DART: A Framework for Regression Testing "Nightly/daily Builds" of GUI Applications

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

"Nightly/daily building and smoke testing" have become widespread since they often reveal bugs early in the software development process. During these builds, software is compiled, linked, and (re)tested with the goal of validating its basic functionality. Although successful for conventional software, smoke tests are difficult to develop and automatically rerun for software that has a graphical user interface (GUI). In this paper, we describe a framework called DART (Daily Automated Regression Tester) that addresses the needs of frequent and automated re-testing of GUI software. The key to our success is automation: DART automates everything from structural GUI analysis, test case generation, test oracle creation, to code instrumentation, test execution, coverage evaluation, regeneration of test cases, and their re-execution. Together with the operating system's task scheduler, DART can execute frequently with little input from the developer/tester to retest the GUI software. We provide results of experiments showing the time taken and memory required for GUI analysis, test case and test oracle generation, and test execution. We also empirically compare the relative costs of employing different levels of detail in the GUI test cases.

1. Introduction

"Nightly/daily builds and smoke tests" [19, 23, 32] have become widespread [34] as many software developers find them useful [15]. Popular software that use daily nightly/builds include *WINE* [1], *Mozilla* [2], *AceDB* [3], and *openwebmail* [4]. During nightly builds, software is compiled, linked and "smoke tested" ("smoke tests" are also called "sniff tests" or "build verification suites" [21]). Typically *unit tests* [34] and sometimes *acceptance tests* [12] are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system [21]. The smoke tests exercise the entire system; they don't have to be an exhaustive test suite but they should be capable of detecting major problems. A build that passes the smoke test is considered "a good build". Bugs are reported, usually in the form of e-mails to the developers [34], who can quickly resolve the bugs. This practice is a useful quality assurance mechanism to catch defects early in software systems that are typically being simultaneously modified by several developers. Frequent building and re-testing is also gaining popularity because new software development processes (such as extreme programming [14, 38]) advocate a tight development/testing cycle [33]. A number of tools support daily builds; some of the popular tools include *CruiseControl* [5], *IncrediBuild* [6], *Daily Build* [7], and *Visual Build* [8].

A limitation of current nightly builds is inadequate testing and re-testing of software that has a **graphical user interface** (GUI).[1] Frequent and efficient re-testing of conventional software has leveraged the strong research conducted for automated regression testing [36], which is a software maintenance activity, done to ensure that modifications have not adversely affected the software's quality [36]. Although there has been considerable success in developing techniques for regression testing of conventional software [11, 35], regression testing of GUIs has been neglected [24]. Consequently, there are no automated tools and efficient techniques for GUI regression testing [25].

Not being able to adequately test GUIs has a negative impact on overall software quality because GUIs have become nearly ubiquitous as a means of interacting with soft-

[1] Note that we focus on testing the functionality of the GUI, not *usability* [37] issues such as user-friendliness.

ware systems. GUIs today constitute as much as 45-60% of the total software code [31]. Currently, three popular approaches are used to handle GUI software when performing nightly builds. First, and most popular, is to perform no GUI smoke testing at all [21], which either leads to compromised software quality or expensive GUI testing later. Second is to use test harnesses that call 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 [22]), it also does not perform testing of the end-user software. Third is to use existing tools to do limited GUI testing [13, 20]. Examples of some tools used for GUI testing include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[2] and capture/replay tools [16] such as WinRunner[3] that provide very little automation [26], especially for *creating* smoke tests. Developers/testers who employ these tools typically come up with a small number of smoke tests [23].

In this paper, we describe a new framework called DART (*Daily Automated Regression Tester*) that addresses the needs of re-testing frequent builds of GUI software. The key to the success of DART is automation. DART automates everything from structural GUI analysis (which we refer to as *GUI ripping*), test case generation [29, 27], test oracle creation [28], and code instrumentation to test execution, coverage evaluation [30], regeneration of test cases, and their re-execution. Together with the operating system's task scheduler (e.g., Unix cron job), DART can execute frequently with little input from the developer/tester to smoke test the GUI software. We provide results of experiments showing the time taken by the ripper, test case generator, test oracle generator, and test executor. We also empirically compare the relative costs (in terms of time and memory) of employing different levels of oracle information for re-testing.

The important contributions of the method presented in this paper include the following.

- We define a formal model of a GUI derived from specifications that is useful for smoke testing. In this paper we demonstrate its usefulness in developing an efficient and automated regression tester that can be run daily.
- We develop a new process for re-testing nightly builds of GUI software.
- Our regression testing process can not only be used for nightly builds but for general GUI re-testing.
- We show our re-testing process as a natural extension of our already implemented GUI testing tools [26, 28, 29, 25, 30, 27, 24].

