. This study showed that the smoke tests cannot cover certain parts of the application code. Future work on DART will develop techniques to cover this code.

Effect of Test Oracle. DART's inability to automatically detect the TerpSpreadSheet faults reinforced our original hypothesis that test oracles have an impact on the overall effectiveness of the smoke test cases. It is expected that the test oracles would also have an impact on the number of faults detected by the test suites. Fig. 7 breaks down Fig. 5 by test oracle type. It shows five columns per application; each column represents the number of faults found. The graph indicates that L1 is the least effective, L2 did much better than L1, and L3, L4, and L5 did equally well.

To determine if the observed difference between test oracles was statistically significant, a one-way ANOVA (analysis of variance [37], [38]) with factor being the oracle level and response being the number of faults was conducted. The ANOVA test would indicate, with a certain degree of confidence, that the observed differences were statistically significant. The observed P-value was less than 0.05, leading to the conclusion that the oracle level has a statistically significant impact on number of faults found.

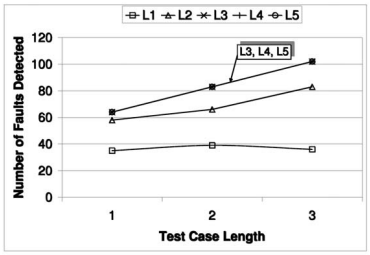

Fig. 8. Fault detection by Test Oracle (TerpWord).

The previous results suggested that the length and test oracle will also have a combined effect on the number of faults detected. Therefore, the number of faults detected by different-length test cases using different oracles was studied. The result for one typical application (TerpWord) is shown in Fig. 8. The x-axis shows the length of the test case, and the y-axis shows the number of faults detected. There are five lines in the graph, one for each oracle. The figure shows that length 1 test cases, combined with comprehensive oracles detected more faults than length 2 test cases that used L1. This pattern was seen in three out of five applications.

Result 3. This study showed that having comprehensive test oracles may make up for not having long test cases.

6.3 Study 2: Effect of Test Suite Size

In the above study, a fixed test-suite size was used for all applications. The impact of test-suite size on the number of faults detected is now studied.

Since a large number of test cases and results of their execution on fault-seeded versions of the subject applications were available from the previous study, there was no need to regenerate and reexecute new test cases for this study. The effect of different-sized test suites could be simulated by treating the existing suite as a *test pool* and selecting different sized test suites fom them. More specifically, for each subject application, the test pool was used to create 800 test suites: 200 of size 100, 200 of size 500, 200 of size 1,000, and 200 of size 2,000. Each suite was obtained independently using random selection without replacement.

For each test suite, the value "number of faults detected" was computed. Also, since one goal of this study was to study the interplay between test suite size and oracle type, the fault information for each oracle type was also computed.

Effect on Number of Faults Detected. Since there are 200 test suites of each size, the results are shown in the form of boxplots. The boxplots provide a concise display of each distribution. The black square inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover the entire distribution. Fig. 9 summarizes the results for one typical application (TerpPresent). As the boxplots show, the number of faults detected grows with test suite size irrespective of oracle. However, the rate of growth slows from 1,000 test cases to 2,000 test cases. Although the medians show a clear increase, the overlap in distributions is significant. Since a finite number of faults (200) have been

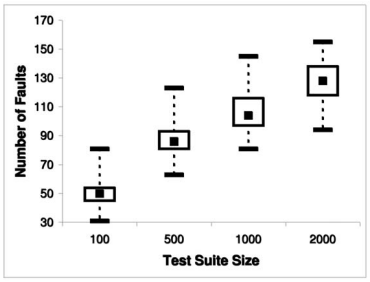

Fig. 9. Test suite size and faults (TerpPresent).

seeded, the number of faults detected will level-off after a certain test suite size.

Combined Effect of Test Oracles and Size on Number of Faults Detected. To study the interplay between test oracles and test suite size, the distributions of Fig. 9 are split into five parts, corresponding to the five test oracles. The results for a typical application (TerpPresent) are shown in Fig. 10.

These results give significant insight into the relationship between test suite size and test oracles. It is seen that even large test suites (2,000 test cases) with oracle L1 do much worse than the smallest suites (100 test cases) with oracle L2. Oracles L2 through L5 show very similar fault-detection ability for the same test suite size. Since L2 and L5 are cheaper than L3 and L4, it is better to use large test suites with these oracles for effective fault-detection.

Result 4. This study showed that using certain oracles can make up for not having large smoke test suites.

7 Conclusions

Previous work had presented the design of DART that automatically smoke tested GUI event interactions. This paper extended the previous work and demonstrated, via two empirical studies, several characteristics of GUI smoke tests. The results show that:

1. The entire smoke testing process is feasible in terms of execution time, storage space, and manual effort.
2. Smoke tests cannot cover certain parts of the application code.
3. Having comprehensive test oracles may make up for not having long smoke test cases.
4. Using certain oracles can make up for not having large smoke test suites.