[2] `http://junit.org/news/extension/gui/index.htm`
[3] `http://mercuryinteractive.com`

Phase	Step	Developer/tester	DART
Identification	1	Identify AUT	
Analysis	2		Rip AUT's GUI
	3	Verify and modify structure	
Test Adequacy Definition	4		Create event-flow graphs and integration tree
	5		Create matrix M
	6	Define M'	
Test Generation	7		Generate test cases
	8		Generate expected output
Modification	9	Modify AUT	
Regression Testing	10		Instrument code
	11		Execute test cases and compare with expected output
	12		Generate execution report
	13		Generate coverage report
	14		E-mail reports
Analysis and Regeneration	15	Examine reports and fix bugs	
	16	Modify M' if needed	
	17		Generate additional test cases
	18		Generate additional expected output

Table 1. Roles of the Developer/tester and DART.

In the next section, we describe the process employed by DART for GUI re-testing. In Section 3, we present details of the design of DART. Results of experiments in Section 4 show that DART is efficient enough for frequent re-testing. We discuss related research and practice in Section 5 and finally conclude in Section 6 with a discussion of ongoing and future work.

2. The DART Process

In this section we present the steps of the DART process. The goal is to provide the reader with a high-level picture of the operation of DART and highlight the role of the developer/tester in the overall process. Details of technologies used to develop DART are given in Section 3. Some of the terms used here will be formally defined later. These steps are also summarized in Table 1.

1. The developer identifies the application under test (AUT). This essentially means that the source files and executables are identified.
2. DART automatically analyzes the AUT's GUI structure by a dynamic process that we call *GUI ripping*.

Matrix M	Test Case Length			
Component Name	1	2	3	4
Main	56	791	14354	255720
FileOpen	10	80	640	5120
FileSave	10	80	640	5120
Print	12	108	972	8748
Properties	13	143	1573	17303
PageSetup	11	88	704	5632
FormatFont	9	63	441	3087

Figure 1. Matrix M for MS WordPad.

Matrix M'	Test Case Length			
Component Name	1	2	3	4
Main	56	791	50	0
FileOpen	10	80	80	0
FileSave	10	80	70	0
Print	12	108	0	0
Properties	13	143	0	0
PageSetup	11	88	25	0
FormatFont	9	63	400	0

Figure 2. Matrix M' for MS WordPad.

It automatically traverses all the windows of the GUI, identifies all the GUI objects and their properties, and saves the extracted structure in an XML file.

3. The developer then verifies the correctness of the structure and makes any needed changes by using an editing tool. The number of changes needed depend on the AUT and the implementation platform. Common examples include missed events and windows. The changes are saved so that they can be automatically applied to future versions of the AUT.
4. DART uses the GUI structure to create *event-flow graphs* and an *integration tree* [30] (Section 3). These structures are used in the next step and in Step 7 to automatically generate test cases and evaluate test coverage.
5. The developer is then presented with a matrix $M(i, j)$, where i is a *GUI component* (a *modal* dialog with associated *modeless* windows; defined formally in Section 3) and j is the length of a test case. $M(i, j) = N$ means that N test cases of length j *can* be executed on component i. Although we advocate running at least all test cases of length 1 and 2 for smoke testing, the developer is free to choose test cases of any length. An example of such a matrix for MS WordPad is shown in Figure 1. The rows show the components of the WordPad GUI and columns show the length of the test cases.
6. The developer creates a new matrix $M'(i, j)$; the entries of M' specify the number of test cases of length j that *should* be executed on component i. The developer needs to fill in the required number of test cases, a task that typically requires a few minutes. An example is seen in Figure 2. Note that the test designer has chosen to not generate any length 4 test cases, indicated by "0" in Column 4.
7. DART uses an automated test case generator to generate the smoke test cases.
8. A test oracle generator is used to automatically create an expected output for the next version of the AUT. The *smoke test suite* for subsequent versions is now ready.
9. The development team modifies the AUT.
10. The operating system's task scheduler launches DART, which in turn launches the AUT. DART automatically instruments the AUT's source code. A code instrumenter (e.g., Instr [9]) is used to instrument the code. This code is executed during testing to gather code coverage information.
11. Test cases are executed on the AUT automatically and the output is compared to the stored expected output.
12. An execution report is generated in which the executed test cases are classified as *successful* or *unsuccessful*.
13. Two types of coverage reports are generated: (1) statement coverage showing the frequency of each statement executed, and (2) event coverage, reported as a matrix $C(i, j)$. The format of C is exactly like M', allowing direct comparison between M' and C. $C(i, j) = N'$ shows that N' test cases *were* executed on the AUT.
14. These results of the test execution are e-mailed to the developers.
15. The next morning, developers examine the reports and fix bugs. They also examine the unsuccessful test cases. Note that a test case may be unsuccessful because of (1) the expected output did not match the actual output. If the expected output is found to be incorrect, then a test oracle generator is used to automatically update the expected output for the modified AUT, or (2) an event in the test case had been modified (e.g., deleted) preventing the test case from proceeding. These test cases can no longer be run on the GUI and are deleted.
16. Using the coverage reports, the developers identify new areas in the GUI that should be tested. They modify M' accordingly.
17. The new test cases, and
18. oracle information are generated.

Steps 10 through 18 are repeated throughout the development cycle of the AUT.

Note that we do not mention test cases other than those generated for GUI testing. Additional test cases (such as code-based tests) can easily be integrated in the above cycle to improve overall test effectiveness.

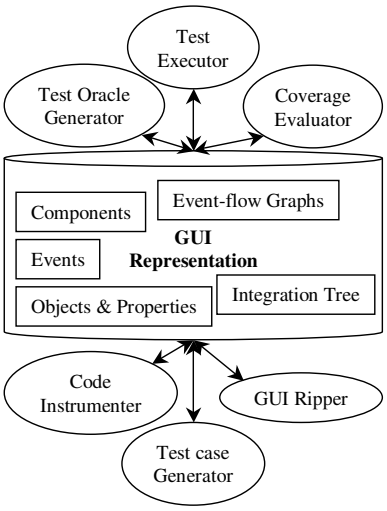


Figure 3. Modules of DART.

3. Design of DART

Before we discuss the details of the design of DART, we will first mention the requirements that provided the driving philosophy behind this design. We required that DART be:

- **automated** so that the developer's work is simplified. This is especially necessary for first-time generation of smoke test cases;
- **efficient** since GUI testing is usually a tedious and expensive process. Inefficiency may lead to frustration and abandonment;
- **robust**; whenever the GUI enters an unexpected state, the testing algorithms should detect the error state and recover so that the next test case can be executed;
- **portable**; test information (e.g., test cases, oracle information, coverage report, and error report) generated and/or collected on one platform should be usable on other platforms if the developers choose to change the implementation platform during development;
- **general** enough to be applicable to a wide range of GUIs.

Figure 3 shows the primary modules of DART and their interaction. The GUI representation is the "glue" that holds all modules together. All modules interact with each other via the representation. We first describe the representation and then briefly describe each module.

3.1. GUI Representation

The GUI representation is a formal model of the AUT's GUI. Note that the entire representation is extracted automatically from the implemented GUI.

Objects & Properties: A GUI is modeled as a set of *objects* $O = \{o_1, o_2, \ldots, o_m\}$ (e.g., `label`, `form`, `button`, `text`) and a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (e.g., `background-color`, `font`, `caption`). Each GUI will use certain types of objects with associated properties; at any specific point in time, the state of the GUI can be described in terms of all the objects that it contains, and the values of all their properties. Formally we define the state of a GUI as follows:

Definition: The *state* of a GUI at time t is the set P of all the properties of all the objects O that the GUI contains. □

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.

Events: 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. □

The function notation $S_j = e(S_i)$ is used to denote that S_j is the state resulting from the execution of event e in state S_i. Events may be stringed together into sequences. Of importance to testers are sequences that are permitted by the structure of the GUI. 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. □

An event sequence that is not legal is called an *illegal* event sequence. For example, since in MS Word, `Cut` (in the `Edit` menu) cannot be performed immediately after `Open` (in the `File` menu), the event sequence `Open`; `Cut` is illegal (ignoring keyboard shortcuts).

Components: GUIs, by their very nature, are hierarchical, and this hierarchy may be exploited to identify groups of GUI events that may be performed in isolation. One hierarchy of the GUI, and the one used in this research, is obtained by examining *modal windows* in the GUI, i.e., windows that once invoked, monopolize the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated. The language selection window in MS Word is

an example of a modal window. Other windows, also examined, in the GUI are called *modeless windows*[4] that do not restrict the user's focus; they merely expand the set of GUI events available to the user. For example, in MS Word, performing the event `Replace` opens a modeless window entitled `Replace`.

At all times during interaction with the GUI, the user interacts with events within a modal dialog. This modal dialog consists of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly by X. The modal dialog remains in place until X is explicitly terminated. Intuitively, the events within the modal dialog form a *GUI component*,[5]

Definition: A *GUI component* C is an ordered pair ($\mathcal{RF}$, $\mathcal{UF}$), where $\mathcal{RF}$ represents a modal window in terms of its events and $\mathcal{UF}$ is a set whose elements represent modeless windows also in terms of their events. Each element of $\mathcal{UF}$ is invoked either by an event in $\mathcal{UF}$ or $\mathcal{RF}$. □

Note that, by definition, events within a component do not interleave with events in other components without the components being explicitly invoked or terminated.