In the empirical studies, all the manual steps required at most 1-2 minutes. For example, setup required identifying the Java class files for the subject applications and selecting a number of smoke tests to generate. During DART execution, the bug reports were examined. The bug reports that were produced by DART were extremely easy to read and understand, in that pinpointing the cause of the problem by examining the relevant event handler took 1-2 minutes. The coverage reports (not discussed in this paper due to lack of space) indicated blocks/branches of the code that needed to be tested using additional test cases. The test designer may manually create additional test cases using techniques discussed in Section 2. Overall, the manual effort required for the complete DART process is reasonable for frequent execution.

The current DART implementation has several limitations. First, as is the case with all automated regression testing techniques, some of the bugs reported by DART are *false positives*, i.e., they are not real bugs; they are simply a consequence of modifications made to the software. Second, the overall effectiveness of DART depends largely on the capabilities of the GUI ripper to obtain the state.

There are several new directions for this work. A custom test oracle is being developed for TerpSpreadSheet that will examine the contents of the individual cells, thereby helping to improve the fault-detection effectiveness of the smoke tests for that application. In the future, an interface for DART that will allow for the definition of such domain-specific test oracles will be developed. Fault injection techniques for GUIs will be studied and incorporated into the smoke test cases. The effects of the execution environment on the fault detection effectiveness of the smoke test cases will also be studied. Algorithms will be developed to partition large smoke test suites into several "nightly run" suites and rotating through them. False positives reported by GUI smoke test cases will be studied and techniques will be developed to identify and eliminate them. The GUI-based smoke test cases may be used together with code-based smoke tests; characteristics and strengths/weaknesses of both these types of test cases will be studied and mechanisms to combine them will be devised.

Acknowledgments

The authors would like to thank the anonymous reviewers whose comments and suggestions greatly helped to improve the presentation of this paper. The authors also thank Adithya Nagaragan and Ishan Banerjee who helped to lay the foundation for this research. This work was partially supported by a grant from the US National Science Foundation (CCF0447864). A preliminary report of this work appeared in the *Proceedings of the International Conference on Software Maintenence*, 2004.

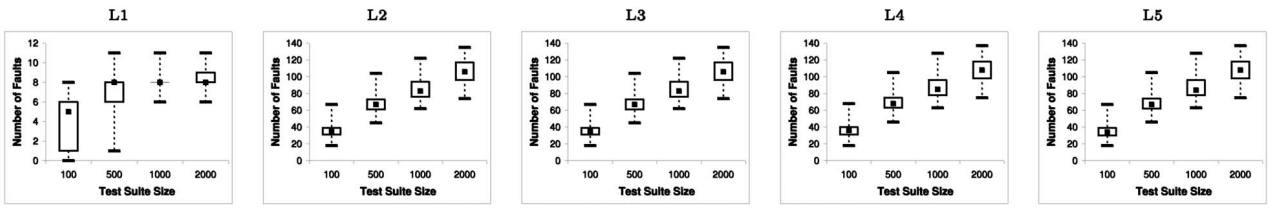

Fig. 10. Faults detected by Test Suite and Oracle (TerpPresent).

REFERENCES

[1] A.M. Memon and Q. Xie, "Empirical Evaluation of the Fault-Detection Effectiveness of Smoke Regression Test Cases for GUI-Based Software," *Proc. Int'l Conf. Software Maintenance 2004 (ICSM '04)*, pp. 8-17, Sept. 2004.

[2] A. Memon, A. Nagarajan, and Q. Xie, "Automating Regression Testing for Evolving GUI Software," *J. Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 1, pp. 27-64, 2005.

[3] A.M. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," *Proc. 10th Working Conf. Reverse Eng.*, pp. 260-269, Nov. 2003.

[4] A.M. Memon, M.E. Pollack, and M.L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 144-155, Feb. 2001.

[5] A.M. Memon, I. Banerjee, and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?," *Proc. IEEE Int'l Conf. Automated Software Eng.*, pp. 164-173, Oct. 2003.

[6] E.-A. Karlsson, L.-G. Andersson, and P. Leion, "Daily Build and Feature Development in Large Distributed Projects," *Proc. 22nd Int'l Conf. Software Eng.*, pp. 649-658, 2000.

[7] S. McConnell, "Best Practices: Daily Build and Smoke Test," *IEEE Software*, vol. 13, no. 4, pp. 143-144, July 1996.

[8] K. Olsson, "Daily Build—The Best of Both Worlds: Rapid Development and Control," technical report, Swedish Eng. Industries, 1999.

[9] J. Robbins, *Debugging Applications.* Microsoft Press, 2000.