Event-flow Graphs: A GUI component may be represented as a flow graph. Intuitively, an *event-flow graph* (EFG) represents all possible interactions among the events in a component.

Definition: An *event-flow graph* for a component C is a 4-tuple $<$**V**, **E**, **B**, **I**$>$ where:

1. **V** is a set of vertices representing all the events in the component. Each $v \in$**V** represents an event in C.
2. **E** $\subseteq$ **V** $\times$ **V** is a set of directed edges between vertices. Event e_j `follows` e_i iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in$ **E** iff the event represented by v_y `follows` the event represented by v_x.
3. **B** $\subseteq$ **V** is a set of vertices representing those events of C that are available to the user when the component is first invoked.
4. **I** $\subseteq$ **V** is the set of events that invoke other components.

□

Note that an event-flow graph is not a state machine. The nodes represents events in the component and the edges show the `follows` relationship. An example of an event-flow graph for a part of the `Main`[6] component of MS WordPad is shown in Figure 4. At the top are three vertices (`File`, `Edit`, and `Help`) that represent part of the pull-down menu of MS WordPad. They are events that are available when the `Main` component is first invoked. Once `File` has been performed in WordPad, any of `Edit`, `Help`, `Open`, and `Save` events may be performed. Hence there are edges in the event-flow graph from `File` to each of these events. Note that `Open`, `About` and `Contents` are shown with dashed ovals. We use this notation for events that invoke other components, i.e., **I** = {`Open`, `About`, `Contents`}. Other events include `Save`, `Cut`, `Copy`, and `Paste`. After any of these events is performed in MS WordPad, the user may perform `File`, `Edit`, or `Help`, shown as edges in the event-flow graph.

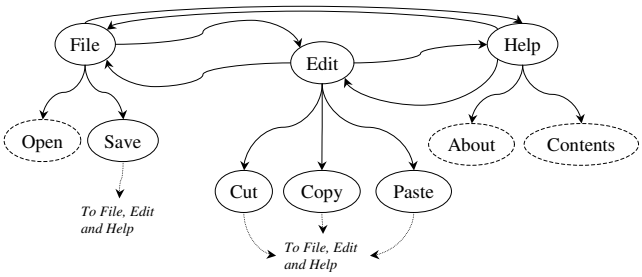


Figure 4. EFG for Part of MS WordPad.

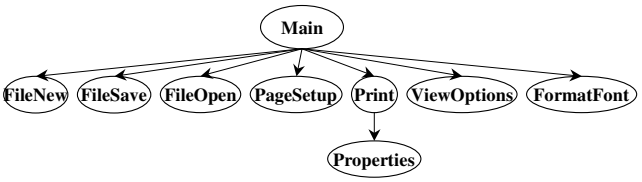


Figure 5. IT for Part of MS WordPad.

Integration Tree: Once all the components of the GUI have been represented as event-flow graphs, the remaining step is to construct an *integration tree* (IT) to identify interactions among components. These interactions take the form of *invocations*, defined formally as:

Definition: Component C_x **invokes** component C_y iff C_x contains an event e_x that invokes C_y. □

Intuitively, the integration tree shows the invokes relationship among all the components in a GUI. Formally, an integration tree is defined as:

Definition: An *integration tree* is a triple $< \mathcal{N}, \mathcal{R}, \mathcal{B} >$, where $\mathcal{N}$ is the set of components in the GUI and $\mathcal{R} \in \mathcal{N}$ is a designated component called the `Main` component. $\mathcal{B}$ is the set of directed edges showing the invokes relation between components, i.e., $(C_x, C_y) \in \mathcal{B}$ iff C_x **invokes** C_y. □

[4] Standard GUI terminology, e.g., see http://java.sun.com/products/jlf/ed2/book/HIG.Dialogs.html.

[5] GUI components should not be confused with *GUI widgets* that are the building blocks of a GUI.

[6] The component that is presented to the user when the GUI is first invoked.

Note that in general, the relationship among components may be represented by a dag, since multiple components may invoke a component. However, the dag can be converted into a tree by copying nodes. The tree model also simplifies our algorithms based on tree traversals of the integration tree. Figure 5 shows an example of an integration tree representing a part of the MS WordPad's GUI. The nodes represent the components of the GUI and the edges represent the invokes relationship between the components. Components' names indicate their functionality. For example, `FileOpen` is the component of WordPad used to open files. The tree in Figure 5 has an edge from `Main` to `FileOpen` showing that `Main` contains an event, namely `Open` (see Figure 4) that invokes `FileOpen`.

3.2. Modules of DART

Having presented a formal model of the GUI, we now briefly describe each module shown in Figure 3. Note that due to lack of space, we provide, for each module, only the details needed to understand this paper. Additional details and algorithms are available in previously reported literature [24, 28, 30, 29].

Test Case Generator: Our concepts of events, objects and properties 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 $e_1; e_2; \ldots; e_n$. □

We know from Section 3 that event-flow graphs and the integration tree represent legal sequences of events that can be executed on the GUI. To generate test cases, we start from a known initial state S_0 and use a graph traversal algorithm, enumerating the nodes during the traversal, on the event-flow graphs. Sequences of events $e_1; e_2; \ldots; e_n$ are generated as output that serve as a GUI test case $< S_0, e_1; e_2; \ldots; e_n >$.

Note that all test cases of length 1 and 2 execute all GUI events and all pairs of events. We recommend that the smoke test suite contain at least these test cases, although the final choice of smoke tests lies with the developer.

Test Oracle Generator: *Test oracles* are used to determine whether or not the software executed correctly during testing. They determine whether or not the output from the software is equivalent to the expected output. In GUIs, the expected output includes screen snapshots and positions and titles of windows. Our model of the GUI in terms of objects/properties can be used to represent the expected state of a GUI after the execution of an event.

For any test case $< S_0, e_1; e_2; \ldots e_n >$, the sequence of states $S_1; S_2; \ldots S_n$ can be computed by extracting the complete (or partial) state of the GUI after each event. Depending on the resources available, DART can collect and compare oracle information at the following different levels (LOI) of (decreasing) cost and accuracy.[7] Detailed comparison between these levels is given in Section 4.

Complete: $LOI1 = \{(w, p, o), \ \forall w \in Windows, \ \forall o = objects \in w, \ \forall p = properties \in o\}$, i.e., the set containing triples of all the properties of all the objects of all the windows in the GUI.

Complete visible: $LOI2 = \{(w, p, o), \ \forall (w \in Windows) \& (w \ is \ visible), \ \forall o = objects \in w, \ \forall p = properties \in o\}$, i.e., the set containing triples of all the properties of all the objects of all the *visible* windows in the GUI.

Active window: $LOI3 = \{(w, p, o), \ (w = active \, Window), \ \forall o = objects \in w, \ \forall p = properties \in o\}$, i.e., the set containing triples of all the properties of all the objects of the *active* window in the GUI.

Widget: $LOI4 = \{(w, p, o), \ (w = active \, Window), \ o = current \ object, \ \forall p = properties \in o\}$, i.e., the set containing triples of all the properties of the object in question in the active window.

In practice, a combination of the above may be generated for a given test case.

Coverage Evaluator: Although smoke tests are not meant to be exhaustive, we feel that coverage evaluation serves as a useful guide to additional testing, whether it is done for the next build or for future comprehensive testing. Also, our use of the matrix to specify test requirements is an intuitive way for the developer to specify smoke testing requirements and analyze testing results. In DART, two different types of coverage are evaluated – code based and event based. Code based coverage is the conventional statement/method coverage that requires the code to be instrumented by a code instrumenter. In addition, we employ a new class of coverage criteria called *event-based coverage criteria* to determine the adequacy of tested event sequences. The key idea is to define the coverage of a test suite in terms of GUI events and their interactions.

An important contribution of event-based coverage is the ability to intuitively express GUI testing requirements and examine test adequacy via a matrix. The entries of the matrix can be interpreted as follows:

Event Coverage requires that individual events in the GUI be exercised. These individual events correspond to length 1 event-sequences in the GUI. $\mathbf{Matrix}_{j,1}$, where $j \in S$, represents the number of individual events covered in each component.

[7] The need for these levels is explained in detail in earlier reported work [28].

		Generation			
		Complete (LOI1)	*Complete Visible (LOI2)*	*Active Window (LOI3)*	*Widget (LOI4)*
Execution	*Complete (LOT1)*	x			
	Complete Visible (LOT2)	x	x		
	Active Window (LOT3)	x	x	x	
	Widget (LOT4)	x	x	x	x

Figure 6. Possibilities Available to the Test Designer for Level of Detail of Oracle Information.

Event-interaction Coverage requires that all the edges of the event-flow graph be covered by at least one test case. Each edge is effectively captured as a length 2 event-sequence. $\mathbf{Matrix}_{j,2}$, where $j \in S$, represents the number of branches covered in each component **j**.

Length-n Event-sequence Coverage is available directly from **Matrix**. Each column **i** of **Matrix** represents the number of length-**i** event-sequences in the GUI.