[10] T.J. Halloran and W.L. Scherlis, "High Quality and Open Source Software Practices," *Meeting Challenges and Surviving Success: Second Workshop Open Source Software Eng.*, May 2002.

[11] B. Marick, "When Should a Test Be Automated?" *Proc. 11th Int'l Software/Internet Quality Week*, May 1998.

[12] L. Crispin, T. House, and C. Wade, "The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment," *Proc. Second Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pp. 96-104, 2001.

[13] "WINE Daily Builds," 2003, http://wine.dataparty.no/.

[14] "Mozilla," 2003, http://ftp.mozilla.org/.

[15] "Open WebMail," 2003, http://openwebmail.org/.

[16] "Cruise Control," 2003, http://cruisecontrol.sourceforge.net/.

[17] "FAST C++ Compilation—IcrediBuild by Xoreax Software," 2003, http://www.xoreax.com/main.htm.

[18] "Positive-g-Daily Build Product Information—Mozilla," 2003, http://positive-g.com/dailybuild/.

[19] "Kinook Software—Automate Software Builds with Visual Build Pro," 2003, http://www.visualbuild.com/.

[20] N. Baran, "Load Testing Web Sites," *Dr. Dobb's J. Software Tools*, vol. 26, no. 3, pp. 112, 114, 116, 118-119, Mar. 2001.

[21] S. Ellis, D. Johnson, M. Schmit, J. Jones, S. Cooke, and K. Granroth, "Letters: Open Source Cobol; Setting the Debian Record Straight; Back to Basics; Load Testing Web Sites; Open Source Hat Tricks; KDE Insider," *Dr. Dobb's J. Software Tools*, vol. 26, no. 7, pp. 10-12, July 2001.

[22] J. Weirich, "Using Perl to Check Web Links," *Linux J.*, vol. 36, Apr. 1997

[23] H. Berghel, "Using the WWW Test Pattern to Check HTML Client Compliance," *Computer*, vol. 28, no. 9, pp. 63-65, Sept. 1995.

[24] B. Marick, "Bypassing the GUI," *Software Testing and Quality Eng. Magazine*, pp. 41-47, Sept. 2002.

[25] M. Finsterwalder, "Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment," *Proc. Second Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.*, pp. 114-117, May 2001.

[26] L. White, H. AlMezen, and N. Alzeidi, "User-Based Testing of GUI Sequences and Their Interactions," *Proc. 12th Int'l Symp. Software Reliability Eng.*, pp. 54-63, 2001.

[27] "JUnit, Testing Resources for Extreme Programming," http://junit.org/news/extension/gui/index.htm, 2004.

[28] J.H. Hicinbothom and W.W. Zachary, "A Tool for Automatically Generating Transcripts of Human-Computer Interaction," *Proc. Human Factors and Ergonomics Society 37th Ann. Meeting*, p. 1042, 1993.

[29] "Capture-Replay Tool," 2003, http://soft.com.

[30] "Mercury Interactive WinRunner," 2003, http://www.mercuryinteractive.com/products/winrunner.

[31] "Abbot Java GUI Test Framework," 2003, http://abbot.sourceforge.net.

[32] "Rational Robot," 2003, http://www.rational.com.ar/tools/robot.html.

[33] A.M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," PhD thesis, Dept. of Computer Science, Univ. of Pittsburgh, July 2001.

[34] "Java Test Coverage Analyzer," http://www.codework.com/JCover/product.html, 2004.

[35] J. Su and P.R. Ritter, "Experience in Testing the Motif Interface," *IEEE Software*, vol. 8, no. 2, pp. 26-33, Mar. 1991.

[36] P.A. Vogel, "An Integrated General Purpose Automated Test Environment," *Proc. Int'l Symp. Software Testing and Analysis*, T. Ostrand and E. Weyuker, eds., pp. 61-69, June 1993.

[37] J.D. Jobson, *Applied Multivariate Data Analysis Volume 1: Regression and Experimental Design.* Springer, 1991

[38] H. Sahai and M. Ageel, *The Analysis of Variance: Fixed, Random and Mixed Models.* Birkhauser, 2000.

Atif M. Memon received the BS, MS, and PhD degrees in computer science in 1991, 1995, and 2001, respectively. He is an assistant professor in the Department of Computer Science, University of Maryland. He was awarded fellowships from the Andrew Mellon Foundation for his PhD research. He received the US National Science Foundation CAREER award in 2005. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is a member of the ACM and the IEEE Computer Society.

Qing Xie received the BS degree from South China University of Technology, Guangzhou, China, in 1996. She recieved the MS degree in computer science from the University of Maryland in 2003. She is currently a PhD student in the Department of Computer Science at the University of Maryland. Her research interests include software testing, software maintenance, mutation techniques, and empirical studies. She is a student member of the ACM, the IEEE and the IEEE Computer Society.