Details of algorithms to compute the matrix are presented in earlier reported work [30]. We have already shown examples of matrices in Figures 1 and 2.

Test Executor: The test executor is capable of executing an entire test suite automatically on the AUT. It performs all the events in each test case and compares the actual output with the expected output. Events are triggered on the AUT using the native OS API. For example, the windows API *SendMessage* is used for windows application and Java API *doClick* for Java application.

The remaining question, then, is what properties should be compared. There are several possible answers to this question, and the decision amongst them establishes the *level of testing* (LOT1-LOT4) performed. These levels of testing correspond directly to the oracle information that was collected, i.e., complete, complete-visible, active-window, and widget. During test execution, depending on the resources available, the test designer may choose to employ partial oracle information, even though more detailed information may be available. For example, the test designer may choose to compare only the properties of the current widget even though the complete property set for all windows may be available. In fact, the test designer has the ability to execute at least 10 different such combinations. Figure 6 shows all these combinations, marked with an "x". Note that information cannot be used unless it has been generated, i.e., if LOI4 is available, then LOT1-LOT3 cannot be performed. We compare these combinations in an experiment in Section 4.

Subjects	Windows	LOC	Classes	Components
TerpPaint	8	9287	42	7
TerpSpreadsheet	6	9964	25	5
TerpPad	8	1747	9	5
TerpCalc	3	4356	9	3
TerpDraw	5	4769	4	3
TerpManager	1	1452	3	1
TOTAL	***31***	***31575***	***92***	***24***

Table 2. Our Subject Applications.

4. Experiments

Having presented the design of DART, we now examine its practicality using actual test runs and reporting execution time and memory requirements.

4.1. Open Questions

We identified the following three questions that needed to be answered to show the practicality of the process and to explore the cost of using different levels of testing.

1. How much time does DART take for complete smoke testing?
2. What is the additional cost (in terms of time and memory) of generating detailed test oracle information?
3. What is the additional cost of test execution when using detailed test oracle information?

To answer our questions we needed to measure the cost of the overall smoke testing process while controlling the details of the test oracle and the different levels of testing.

4.2. Subject Applications

For our study, we used six Java programs as our subjects. These programs were developed as part of an Open-Source office suite software[8]. Table 2 describes these subjects, showing the number of windows, lines of code, number of classes and components. Note that these are not toy programs. In all, they contain more than 30 KLOC, with at least two programs almost 10 KLOC.

4.3. Experimental Design

Variables: In the experiment, we manipulated three independent variables:

1. **P**: the subject programs (6 programs),
2. **LOI**: level of oracle information detail (4 levels: complete, complete visible, active window, widget),

[8]The software can be downloaded from `http://www.cs.umd.edu/users/atif/TerpOffice`

3. **LOT**: levels of testing (4 levels). Note that for a given test run, $LOI \geq LOT$, i.e., the information must be generated before it can be used.

On each run, with program P, levels LOI, levels LOT, we generated test information for 1000 test cases and measured the total generation time and memory required. We then executed all these test cases for each of the 10 possible LOI and LOT combinations (Figure 6). In all, for our complete experiment, we generated and executed $6 \times 10 \times 1000 = 60000$ test cases.

Threats to internal validity are influences that can affect the dependent variables without the researchers knowledge. Our greatest concerns are test case composition and platform-related effects that can bias our results. We have noticed that some events, e.g., file operations, take longer than others (e.g., events that open menus); hence a short test case with a file event may take more time than a long test case without a file event. Also, performance of the Java runtime engine varies considerably during test execution; the overall system slows down as more test cases are executed. The performance improves once the garbage collector starts. To minimize the effect of this threat we executed each test independently, completely restarting the JVM each time.

Threats to external validity are conditions that limit our ability to generalize the results of our experiment. We consider at least one source of such threats: artifact representativeness, which is a threat when the subject programs are not representative of programs found in general. There are several such threats in this experiment. All programs are written in Java and they were developed by students. We may observe different results for C/C++ programs written for industry use. As we collect other programs, we will be able to reduce these problems.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this experiment our measure of cost is CPU 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.

4.4. Results

The results of these experiments should be interpreted keeping in mind the above threats to validity.

Figure 7 shows the memory requirements for the six subject programs. $LOI0$ represents test cases that contain no oracle information. We had expected that the memory requirements would increase as the level of oracle detail increases. Figure 7 shows that the memory requirements grow very rapidly when using a detailed level of test oracle. Note that we are using a logarithmic scale to improve readability. The memory demands are not so serious for our smaller subject programs. However, the memory requirements become very high for our large programs (TerpPaint and TerpDraw) that contain a large number of windows. We also observe that many GUIs create multiple windows (few invisible to the user) after the software is launched. When a user "opens" a window, it is programatically made visible. Although good for speed, this practice imposes unnecessary demands on memory.

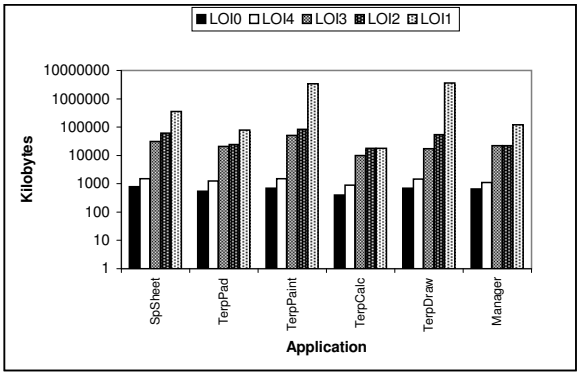
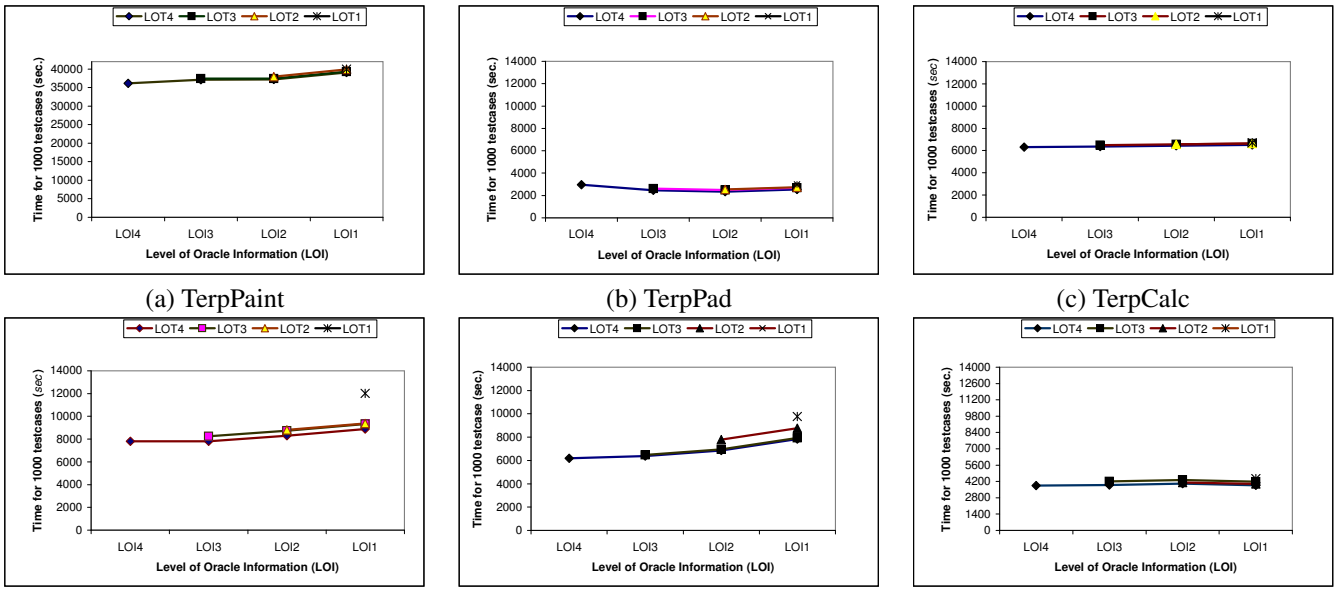


Figure 7. Memory Requirements of our Subject Applications for Different Level of Detail of Oracle Information.

Figure 8 shows the total execution time for all levels of detail of oracle information. We see that using all-windows is, in general, expensive compared to widget. As noted earlier, a test designer may choose a lower LOT even if a high LOI is available. Figure 8 shows the total execution times for all our subject programs for all possible combinations of LOI and LOT.

In these experiments, we note that the maximum time taken for generation and execution is that of TerpPaint with LOI = LOT = all-windows. The execution time consists of application launching time and test execution. It was observed that launching time dominates the test execution time. Also note that we have normalized the y-axis for all applications except TerpPaint. We discovered that the splash screen in TerpPaint contains an artificial delay of a few seconds, leading to increased execution time. Hence the times obtained for TerpPaint are skewed.

The total time required to set up DART is usually a few minutes. The developer interaction is mainly required for inspecting and validating the analyzed GUI structure, and also to fill in the M' matrix (Section 2). The rest of the process is automated. The total time to execute 1000 test cases is less than 10 hours for each of these applications.



(a) TerpPaint (b) TerpPad (c) TerpCalc

(d) TerpSpreadSheet (e) TerpDraw (f) TerpManager

Figure 8. Total Execution Times for Individual Subject Applications for Different LOI and LOT.

This makes the process suitable for nightly execution.

5. Related Work

Daily building and smoke testing has been used for a number of large-scale projects, both commercial and Open-Source. For example, Microsoft used daily builds extensively for the development of its popular Windows NT operating system [23]. The GNU project[9] continues to use daily builds for most of its projects. While there are many projects that use daily builds, we found very little reported literature on techniques and tools for daily builds and smoke tests and none for GUI software.

A closely related paper discusses automating acceptance tests for GUIs in an extreme programming environment [13] in which frequent testing of the software is imperative to the overall development process. Programmers create tests to validate the functionality of the software and whether the software conforms to the customer's requirements. These tests are run often, at least once a day [13, 10]. Hence, there is a need to automate the development of re-usable and robust tests. One approach is to implement a framework-based test design [13, 18]; scripts that control the function call are created manually using a capture/replay tool. Another popular method for testing of GUIs in XP environments is the use of xUnit frameworks, such as jUnit and jfcUnit. GUI widgets are accessed from the GUI and tested for existence and functionality [17]. Even with limited automation, the tests have to be written manually and testing GUI functionality becomes complex. Furthermore, these tests are intensely data-driven and very fragile. A variable name change is all that is necessary to break the test.

[9] `http://www.gnu.org`

6. Conclusions and Future Work

Nightly builds and smoke tests have become widespread and they are useful to reveal bugs early in the software development process. Although successful for conventional software, smoke tests are difficult to develop and automatically rerun for software that has a GUI. In this paper, we presented a technique for re-testing software that has a GUI. We empirically demonstrated that the technique is practical and may be used for smoke testing nightly/daily builds of GUI software.

We have implemented our technique in a system called DART. DART is efficient enough to be used for any type of frequent GUI re-testing. Our technique is not restricted to smoke testing of nightly builds only. In the future, we will extend DART to generate and execute tests other than smoke tests. The GUI smoke tests are not meant to replace other code-based smoke tests. However, DART is a valuable tool to add to the tool-box of the tester/developer.

In the future, the effectiveness of the DART process will be studied by analyzing the number of faults detected.

References

[1] http://wine.dataparty.no/.

[2] http://ftp.mozilla.org/pub/mozilla/nightly/latest/.

[3] http://www.acedb.org/Software/Downloads/daily.shtml.

[4] http://openwebmail.org/openwebmail/download/redhat/rpm/daily-build/.

[5] http://cruisecontrol.sourceforge.net/.

[6] http://www.xoreax.com/main.htm.

[7] http://positive-g.com/dailybuild/.

[8] http://www.visualbuild.com/.

[9] http://www.glenmccl.com/instr/instr.htm.

[10] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[11] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, Aug. 1997.

[12] L. Crispin, T. House, and C. Wade. The need for speed: automating acceptance testing in an extreme programming environment. In *Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 96–104, 2001.

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

[14] J. Grenning. Launching extreme programming at a process intensive company. *IEEE Software*, 18:27–33, 2001.

[15] T. J. Halloran and W. L. Scherlis. High quality and open source software practices. In *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, May 2002.

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

[17] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2001.

[18] C. Kaner. Improving the maintainability of automated test suites. In *Proceedings of The 10th International Software/Internet Quality Week*, 1997.

[19] E.-A. Karlsson, L.-G. Andersson, and P. Leion. Daily build and feature development in large distributed projects. In *Proceedings of the 22nd international conference on Software engineering*, pages 649–658. ACM Press, 2000.

[20] H. A. Lee White and N. Alzeidi. User-based testing of gui sequences and their interactions. In *Proceedings of the 12th International Symposium Software Reliability Engineering*, pages 54 – 63, 2001.

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

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

[23] S. McConnell. Best practices: Daily build and smoke test. *IEEE Software*, 13(4):144, 143, July 1996.

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

[25] A. M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):90–91, Aug. 2002.

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

[27] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.

[28] 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.

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

[30] 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.

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

[32] K. Olsson. Daily build - the best of both worlds: Rapid development and control. Technical report, Swedish Engineering Industries, 1999.

[33] C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *IEEE Software*, 18:42–50, 2001.

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

[35] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, Mar. 1997.

[36] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[37] M. C. Salzman and S. D. Rivers. Smoke and mirrors: Setting the stage for a successful usability test. *Behaviour and Information Technology*, 13(1/2):9–16, 1994.

[38] P. Schuh. Recovery, redemption and extreme programming. *IEEE Software*, 18:34–41, 2001.